Otros títulos publicados:

1. Ingeniería de Sistemas. *Benjamin S. Blanchard.*
2. La Teoría General de Sistemas. *Ángel A. Sarabia.*
3. Dinámica de Sistemas. *Javier Aracil.*
4. Dinámica de Sistemas Aplicada. *Donald R. Drew.*
5. Ingeniería de Sistemas Aplicada. Isdefe.
6. CALS (Adquisición y apoyo continuado durante el ciclo de vida). *Rowland G. Freeman III.*
7. Ingeniería Logística. *Benjamin S. Blanchard.*
8. Fiabilidad. *Joel A. Nachlas.*
9. Mantenibilidad. *Jezdimir Knezevic.*
10. Mantenimiento. *Jezdimir Knezevic.*

Isdefe
Ingeniería de Sistemas

c/ Edison, 4
28006 Madrid
Teléfono (34-1) 411 50 11
Fax (34-1) 411 47 03
E-mail: monografias@isdefe.es

P.V.P.: 1.000 Ptas.
(IVA incluido)

11

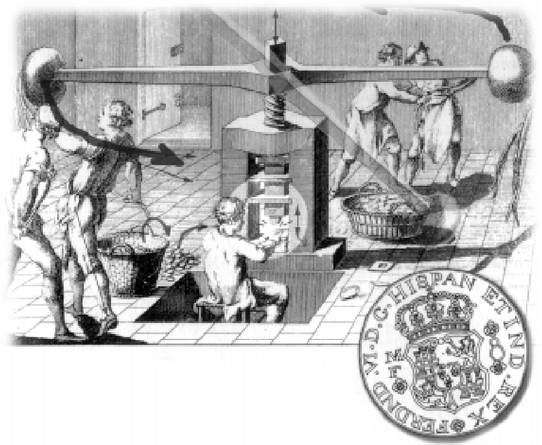
INGENIERÍA DE SISTEMAS DE SOFTWARE. *Gonzalo León Serrano*

Publicaciones de Ingeniería de Sistemas

INGENIERÍA DE SISTEMAS DE SOFTWARE

por

Gonzalo León Serrano



Isdefe

11

Gonzalo León Serrano

Dr. Ingeniero de Telecomunicación por la Universidad Politécnica de Madrid en 1982 es Catedrático de Ingeniería Telemática en la Escuela Técnica Superior de Ingenieros de Telecomunicación de Madrid. Actualmente, ocupa el cargo de Director del Departamento de Ingeniería de Sistemas Telemáticos y Presidente del Consejo Técnico del Centro de Investigación en Tecnologías y Aplicaciones Multimedia (CITAM).

Sus actividades de investigación se centran en el desarrollo de métodos y herramientas para el diseño de sistemas de comunicación y tiempo real.

Ha dirigido en este campo proyectos de investigación en los programas europeos ESPRIT, ACTS, EUREKA, COMETT y en el programa TIC (Tecnologías de la Información y de las Comunicaciones) dentro de los programas nacionales, así como con diversas empresas.

Forma parte del Software and Multimedia Advisory Committee del programa ESPRIT y del Technical Advisory Group del European Software Institute.

Es miembro del IFIP WG 2.4 (System Implementation Languages) y del IFIP WG 8.6 (Diffusion and transference of Information Technology) así como vocal del Consejo Asesor de Telecomunicaciones.

ILUSTRACIÓN DE PORTADA
Grabado de 1771 que representa una prensa de madera para acuñar monedas.

Primera Edición: Mayo - 1996
1.250 ejemplares

c/ Edison, 4
28006 Madrid.

Diseño y fotomecánica:
HB&h Dirección de Arte y Edición

Infografía de portada:
Salvador Vivas

Impresión:
Closas Orcoyen S.L.

ISBN: 84-89338-10-8
Depósito legal: M- -1996
Printed in Spain - Impreso en España.

PRÓLOGO

Esta monografía pretende recoger los aspectos más importantes del desarrollo de sistemas de software. Al formar parte de una serie bajo el epígrafe general de Ingeniería de Sistemas, hemos querido que el concepto de sistema quedase también reflejado en ésta.

No hay duda de que un sistema de software es un sistema, pero ¿tan distinto a otros que no se puedan emplear técnicas generales de ingeniería de sistemas? Si bien es cierto que, como tal sistema, un sistema de software hereda muchos de los aspectos generales de planificación del desarrollo que posee cualquier otro tipo de sistema complejo, las fuentes de su complejidad y las características especiales que su desarrollo conlleva, hacen de ellos unos sistemas bastante especiales.

Por indicar solamente algunas de sus características más sobresalientes en la problemática que nos interesa, los conceptos de fabricación, aprovisionamiento y distribución son claramente diferentes. La fabricación, porque es el único caso en el que el coste de replicación es prácticamente nulo; los de aprovisionamiento y distribución, porque los mecanismos de acceso y distribución electrónica de software a través de redes de datos implican problemas logísticos y soluciones muy diferentes a los clásicos en el desarrollo de un sistema.

Otro aspecto claramente diferenciador es el tipo de complejidad que estos sistemas poseen. No procede, en el caso de sistemas de software, de la multiplicidad de partes diferentes sino de la interrelación entre sus componentes que una persona aislada no puede percibir en su total complejidad. El manejo adecuado de niveles de abstracción y la capacidad de moverse de un nivel a otro dentro de una tecnología de software dada, es la base que posibilita el desarrollo de sistemas de software complejos.

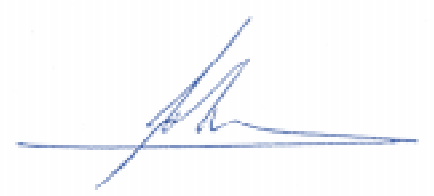
La otra perspectiva que quisiera destacar es que, muchos de los sistemas de software existentes son, a su vez, componentes de sistemas más complejos. Derivadas de un proceso de flexibilización y adecuación rápida y progresiva al entorno, muchas aplicaciones actuales han incorporado sistemas de software como forma de responder a necesidades cambiantes. Dicho de otro modo, los sistemas de software han penetrado y penetrarán aún más en muchos aspectos de nuestra vida; y no estarán aislados de otros componentes. Cada vez más, su desarrollo estará embebido en el de un sistema y su ingeniería será, ante todo, una ingeniería de sistemas.

A la hora de seleccionar los temas y el nivel de los mismos para la confección de esta monografía, hemos tenido presente que si se abstrae de la problemática concreta de que lo que se diseña es un sistema de software, las generalizaciones nos pueden hacer caer en ideas generales abordadas en otras monografías de esta serie. Si, por el contrario, nos centramos en las técnicas concretas que permiten abordar el desarrollo de un sistema de software, entramos en un cúmulo de detalles que hacen difícil captar los problemas generales del desarrollo de un sistema complejo.

Nuestro planteamiento ha sido pues el de mantener una visión global del desarrollo, combinando aspectos técnicos y de gestión, sin caer en detalles del uso de ninguna de las tecnologías existentes. Únicamente en el análisis de los sistemas de tiempo real se han empleado notaciones concretas para ayudar a la comprensión de su problemática.

Con todo lo anterior, a lo largo de la monografía tenemos que pensar en un sistema de software genérico y referirnos a la problemática concreta de su desarrollo a partir del marco genérico de la ingeniería de sistemas y particular de la ingeniería de sistemas de software. Esta es la línea directriz de nuestro trabajo.

Quisiera finalmente agradecer la colaboración recibida en la confección de esta monografía. Por un lado, a los miembros del Comité de Redacción de ISDEFE y, en especial a Alberto Sols, por los comentarios y sugerencias recibidas sobre el contenido de la monografía. Por otra, a Alejandro Alonso, Gregorio Fernández, Mercedes Garijo y Fernando Sáez Vacas, compañeros del Departamento de Ingeniería de Sistemas Telemáticos de la UPM quienes leyeron y comentaron el manuscrito inicial.

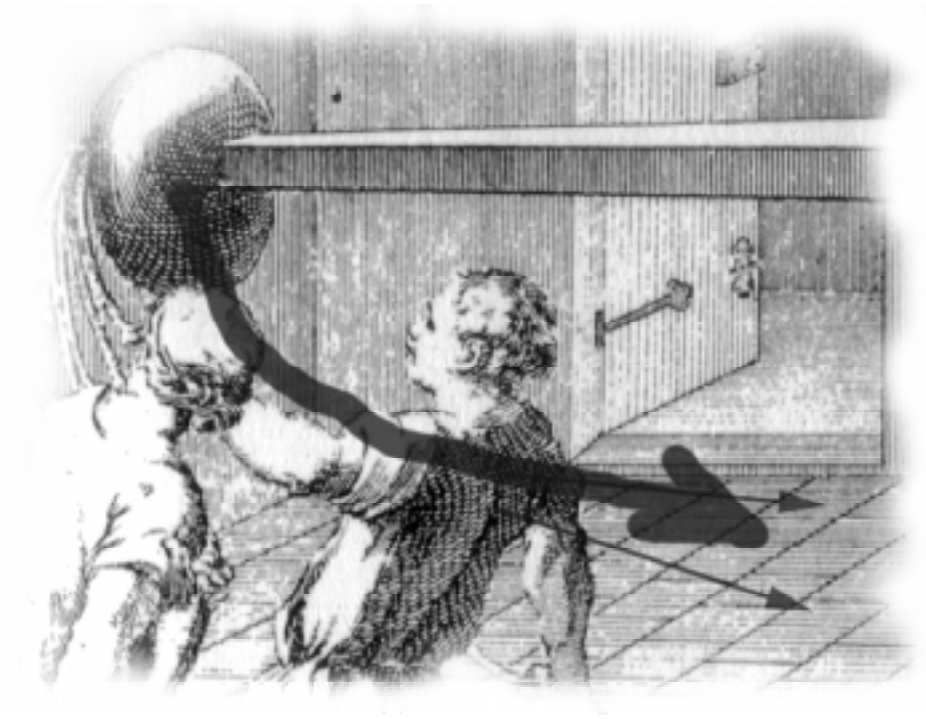


Gonzalo León Serrano

ÍNDICE GENERAL

1

La complejidad de los sistemas de software



1.1. Introducción

Que el «software» es un elemento básico en nuestra sociedad actual como generador de servicios parece un hecho evidente. Que sus costes de desarrollo (y, en muchos casos, de adquisición) sean cada vez más altos respecto al hardware sobre el que se ejecuta también es evidente aunque nos resistamos a aceptarlo en nuestra práctica cotidiana. Que un sistema de software envejece sin necesidad de estropearse (haciéndose inútil en un contexto dado) es un hecho al que nos hemos acostumbrado a pesar de ser una gran paradoja en productos sin partes mecánicas y con un coste de replicación prácticamente nulo.

Basta echar una ojeada a nuestro alrededor para ver cómo estos sistemas de software están teniendo una importancia creciente, responsabilizándose de los éxitos y fracasos de muchos sistemas basados en ellos y siendo también responsables de los éxitos y fracasos de las empresas que los construyen o utilizan.

Este Capítulo va a profundizar en el concepto de sistema de software con el fin de entender cómo esos hechos acabados de mencionar tienen su justificación. Deseamos que, al final del Capítulo, se disponga de una mejor comprensión del papel que estamos haciendo jugar a los sistemas de software y del que se deriva su complejidad actual.

Finalizaremos con una introducción a la idea de ingeniería de sistemas de software que da título a la monografía. Deseamos aquí

relacionarla con la ingeniería de sistemas y determinar su importancia; tiempo tendremos en Capítulos posteriores de analizar cómo la complejidad de los sistemas de software se puede controlar en el proceso de desarrollo a través de técnicas concretas.

1.2. El papel de los recursos software en sistemas complejos

Blanchard define un **sistema** en la primera monografía de esta serie como: una combinación de recursos (como seres humanos, materiales, equipos, software, instalaciones, datos, etc.) integrados de forma tal que cumplan una función específica en respuesta a una necesidad designada de un usuario [1]. No sólo incluye los recursos utilizados directamente en el cumplimiento de la misión (esto es, equipo principal, software operativo, personal usuario), sino también los diferentes elementos del apoyo (como por ejemplo: equipos de apoyo y prueba, repuestos y requisitos relacionados de inventario, personal de mantenimiento e instalaciones).

Esta es una definición genérica que incluye todo tipo de sistemas. Desde sistemas en los que no existen «recursos software» hasta aquellos otros en los que ésos son los elementos fundamentales para conseguir la funcionalidad pretendida. Llamamos **recurso software** a un programa o conjunto de programas ejecutables que proporcione algunas de las funciones requeridas por el sistema.

Desde esta perspectiva tan amplia, un sistema se considerará como **sistema de software** cuando sus recursos software constituyan su elemento básico y la fuente de su funcionalidad básica. Dicho de otro modo, cuando en el proceso de desarrollo sean los recursos software los que determinan el proceso general de desarrollo de todo el sistema y cuando su ejecución pueda realizarse sobre una plataforma hardware genérica.

Conviene distinguir entre un sistema de software y un programa ejecutable. Siguiendo una definición clásica de Wirth, un programa es

un algoritmo codificado junto con unas estructuras de datos. Algunas veces se emplea el término **paquete ejecutable** para referirse a un conjunto de programas que se necesitan mutuamente durante la ejecución del sistema y que deben distribuirse conjuntamente al usuario final.

Un sistema de software, por el contrario, es mucho más. Implica una interacción con el contexto al que sirve que constituye el referente básico de su utilidad. Un sistema de software posee programas ejecutables pero también otros tipos de recursos (ficheros de datos, de documentación, etc.).

Gran parte de los problemas que acechan a los diseñadores e implementadores actuales reside en que emplean durante el proceso de desarrollo una perspectiva limitada a los programas necesarios y no a una concepción sistémica del desarrollo del mismo ni del contexto social, humano y técnico que enmarca su ejecución. La ingeniería de sistemas de software es, ante todo, una ingeniería.

La **complejidad** de un sistema tal y como queda descrito a partir de la definición de Blanchard depende no sólo de las múltiples interacciones entre los recursos de que consta sino también de la forma en la que puede evolucionar en respuesta a las necesidades del entorno. Pues bien, el control de la complejidad de un sistema depende generalmente de las funciones dependientes de sus recursos software y de como éstas se adapten al mundo externo.

Queremos centrar nuestra atención en esta monografía sobre los sistemas de software **complejos** entendidos como aquellos cuyo desarrollo no es abordable por una única persona y en los que no existe seguridad absoluta de que se han implementado fielmente los requisitos exigidos por el usuario ni de que se ejecuta correctamente. La ingeniería de sistemas de software pretende, justamente, incrementar esta seguridad durante el proceso de desarrollo hasta alcanzar un nivel de confianza similar al existente en otras ingenierías.

Desde el punto de vista del diseñador y operador humano se conjugan, por tanto, dos propiedades que actúan como una espada de Damocles durante toda su vida útil: la **incertidumbre** de lo que realmente harán y la **ignorancia** en cómo lo consiguen. Reducirlas y dominarlas ha constituido la directriz básica en la evolución de la Ingeniería Software durante el último cuarto de siglo.

1.3. Una perspectiva histórica

El término **Ingeniería del Software** (ampliamente utilizado hoy día, aunque en esta monografía hemos preferido el de ingeniería de sistemas de software por recalcar el aspecto de sistema) fue acuñado en 1969 en el transcurso de un curso de verano de la OTAN en Garmisch.

Centrándonos en la ingeniería de sistemas de software, su consolidación ha sufrido una evolución en etapas en paralelo con la propia evolución de la programación. Destacamos cuatro etapas:

- **La programación como base del desarrollo** (1955-1965).
  Énfasis absoluto en la tarea de escribir el código en un lenguaje de programación. Alrededor de los nuevos lenguajes de alto nivel, los programadores se alejan de la estructura de los ordenadores y comienzan a acercarse a la complejidad de las aplicaciones de usuario.

- **La génesis** (1965-1975).
  Ligada a la crisis de la programación se plantea la necesidad de controlar el proceso de desarrollo. Se definen modelos de ciclo de vida como una referencia en la que enmarcar las actividades requeridas (se analizarán en el Capítulo 2). El concepto de ciclo de vida en cascada surge de la necesidad del Departamento de Defensa de EE.UU. de disponer de una documentación normalizada para todas las etapas del desarrollo y poder controlar en base a ella a los suministradores de productos software.

- **La consolidación** (1975-1985).
  El control de las actividades de desarrollo debería permitir gestionar el proceso. Durante esta etapa aparecen métricas para estimar a priori el coste o el tamaño del sistema; se difunde el uso de métodos de desarrollo. Con ello, el programador se convierte en analista, diseñador o gestor. Se vislumbra la idea de ingeniero (software).

- **Hacia una ingeniería** (1985-1995).
  Aceptando una consolidación de las tecnologías de software, la mejora viene de la mano de un mejor conocimiento de los procesos con el fin de incrementar la calidad de los productos. Aparece una gestión sofisticada del proceso de desarrollo ligada al control de riesgos y a la madurez de los procesos.

A lo largo de estas etapas, han existido avances puntuales significativos tanto en la tecnología empleada como en la propia percepción del proceso de desarrollo. En la Figura 1 hemos indicado los hitos más importantes de esta evolución. No pretenda el lector comprenderlos en este momento; serán descritos posteriormente. Sí queremos ofrecer una visión general y hacer ver con ella la relación existente entre ellos y su importancia relativa; el progreso hacia la ingeniería de sistemas de software ha sido acumulativo en los años cubiertos por las etapas mencionadas y no se puede entender ningún progreso sin la experiencia obtenida de éxitos y fracasos anteriores.

Analizados globalmente, estos hitos significativos en el desarrollo de la ingeniería de sistemas de software han ido intercalando el énfasis sobre las tecnologías de desarrollo con el énfasis en la gestión del proceso. En cada una de estas etapas se consiguió un incremento de la calidad del proceso de desarrollo.

Si inicialmente la esperanza de una mejora de calidad del producto se centraba en el empleo de nuevos lenguajes de programación,

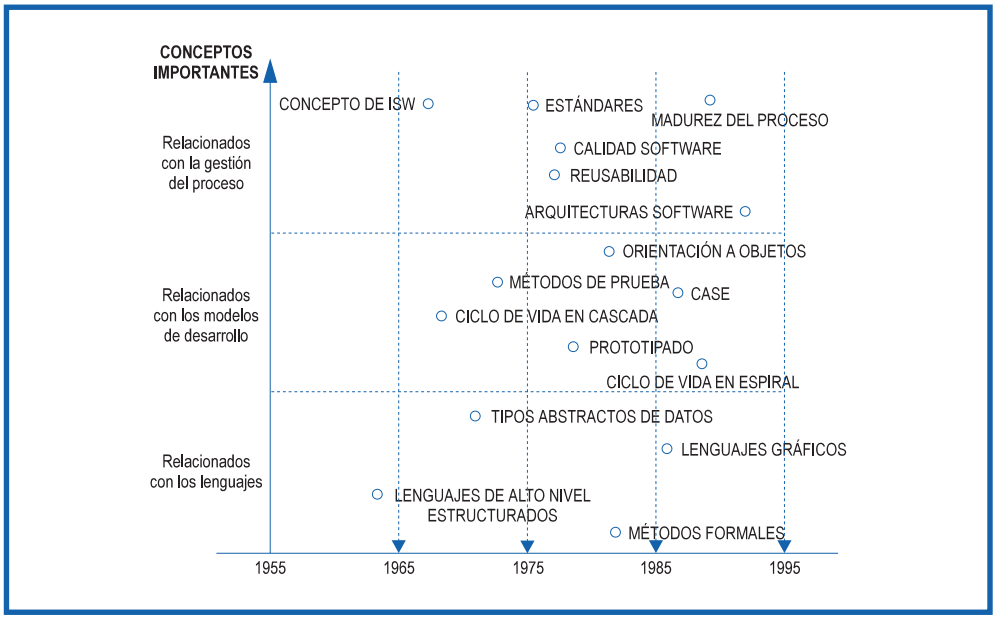


Figura 1 - EVOLUCIÓN HISTÓRICA DE LA INGENIERÍA DE SISTEMAS DE SOFTWARE -

después se hizo necesario una mejor comprensión del ciclo de vida. Posteriormente, fue necesario robustecer ese ciclo de vida en cascada con tecnologías de software que facilitasen las primeras fases del ciclo de vida. Pero el empleo de métodos y herramientas software de ayuda no hacía más predecible y eficiente el desarrollo de un gran sistema: el énfasis se situó de nuevo sobre los aspectos de gestión con la mejora del proceso. Esta es también la evolución que hemos seguido en la presente monografía.

1.4. Enfoques complementarios de los sistemas de software

Para entender un sistema de software necesitamos emplear diferentes perspectivas complementarias que, globalmente, nos permiten comprender su función dentro de un sistema socio-técnico y la tecnología necesaria para su desarrollo. Identificamos las siguientes:

A) La infraestructura de ejecución.

Todo sistema de software requiere un procesador que interprete o ejecute sus instrucciones y el apoyo de otros programas que le ofrezcan una serie de servicios útiles (como los ofrecidos por el sistema operativo).

Estos programas de apoyo o los intérpretes o ejecutores requeridos constituyen una **máquina virtual**. Se denomina virtual porque puede estar constituida por otros sistemas de software y no necesariamente por una máquina física que para muchos programas de aplicación está oculta. Denominaremos **infraestructura de ejecución** al conjunto de estas máquinas virtuales.

La mayor parte de los sistemas de software actuales requieren para su ejecución de infraestructuras de ejecución cada vez más complejas, incluyendo procesadores especializados, sistemas operativos con funcionalidades determinadas (como soporte a la distribución), muchas veces satisfaciendo normas internacionales, paquetes de aplicación necesarios para su ejecución (como entornos de ventanas, bases de datos, etc.).

Como ejemplo, un programa que se ejecute utilizando los servicios proporcionados por el sistema operativo (S.O.) para la gestión de las operaciones de entrada-salida no tiene por qué conocer la estructura del ordenador; el sistema operativo, por el contrario, sí lo requiere. En la Figura 2 podemos ver un caso muy común en el que un programa de aplicación (por ejemplo, uno de análisis de requisitos de usuario) se apoya en un paquete de uso horizontal muy extendido (como una hoja de cálculo) que, a su vez, requiere un entorno de ventanas (como el conocido «Windows») que se apoya en un S.O. (como «MS-DOS»). Es el S.O. quien interacciona con los recursos de entrada-salida (en una arquitectura típica de un ordenador personal).

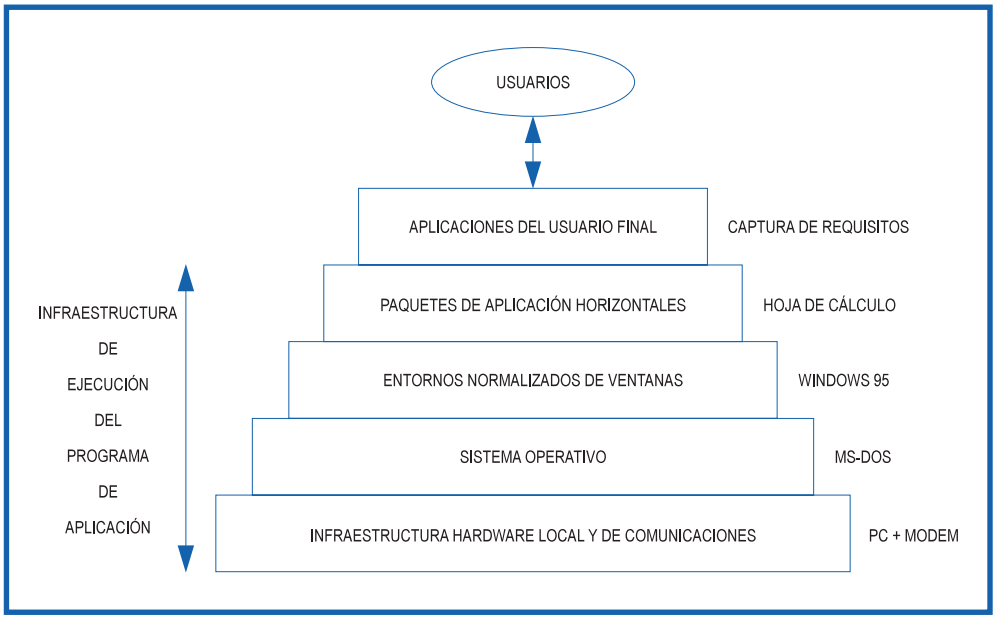


Figura 2 - EJEMPLO DE MÁQUINAS VIRTUALES -

B) El proceso de desarrollo.

No todos los sistemas de software se conciben y desarrollan de la misma manera. Su desarrollo, desde las primeras ideas sobre lo que deberían hacer y las restricciones de operación, hasta su entrega al usuario, pasan por una serie de fases. A estas fases, junto con las que siguen una vez entregado el producto al usuario hasta su retirada del mercado se les denomina globalmente **ciclo de vida**.

Existen muchas maneras diferentes de concebir las fases, las actividades que se realizan en ellas, las técnicas utilizadas para cubrir sus objetivos y los criterios por los que consideramos que esa fase ha terminado y debemos comenzar la siguiente.

Gran parte de la actividad en ingeniería de sistemas de software ha estado ligada a un mejor conocimiento del

proceso seguido (o propuesto) en el desarrollo de un sistema. Obsérvese que desde esta perspectiva no sólo estamos interesados en el sistema creado sino en el proceso de gestión de su desarrollo.

El Capítulo 2 describirá diferentes modelos de ciclo de vida así como sus ventajas e inconvenientes.

C) La evolución.

La atención sobre un sistema de software no termina cuando se entrega al usuario; de hecho, comienza un largo proceso en el que el sistema es mantenido con el fin de adaptarlo a las necesidades, reparar o completar la funcionalidad requerida y, sobre todo, asegurar que sigue siendo útil en el entorno de ejecución.

La infraestructura de ejecución tampoco es estable y su evolución obliga a que el sistema de software considerado también deba evolucionar. Todos hemos sufrido las continuas adaptaciones sobre nuestras aplicaciones derivadas de cambios de versión de los S.O. o de las rápidas y continuas mejoras en las arquitecturas de los ordenadores utilizados.

En muchos casos la funcionalidad del sistema de software consiste en dar un servicio a un usuario que interacciona con el sistema a través de una determinada interfaz hombre-máquina. El perfil del operador, sus conocimientos necesarios y la previsible evolución de su interés, condiciona también la evolución del sistema de software que le da servicio.

D) El dominio de la aplicación.

Muchos sistemas de software comparten una misma problemática con otros sistemas similares o pertenecientes al mismo tipo de aplicación. Por ejemplo, los sistemas de

tiempo real presentan una serie de características que obligan a emplear determinadas técnicas durante el proceso de construcción que no son necesarias para otros tipos de sistemas.

Igualmente, soluciones ya probadas en otros sistemas pueden también ser útiles aquí. Un aspecto tan importante como el de la reutilización está condicionada por un exhaustivo conocimiento del dominio de aplicación.

En el Capítulo 4 nos referiremos a un dominio de aplicación concreto: los sistemas de tiempo real. La comunidad de usuarios en ese dominio no sólo comparte un conjunto de técnicas específicas sino que también comparte un vocabulario común y unas prioridades en los problemas que les afectan.

La ingeniería de sistemas de software va a emplear en el desarrollo de un sistema concreto una tecnología constituida por lenguajes, herramientas, métodos, etc., que es la que permite obtener el sistema final. La tecnología de software no es, sin embargo, observable externamente una vez que el sistema se ha terminado aunque la calidad del sistema final depende de la tecnología empleada en el proceso de desarrollo.

La Figura 3 nos permite ver la relación entre las perspectivas mencionadas y los elementos básicos a tener en cuenta. Estas perspectivas no son independientes. Hemos identificado dos planos; uno de ellos, denominado de ejecución, afecta al sistema construido; el otro plano, de gestión, afecta al proceso de construcción del sistema. Ambos están imbricados.

La infraestructura de ejecución impone restricciones al proceso de desarrollo y al dominio de aplicación. Al proceso de desarrollo porque algunas cosas no serán posibles; al dominio de aplicación porque éste puede no ser abordado con la infraestructura disponible.

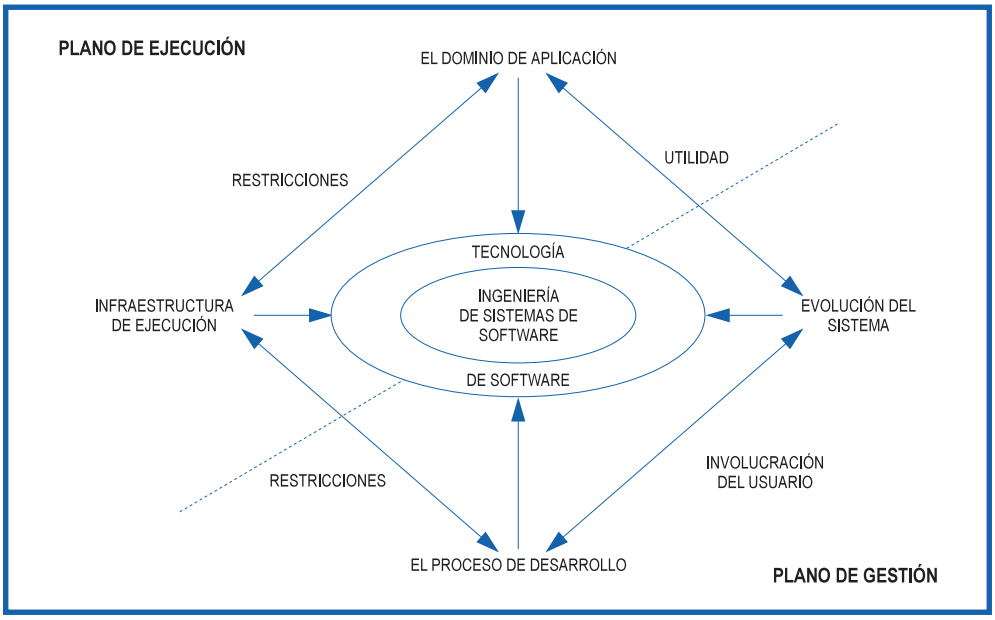


Figura 3 - PERSPECTIVAS DE UN SISTEMA DE SOFTWARE COMPLEJO -

La evolución del sistema está asimismo ligada al dominio de aplicación con el fin de asegurar la utilidad del sistema en ese dominio, y al proceso de desarrollo ya que el usuario debe involucrarse en éste para asegurar que la evolución del sistema sea aceptada a partir de las modificaciones requeridas por el usuario.

Estas perspectivas, que van a ser contempladas en futuros Capítulos de esta monografía, son capturadas por la tecnología de software tal y como se representa en la Figura 3; la tecnología empleada debe dar respuesta a las necesidades planteadas desde cada uno de los enfoques.

1.5. Caracterización de los sistemas de software

Antes de entrar en la descripción de cada uno de los enfoques mencionados, y con objeto de ofrecer una visión general más completa, vamos a caracterizar los diferentes tipos de software existentes.

No buscamos una clasificación exhaustiva sino criterios de clasificación que permitan encuadrar los sistemas de software existentes y obtener una rápida panorámica de su problemática.

1.5.1. Características relevantes de un sistema de software

Con el fin de clasificar a los sistemas de software hemos seleccionado un conjunto de características relevantes de los sistemas de software complejos. No queremos con ello decir que estas características sean fundamentales en todos los sistemas de software; probablemente, para algunos de ellos constituyan un elemento básico en su desarrollo y mantenimiento, y para otros no lo sean. Las características consideradas son las siguientes:

A) **Tamaño**. Que el tamaño condiciona el desarrollo de un sistema es algo que intuitivamente todos podemos suponer. Más dificil es acotar el concepto de tamaño y caracterizar en función de él los sistemas actuales.

La primera decisión reside en determinar a qué se aplica el concepto de tamaño. Durante mucho tiempo (y aún hoy) este concepto se aplica al código fuente (escrito en un lenguaje de programación utilizado por el implementador). Se han empleado diversas medidas basadas en contar las líneas de código fuente para estimar costes de desarrollo (esfuerzo requerido).

El tamaño final del sistema de software ha sido también empleado para conocer los requisitos sobre la infraestructura de ejecución y/o comunicación y, por comparación con otros sistemas, aspectos de prestaciones en tiempos de ejecución.

No obstante lo anterior, en el desarrollo de un sistema de software en el que se hayan empleado generadores de

código (programas especializados que, a partir de una especificación de muy alto nivel de lo que el usuario desea, son capaces de generar código fuente en un lenguaje de programación convencional) en el proceso de construcción, el tamaño no puede emplearse como una medida relativa del esfuerzo necesario. Si el código es parcialmente generado de forma automática, el esfuerzo del diseñador se desplaza hacia el diseño o la especificación de requisitos del sistema.

Los sistemas de software que nos interesan en esta monografía son grandes. En ellos, con independencia del objeto (especificación, código fuente) al que calificamos como grande (el tamaño de todos los posibles objetos están relacionados), hay una consecuencia general: su desarrollo no es abordable por una única persona. Son desarrollados por un grupo de trabajo.

B) **Vida útil**. Aún hoy día, muchos sistemas críticos en la Banca, Defensa, etc. fueron concebidos en una época en la que muchas de las técnicas actuales no existían o no eran ampliamente utilizadas.

Es común encontrarse con sistemas con más de 25 años de vida útil. Obviamente, en ese periodo han sufrido un sinfín de modificaciones, adiciones y sustituciones de funciones... pero ahí siguen. Que los sistemas tengan larga vida hace que un porcentaje apreciable del esfuerzo y costes se desplacen desde las fases de desarrollo a las de mantenimiento o evolución del mismo.

El concepto de evolución no debe entenderse como algo negativo. Por el contrario, es un elemento básico para asegurar su utilidad futura. Como consecuencia, la evolución debe considerarse desde el principio del desarrollo. Téngase

presente que no es sencillo controlar la evolución de un sistema de software que no ha sido diseñado para ello.

C) **Información manipulada**. Todo sistema de software necesita manipular información durante su ejecución; por información entendemos datos, texto, imágenes, etc. que deban ser procesadas o transmitidas. En alguno de ellos, no obstante, el volumen de información necesario obliga a que la atención en el desarrollo se centre en su captura, almacenamiento estructurado, procesamiento y actualización. Los sistemas complejos requieren que su arquitectura interna refleje la estructura de estos datos y los procedimientos de gestión de los mismos.

La gestión de la información no sólo implica que un módulo (parte del sistema) los mantenga sino que puede ser compartida entre varios, estar replicada y/o distribuida en varios emplazamientos dónde el sistema se ejecuta y casi siempre requiere de paquetes software especializados.

D) **Estructura interna**. Un sistema de software posee una estructura interna en el que las funciones a realizar se agrupan en módulos u objetos con cierta interacción entre ellos. Los sistemas de software complejos suelen estar compuestos por módulos que operan concurrentemente entre sí y, en muchos casos, sus módulos se ejecutan en lugares geográficamente distantes.

Gran parte de los lenguajes de programación clásicos (estructurados) poseen un conjunto de construcciones cuya finalidad es determinar el orden de ejecución de las instrucciones. Todos ellos comparten la idea de que en cada momento sólo se ejecuta una instrucción. Esta restricción se ha trasladado a las técnicas de diseño limitando el modelado.

En el caso de los sistemas de software complejos, la necesidad de disponer de módulos que actúen concurrentemente o de forma distribuida ya no es ocultable o prescindible. Los diseñadores deben disponer de técnicas que les permitan definir y controlar la concurrencia entre módulos del sistema o, incluso, entre subsistemas diferenciados asegurando que la interacción entre ellos se lleva a cabo coordinadamente en función del lugar en el que se ejecutarán.

E) **Prestaciones**. De un sistema de software se espera que haga lo que tenga que hacer dentro de límites de tiempo preestablecidos o que sea capaz de manipular un volumen de información dado dentro de restricciones de espacio conocidas. Estos valores determinan las **prestaciones** del sistema.

En resumen, un sistema de software complejo suele ser grande, tener un vida útil muy larga, procesar una información rica y compleja, organizarse en módulos u objetos que operan concurrente y distribuidamente y requerir la satisfacción de prestaciones críticas.

1.5.2. La utilidad de un sistema de software

Si en la sección anterior hemos revisado las características que puede exhibir un sistema de software atendiendo a parámetros técnicos que condicionan la tecnología empleada en su desarrollo, ahora queremos clasificar los sistemas de software por su **utilidad**.

La utilidad es visible al usuario que lo adquiere o encarga, los parámetros técnicos sólo lo son al diseñador aún en el caso de que hayan surgido como respuesta a necesidades expresadas por el usuario. En función de su utilidad, distinguimos tres tipos de sistemas de software:

1) **Software de base**. Es todo aquel sistema de software que proporciona una plataforma de ejecución (como un sistema operativo o una interfaz de usuario o un compilador) para que otros programas puedan desarrollarse o ejecutarse.

   Su utilidad no está ligada directamente a una aplicación definida por un usuario sino a otro programa. Suelen formar parte de la infraestructura de ejecución. Un ejemplo típico es el sistema operativo.

2) **Software de aplicación**. Responde a una necesidad de un usuario en un determinado contexto que se resuelve mediante un paquete de aplicación. El usuario interacciona directamente con el software de aplicación del que obtiene el servicio deseado.

   Se distingue entre software de aplicación **horizontal** cuando responde a necesidades genéricas manifestadas por multitud de usuarios (por ejemplo, un procesador de textos) y software de aplicación **vertical** cuando la necesidad responde a un tipo de usuario concreto o a un mercado sectorial restringido (por ejemplo, un sistema de gestión de hospitales).

3) **Software empotrado**. Hasta ahora hemos considerado el sistema de software ejecutándose potencialmente sobre una máquina genérica. En muchos casos no es posible aislar el sistema de la máquina sobre la que se ejecuta. De hecho, no hay interacción externa sino que se realiza sobre un hardware en el que está «empotrado» o «embebido». El sistema de software considerado forma parte de un sistema de software/hardware mayor sobre el que actúa. No hay interacción con el exterior. Un ejemplo de software empotrado lo tenemos en los sistemas de control automático de vuelo disponibles en muchas aeronaves.

Ya dijimos al comienzo del Capítulo que no era probable encontrar un sistema de software puro. Ante cualquier sistema de mediana complejidad, el «software» es un componente más y como tal debe entenderse.

1.5.3. El valor añadido del software

La última perspectiva de clasificación es la que ofrece el valor añadido que éstos tienen en un determinado contexto socio-económico. En función de ello, podemos determinar los siguientes tipos:

A) **Crítico**. La funcionalidad no puede ser ofrecida de otra manera. Disponer de un sistema de software es la única forma de conseguirlo.

B) **Degradable**. Si bien la funcionalidad ofrecida es importante, pueden existir otras formas de hacerlo (aunque pueden ser más costosas o inconvenientes).

C) **Prescindible**. Si bien ofrece una funcionalidad útil, podría ser sustituido por un sistema hardware o, simplemente, no ofrecerse porque su valor añadido es escaso. Muchas veces, la justificación de realizar un desarrollo software está ligado a condicionantes de tipo económico o de facilitar la evolución futura del sistema.

D) **Superfluo**. El sistema de software no ofrece ninguna funcionalidad necesaria para el usuario; por ejemplo, se ha introducido como forma de mejorar el proceso de comercialización de un producto.

La complejidad del sistema viene ligada a la forma en la que estas perspectivas se relacionan. La Figura 4 relaciona un parámetro técnico, el tamaño, con el valor añadido y el rol jugado por el sistema. Lo que la

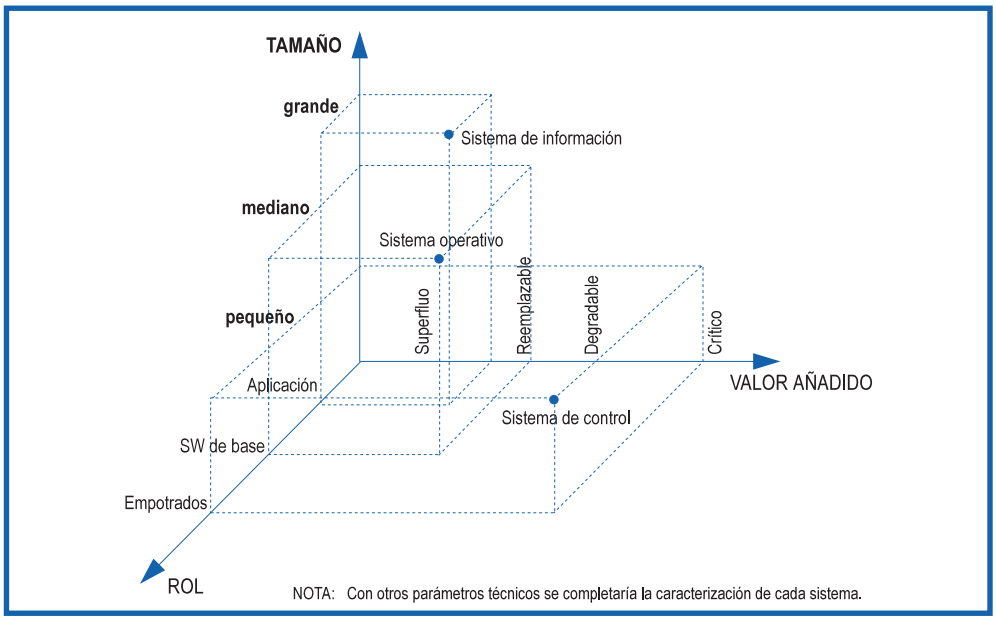


Figura 4 - CLASIFICACIÓN DE LOS SISTEMAS DE SOFTWARE -

figura (con tres ejemplos) sugiere es que cada sistema de software es distinto y ocupa un lugar diferente en el espacio de soluciones.

1.6. Ingeniería de sistemas de software

La ingeniería de sistemas se define en Blanchard como la aplicación efectiva de esfuerzos científicos y de ingeniería para transformar una necesidad operativa en una configuración definida de un sistema mediante el proceso iterativo de análisis de requisitos, la selección del concepto, y asignación, síntesis, soluciones de compromiso y optimización del diseño, prueba y evaluación [1]. Entre sus características se incluye su estructura de arriba-abajo que ve el sistema como un todo; una orientación del ciclo de vida que considera todas las fases desde el diseño conceptual hasta la retirada del sistema; un enfoque interdisciplinar «en equipo» que incluya todas las disciplinas adecuadas de diseño de forma oportuna y concurrente; y la necesaria integración

para asegurar que todos los objetivos de diseño se han cumplido de forma efectiva y eficaz. Está orientada al «proceso» e incluye las provisiones esenciales de realimentación y control.

Deseamos introducir aquí una definición clásica de ingeniería de sistemas de software como contrapunto a la anterior. Pressman la define enfatizando aspectos concretos de calidad: "establecimiento y uso de principios de ingeniería robustos, orientados a obtener software económico que sea fiable y funcione de manera eficiente sobre máquinas reales" [2].

Pressman presupone que el sistema de software a realizar cumpla con la misión encomendada, satisfaga las expectativas del usuario, etc. Si comparamos esta definición con la de ingeniería de sistemas anterior, vemos que la de Pressman está enfocada al proceso de desarrollo y no tanto a la tecnología con la que se desarrolla; para él es más una ingeniería de proceso y no tanto de producto.

De su análisis se desprende que la ingeniería de sistemas de software es una especialización de la ingeniería de sistemas cuya creciente importancia está ligada a la de los sistemas de software en nuestra sociedad.

1.7. Resumen

En este primer Capítulo hemos pretendido ofrecer una visión global de lo que es un sistema de software, su clasificación desde diversas perspectivas y la idea de sistema de software complejo que vamos a utilizar en el resto de la monografía.

La complejidad de los sistemas de software hace que, por un lado, no sea posible desarrollarlos individualmente y, por otra, que se requieran tecnologías de soporte desde su concepción hasta su retirada en el mercado.

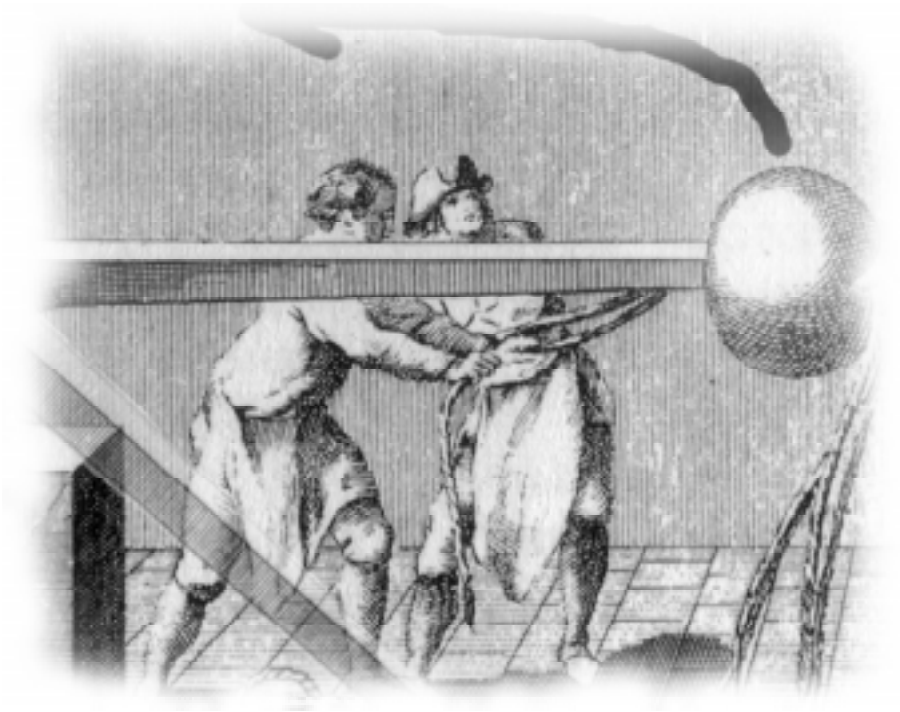
En este contexto, la ingeniería de sistemas de software, enmarcada en la ingeniería de sistemas, pretende ofrecer un marco para poder comprender el desarrollo y gestión de sistemas de software complejos.

Estamos asistiendo en estos días a una polémica sobre la verdadera naturaleza de la ingeniería de sistemas de software y su consideración como una disciplina real de ingeniería [3, 4]. En EE.UU. esta polvareda se ha levantado relacionada con el acuerdo de algún Estado en no aceptarla como profesión reconocida.

En todo caso, si de lo que se trata es de saber si existe un cuerpo de doctrina transferible a los ingenieros de software que permita desarrollar eficiente y controladamente un producto software, debemos decir que el paso de una construcción artesanal a una ingeniería predecible y fundamentada se está produciendo. La aparición de textos que combinan aspectos pragmáticos con fundamentos teóricos de muchas de las prácticas empleadas [5], es un índice de la evolución de esta disciplina. Esta monografía permitirá a los lectores hacerse su propia idea del estado de la misma.

2

Modelos de ciclo de vida



2.1. Perspectivas del proceso de desarrollo de software

El desarrollo de un producto software de cierta complejidad es un desafío intelectual tanto para la organización en la que se desarrolla como para cada una de las personas que intervienen. Estos dos factores, humano y organizativo, se imbrican durante el proceso de gestación del producto.

Producto y proceso concentran por tanto la atención en ingeniería de sistemas de software. Sobre el **producto** porque en él deben incorporarse los requisitos que el usuario desea y es el resultado final del desarrollo; sobre el **proceso de desarrollo** porque de él depende el que esos requisitos sean realmente satisfechos en el producto final dentro de las restricciones de tiempo y coste establecidas.

En el presente Capítulo nos vamos a preocupar del **proceso de desarrollo de un sistema de software**. Dejaremos para el siguiente Capítulo una revisión de las tecnologías software que posibilitan el desarrollo de un producto concreto.

2.1.1. El factor humano

El desarrollo de software sigue siendo una actividad intensiva en **capital humano**. Más que otras técnicas de desarrollo de sistemas, el desarrollo de software se basa en los equipos humanos de trabajo y, en menor medida, en inversiones materiales.

Si adoptamos una perspectiva individual, cada componente del equipo de trabajo ve el desarrollo de un producto desde una perspectiva limitada a las actividades y parte del sistema en las que desarrolla su trabajo.

En sistemas muy pequeños es posible que el trabajo pueda completarse individualmente pero ésto no es factible cuando la complejidad del sistema supera un mínimo (y la capacidad de un individuo pone el límite en valores muy por debajo de lo que es necesario en la mayoría de los sistemas).

La ingeniería de sistemas de software se preocupa principalmente del proceso de desarrollo que implica a un equipo numeroso de personas en el desarrollo de sistemas de software complejos. En estos casos, cada ingeniero de software forma parte de un equipo de trabajo y desarrolla su actividad en relación con los componentes del mismo.

En sistemas de software complejos, el conjunto de actividades ligado a un componente del equipo de trabajo puede ser muy diferente de otro. En función de las actividades y de los conocimientos necesarios para realizarlas, cada componente del equipo de trabajo posee un **perfil técnico** especializado. Los perfiles necesarios, aunque no totalmente disjuntos, permiten establecer una primera división del trabajo durante el proceso de desarrollo.

Cada perfil técnico implica unos conocimientos asociados a las actividades relacionadas con ese perfil en el proceso de desarrollo. Simultáneamente, el perfil también implica la existencia de unas capacidades de comunicación con otras personas (del equipo de desarrollo o externas a él) de acuerdo a intercambios de información y protocolos de cooperación entre ellas que deberán estar bien definidos.

El concepto de perfil técnico y los perfiles concretos han ido evolucionando con el tiempo. Tradicionalmente, se empleaban los perfiles de analista, diseñador, programador, jefe de proyecto, etc.; todos

ellos fuertemente ligados a las etapas en las que se dividía el desarrollo de un determinado sistema y que veremos posteriormente. Bajo esta idea, cuando la actividad ligada a un perfil culminaba su trabajo entraba en juego el siguiente (un modelo derivado de la cadena de montaje en la que se inspiraba el desarrollo software). Era un modelo basado en la división vertical del trabajo.

Los problemas derivados del modelo de desarrollo empleado clásicamente así como la disponibilidad de nuevas tecnologías de desarrollo, están haciendo obsoletos algunos de los perfiles convencionales. Esta evolución tiende a que cada componente del equipo de trabajo posea una visión más amplia del proceso de desarrollo aunque limitada a una perspectiva concreta sobre el sistema en construcción. Sólo con esa visión global es posible actuar sobre aspectos globales del sistema.

Como ejemplo, un arquitecto software (encargado de determinar la estructura global del sistema, el uso de componentes reutilizables, la existencia de interfaces definidas, etc.) debe intervenir desde las primeras etapas del desarrollo hasta el comienzo de la implementación del sistema. En otro ejemplo, una persona encargada de controlar las diferentes versiones y bibliotecas de módulos empleados extiende su actividad sobre toda la duración del desarrollo con el fin de asegurar la consistencia entre las decisiones adoptadas.

Esta situación implica la aparición de un modelo horizontal de división del trabajo que se superpone parcialmente con el vertical mencionado anteriormente. A cada componente del equipo de trabajo se le va a exigir un conocimiento mayor sobre la totalidad del desarrollo aunque su actividad enfatice aspectos concretos.

Las consecuencias que ello tiene sobre la estructura del equipo de trabajo son más amplias de lo que parece a simple vista. Un equipo humano formado por especialistas con formación y actividades más horizontales, requiere de técnicas de gestión y de interacción distintas... y una formación diferente.

El objetivo ideal estriba en conseguir que el equipo funcione globalmente como lo haría un experto que poseyera todos los conocimientos requeridos. Más adelante volveremos sobre los modelos del proceso de desarrollo y el uso de estos perfiles.

2.1.2. La organización

Que los individuos que forman parte del desarrollo de un sistema conozcan sus obligaciones a nivel individual, posean los conocimientos requeridos para ello e incluso que realicen adecuadamente las actividades encomendadas, no implica que el desarrollo del producto se lleve a cabo con éxito ni que la institución haya aprendido de ello para futuros desarrollos.

Para ello, es necesario establecer una clara dependencia y control de las actividades de desarrollo, conseguir una eficaz gestión de los recursos implicados y, finalmente, asegurar un nivel de calidad adecuado a lo largo de todo el proceso.

Estas actividades, no estrictamente ligadas al desarrollo técnico del producto sino a la forma en la que éste se obtiene, recaen en la estructura de la organización. Un conjunto de actividades, orientadas a un fin concreto, empleando y generando determinada información relativa al sistema en desarrollo se denomina **proceso**. Por extensión, también se denomina **proceso de desarrollo** al conjunto de todos ellos.

Como ejemplo, un proceso sería la obtención de una nueva versión de un sistema; otro, la prueba de módulos ya implementados; otro más, la animación de un modelo del sistema de software durante las primeras fases; en otro caso, la subcontratación externa de una actividad. En la literatura sobre modelado de procesos de desarrollo de software se ha estimado que existen unos 200 procesos específicos necesarios para desarrollar un producto software por un equipo de

trabajo; todos ellos deben ser definidos y controlados a lo largo del desarrollo.

Para cada uno de los **procesos** se requiere fijar unas actividades, responsables, entradas y salidas, planificación temporal y de recursos necesarios, y mecanismos para asegurar que se realiza correctamente. Y no todas las organizaciones poseen la madurez requerida para ello.

Existen, evidentemente, muchas maneras de organizar un desarrollo, debiendo adaptar los procesos necesarios al tipo de software que se desea desarrollar, a la duración del proyecto (parcialmente ligado a su tamaño), o al uso futuro del modelo de desarrollo (reutilización de los procesos). Todas las organizaciones aprenden de los desarrollos actuales y van madurando. En el Capítulo 6 volveremos sobre los aspectos de madurez; bástenos decir, por ahora, que la mayor parte de las empresas dedicadas al desarrollo de sistemas de software no tienen completamente definidos ni controlados todos los procesos implicados.

2.2. Modelos de ciclo de vida: análisis comparativo

A partir de los años setenta, se había acumulado suficiente experiencia sobre el desarrollo de productos software para poder identificar un conjunto de actividades comunes a todos los proyectos. Asimismo, la estructura organizativa requerida para controlar su ejecución y la forma en la que el control de calidad aseguraba la validez de un determinado producto eran suficientemente conocidas como para definir marcos de referencia utilizables en nuevos proyectos de desarrollo. Esta idea desembocó en el concepto de **modelo de ciclo de vida** de un producto software.

Por ciclo de vida de un producto software se entiende el secuenciamiento de **fases**, **actividades** en cada una de ellas, **controles** para pasar de una fase a otra y **resultados** generados en cada una de ellas

que permiten el desarrollo de un producto desde su concepción, entrega al usuario, y evolución posterior, hasta su retirada del mercado.

Actualmente, a las fases de desarrollo del modelo de ciclo de vida se las conoce como **proceso de desarrollo software** tal y como le hemos empleado en la Sección previa.

No existe un modelo de ciclo de vida único. Tanto el tipo, orden y actividades en cada fase pueden cambiar adaptándose a las necesidades del producto a realizar y a la propia estructura de la organización que lo desarrolla a partir de las posibilidades que ofrece la tecnología de software empleada.

Aunque en la bibliografía es común encontrarse con clasificaciones de modelos de ciclo de vida muy detalladas pueden considerarse como variaciones de modelos básicos [6]. Nos reduciremos a ellos. Al menos, debemos considerar los siguientes:

- Modelo de ciclo de vida en cascada o convencional
- Modelo incremental.
- Modelo de síntesis automática.

A ellos podemos añadir el denominado **modelo en espiral** y que, realmente es un meta-modelo (modelo de modelos) en el que tienen cabida cualquiera de los anteriores desde una perspectiva de control de los riesgos del desarrollo.

Dedicaremos especial atención en esta monografía al modelo en cascada dado que sigue siendo, con diferencia, el más utilizado.

2.3. Modelo en cascada

El **modelo en cascada**, también denominado modelo convencional, responde a la secuencia de pasos de desarrollo de un producto

empleada desde el comienzo del desarrollo de software para la mayor parte de los sistemas de software.

Este modelo se caracteriza por la existencia de un conjunto de fases secuenciadas en el tiempo. A la finalización de una fase se comienza la siguiente tomando como datos de entrada los resultados de la anterior. En cada una de las fases se introduce más detalle hasta obtener un código ejecutable y eficiente que incorpore los requisitos necesarios. Tomaremos como referencia en la siguiente descripción el modelo de ciclo de vida propuesto por la Agencia Espacial Europea (ESA) [7] y que se corresponde con el modelo en cascada.

Las fases principales consideradas son las siguientes:

- Definición de requisitos.
- Diseño.
- Implementación.
- Transferencia del producto.
- Evolución.

Aún hoy se considera que todos los demás modelos no son más que modificaciones de este modelo básico. Aunque es cierto que todos lo demás modelos han derivado de él para evitar algunos de sus problemas, preferimos tratarlos de forma diferenciada.

2.3.1. Definición de requisitos

El objetivo de la fase de definición de requisitos (también se le suele denominar de «especificación» de requisitos) es obtener una clara comprensión del problema a resolver, extraer las necesidades del usuario y derivar de ellas las funciones que debe realizar el sistema.

Posiblemente con anterioridad a esta fase ha existido algún estudio de factibilidad que permite asegurar que el software a realizar es realizable, responde a un mercado potencial o real, etc.

La fase de definición de requisitos suele subdividirse en dos subfases: análisis de requisitos de usuario y análisis de requisitos de sistema.

La subfase de **análisis de requisitos de usuario** tiene como objetivo conocer las necesidades de los usuarios y cuáles deben ser los servicios que un sistema de software debe ofrecerles para satisfacerlas. La fase implica la creación del **Documento de Requisitos de Usuario (DRU)** que constituye el documento base para que, al final del desarrollo, el sistema sea aceptado por el usuario.

Los requisitos de los usuarios pueden ser de muchos tipos diferentes. Por un lado, el usuario tiene requisitos que corresponden al servicio que el sistema de software que se pretende construir le debe proporcionar. En otros casos, se trata de limitaciones existentes en la operación del sistema.

Como ejemplo, un usuario puede requerir un sistema de control de un ascensor que, entre otros muchos requisitos, desde que el usuario pulsa un botón en la puerta hasta que ésta se cierre no deba transcurrir más de 3 segundos. Este requisito corresponde a un servicio que debe proporcionar al usuario. Otro ejemplo, podría ser que el sistema de software tenga que utilizar un determinado sistema operativo (lo cual limita el tipo de soluciones posibles).

Esta subfase se aprovecha también para generar el plan de desarrollo con una estimación de costes y recursos (en el Capítulo 5 entraremos en el detalle de cómo se realizan estas estimaciones).

La subfase de **análisis de requisitos del sistema** consiste en la construcción de un **modelo lógico** del sistema de software

describiendo las funciones que sean necesarias (sin tomar ninguna decisión sobre cómo implementarlas) y las relaciones entre ellas suponiendo que no existen limitaciones de recursos.

Por modelo lógico se entiende la identificación de las funciones de software requeridas para satisfacer los requisitos del usuario. Esta identificación se suele realizar en varios niveles de detalle hasta llegar a uno en el que las funciones identificadas estén suficientemente claras de tal forma que no exija un refinamiento posterior.

El producto generado en esta fase es el **Documento de Requisitos de Software (DRS)**. También se genera en esta sub-fase el plan de gestión del desarrollo con estimaciones de costes y recursos más ajustados que en la sub-fase anterior.

Es importante distinguir en esta subfase entre **requisitos funcionales** (aquellos ligados a la relación entre datos de entrada y resultados (datos de salida) que debe presentar el sistema, incluidos los derivados de restricciones temporales cuando éstas están cuantificadas) y **requisitos no funcionales** (que incluyen todos los aspectos de calidad del sistema: por ejemplo, mantenibilidad (facilidad para que el sistema evolucione y se modifique una vez entregado al usuario), escalabilidad (posibilidad de incrementar sustancialmente el número de usuarios u otros parámetros), facilidad de uso, etc., que no pueden ligarse a funciones concretas dentro del sistema.

Los mecanismos de control (en estas etapas son las revisiones de los documentos generados) deben asegurar que los requisitos software satisfacen completamente los requisitos de usuario. Esta actividad será parte de la validación del sistema y que veremos en el Capítulo 6.

Siguiendo con el ejemplo anterior, el requisito de usuario puede implicar un conjunto de funciones de software cuya relación debe establecerse en el modelo lógico. Concretamente, implicará una función de «cierre de puerta» cuyo tiempo de ejecución deberá ser inferior a 3

segundos (¡aunque en esta fase no podamos asegurarlo!; será posteriormente cuando podamos estimar y luego comprobar que, efectivamente, ello es posible).

En el caso de la limitación del usuario respecto del sistema operativo, implicará el uso de determinados servicios del mismo (llamadas al sistema).

2.3.2. Diseño

La fase de diseño tiene como objetivo determinar una solución a los requisitos del sistema definidos en la fase anterior. Obviamente, existen muchas maneras de satisfacer los requisitos y, por tanto, multitud de diseños posibles.

Es conveniente distinguir entre diseño de alto nivel o **arquitectónico** y diseño detallado. El primero de ellos tiene como objetivo definir la estructura de la solución (una vez que la fase de análisis ha descrito el problema) identificando grandes módulos (conjuntos de funciones que van a estar asociadas) y sus relaciones. Con ello se define la arquitectura de la solución elegida. El segundo define los algoritmos empleados y la organización del código para comenzar la implementación.

En la subfase de diseño arquitectónico se parte del modelo lógico generado en la fase de definición de requisitos software y se transforma en una arquitectura agrupando las funciones identificadas en componentes software. Asimismo, se define la activación y desactivación de cada uno de los componentes y el intercambio de información entre ellos (ahora con limitaciones de espacio).

Como ejemplo, si en el modelo lógico se ha establecido que una función entrega una información a otra, ahora es necesario determinar si ese intercambio de información se realiza mediante memoria

compartida o mediante mensajes. En ambos casos, hay que determinar el tamaño.

El resultado de esta fase es el **Documento de Diseño Arquitectónico (DDA)**.

Definir una buena arquitectura del sistema constituye un elemento básico para asegurar que el sistema sea luego mantenible e integrable con otros. En los últimos años se ha dedicado atención a la última subfase del diseño se conoce como **diseño detallado**. El diseño detallado en el modelo de la ESA engloba la codificación y prueba; nosotros, sin embargo, lo trataremos como parte de la fase de implementación. Al final de la fase, se genera el **Documento de Diseño Detallado (DDD)**.

El proceso de diseño detallado suele requerir diversos pasos de refinamiento en los que múltiples decisiones de diseño permiten incorporar restricciones de implementación derivadas del uso de recursos limitados: la ejecución de las funciones identificadas requiere tiempo y espacio de memoria o búfferes así como recursos de CPU que no son ilimitados.

El refinamiento culmina cuando la descomposición modular ha alcanzado el nivel suficiente como para poder comenzar la implementación del sistema en un lenguaje ejecutable con las prestaciones adecuadas en módulos de complejidad abordable por una persona (o un pequeño grupo).

Si recordamos el requisito de usuario anterior y la función software identificada, es ahora en la fase de diseño en la que podemos conocer los recursos software requeridos para ello y hacer una estimación (conocida la tecnología de software a utilizar) de que efectivamente es posible satisfacer ese requisito.

En el modelo propuesto por la Agencia Espacial Europea, el diseño detallado culmina con la realización del código; preferimos hablar de una fase de implementación independiente al ser una terminología más comúnmente aceptada.

2.3.3. Implementación

Su objetivo es producir una solución eficiente en un lenguaje ejecutable que implemente las decisiones adoptadas en la fase de diseño. Suele incluir la codificación y la prueba del sistema hasta obtener un paquete ejecutable sobre la plataforma (hardware y S.O.) requerida por el usuario.

Es interesante mencionar que todas las fases anteriores son conceptualmente independientes del lenguaje de programación seleccionado. Es ahora en la fase de implementación cuándo se selecciona y utiliza un lenguaje de programación determinado; lo que sí es evidente es que el conocimiento del lenguaje de implementación puede orientar la fase de diseño (como ocurre en el caso de los lenguajes de programación orientados a objetos) relacionando de forma más directa los objetos o módulos identificados con las construcciones del lenguaje.

Como en el proceso de refinamiento se ha dividido el trabajo entre diversos componentes del equipo de trabajo, éstos han trabajado concurrentemente en el diseño detallado y en la subsiguiente implementación de diversos módulos.

El problema es que ahora es necesario integrar los diversos módulos y construir el sistemas de software completo.

Se denomina **integración** al proceso de construir un sistema de software combinando componentes individuales en una unidad ejecutable. Este proceso de integración debe hacerse de forma ordenada para que se integren los módulos en función del uso que unos hacen de otros. La gestión del proyecto deberá asegurar que la integración se realiza adecuadamente.

Una vez obtenida la implementación del sistema es necesario probar que satisface los requisitos definidos inicialmente. Posiblemente, cada uno de los diseñadores que ha estado construyendo cada uno de

los módulos ha probado que su implementación está de acuerdo con las decisiones tomadas en el diseño pero no puede asegurar que al integrarlo con otros no existan problemas de incompatibilidades o aspectos no considerados individualmente en cada módulo. Es necesario, por tanto, realizar pruebas a diferentes niveles hasta que el sistema en su conjunto sea aceptado por el usuario (en el Capítulo 6 volveremos a la forma en la que se gestiona su desarrollo).

Al final de la fase, se genera el **Manual de Usuario** junto con el **código fuente** del sistema y las **pruebas** asociadas.

Aunque con la fase de implementación se dispone del «producto», no acaba con ella la actividad del equipo de desarrollo ni el ciclo de vida del sistema de software construido. A estas fases se les suelen añadir otras dos que son cada vez más importantes: la fase de transferencia y la de evolución.

2.3.4. Transferencia del producto

La fase de **transferencia** del producto tiene como objetivo instalar el sistema de software desarrollado en el entorno del cliente y realizar las pruebas de aceptación necesarias. En muchas ocasiones el proceso de transferencia implica un período largo en el que se incluye la formación del usuario en el producto y la realización de las pruebas de aceptación junto con el usuario.

Debemos tener presente que el usuario deberá aceptar el sistema que se le entrega en función de los requisitos de usuario que dieron origen a todo el proceso. Por ello, es importante que durante el desarrollo sea posible conocer las decisiones asociadas con los requisitos de usuario (trazabilidad de requisitos).

Si bien este esquema es válido para productos que se realizan bajo encargo para un cliente determinado (por tanto, satisfacen requi-

sitos concretos de este cliente), existen otros muchos productos desarrollados para un mercado abierto en los que los clientes no existen de forma individualizada sino que son clientes anónimos tipificados a partir de técnicas de mercadotecnia.

Para muchos productos de consumo general, la fase de transferencia continúa las actividades de prueba iniciadas durante la implementación con la colaboración del cliente. Es típico considerar en esta fase un número reducido y controlado de clientes que, a cambio de obtener un producto no totalmente probado (conocido como «beta» o «alfa test»), pueden disponer de él mucho antes de que se comercialice de forma general.

La entrega de productos «alfa» o «beta» es un reconocimiento implícito de que pueden existir problemas tanto de errores ocultos como de adecuación del producto al usuario que saldrán a la luz mediante la interacción con usuarios reales. No olvidemos que la prueba de un sistema de software puede demostrar la presencia de errores pero nunca su ausencia.

Se suele generar también en esta fase el documento de **Historia del Proyecto** que resume las lecciones aprendidas y de cuyo análisis se pueden extraer conclusiones para la mejora de los procesos de desarrollo en futuros proyectos.

2.3.5. Evolución

Una vez que el producto de software ha entrado en operación regular por el usuario no es de ningún modo un sistema inmutable. Todo producto software complejo debe adaptarse a un entorno que va cambiando (nuevas necesidades del cliente, evolución de la plataforma de ejecución hardware o software, etc). Un producto software que no evoluciona va haciéndose cada vez menos útil en ese entorno.

La evolución del sistema de software suele incluirse dentro de una fase denominada de **mantenimiento** aunque su implicación es mucho más amplia de lo que el término significa en otras ingenierías. A nadie se le ocurre llevar su coche a un taller para que le incorporen un nuevo cilindro; sin embargo, parece que modificar líneas de código se puede hacer sin alterar la sustancia del producto, lo cual no es cierto.

Se suele hablar de tres tipos diferentes de mantenimiento:

1) **Mantenimiento correctivo**. Pretende eliminar problemas surgidos durante la fase de operación del sistema y que no han sido detectados anteriormente.

2) **Mantenimiento perfectivo**. Pretende mejorar la funcionalidad del sistema ya sea en relación con la eficiencia en ejecución del mismo (menor tiempo de respuesta, optimización del uso de la memoria, etc.), facilitar su uso, etc.

3) **Mantenimiento evolutivo**. Pretende modificar (ampliar, eliminar o sustituir) la funcionalidad del sistema para adaptarla a las nuevas necesidades del usuario o con el objetivo de adaptarlo a nuevas interfaces hardware o software.

En el Capítulo 5 volveremos sobre los aspectos de mantenimiento y su relación con las técnicas de gestión.

2.3.6. Análisis global del modelo en cascada

La Figura 5 representa esquemáticamente las fases seguidas en el desarrollo que hemos expuesto anteriormente. No se ha representado la fase de mantenimiento cuyo análisis se realizará en un Capítulo posterior.

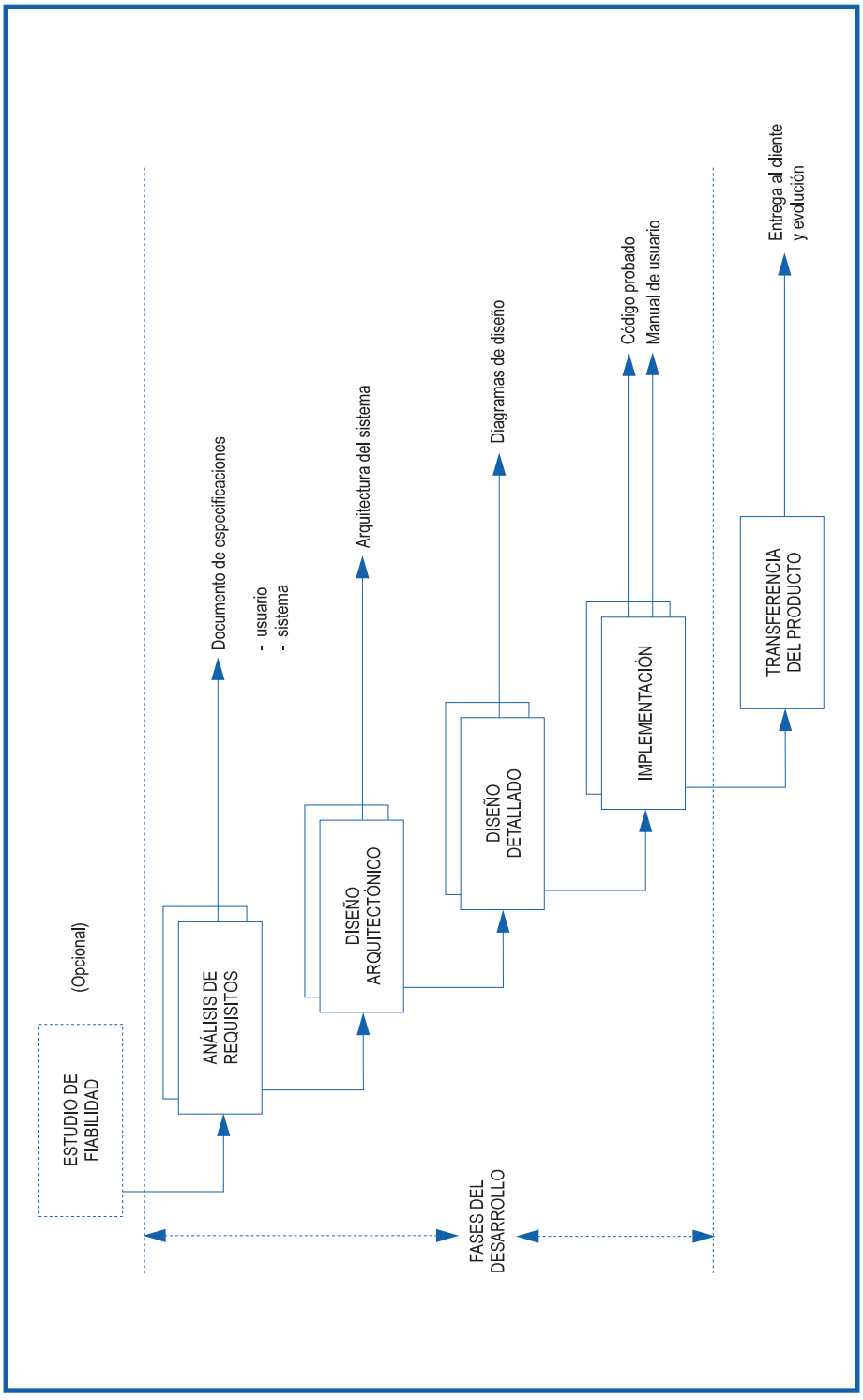


Figura 5 - MODELO DE CICLO DE VIDA CONVENCIONAL -

Las ventajas e inconvenientes del modelo de ciclo de vida convencional son bien conocidas desde hace tiempo y podemos decir que el 90 % de los desarrollos actuales se realizan de acuerdo con él. Como ventajas citamos:

a) Fases conocidas por todos los desarrolladores y ligadas a los perfiles técnicos clásicamente establecidos. Existe gran experiencia documentada sobre el uso del modelo que coincide con la formación típica del ingeniero de software.

b) Es el más eficiente cuando el sistema es conocido y los requisitos estables ya que se puede avanzar rápidamente hacia la fase de diseño arquitectónico sin que exista el peligro de una continua interacción entre las primeras fases.

c) Permite una gestión del proceso de desarrollo basada en revisiones de los documentos generados en cada fase facilitando la ejecución de los procedimientos de gestión.

El modelo en cascada ha sido normalizado por diversas entidades como base para el proceso de contratación y para la aceptación del software entregado. Una de estas entidades ha sido la Agencia Espacial Europea (ESA) quien, en su conjunto de normas de ingeniería software, ha definido el modelo de ciclo de vida a utilizar [7].

Un modelo normalizado como el de la Agencia Espacial Europea no solo presta atención a las actividades técnicas a desarrollar sino también a cómo éstas son gestionadas a lo largo del desarrollo. La imbricación de unas y otras define los procesos definidos por el modelo. Tenemos que destacar, no obstante, que el detalle de los procedimientos a usar a partir de un modelo no está contenido en el mismo sino que debe ser generado por sus usuarios; así, grandes corporaciones suelen definir complejas guías de aplicación en las

que se detallan los procesos a usar en cada una de las etapas adaptados a sus características organizativas o de los productos software que deben desarrollar.

La Figura 6 describe esquemáticamente la idea de este modelo normalizado. La figura está abstraída de las normas de la ESA que describen las actividades técnicas y de gestión [7]. En ella se han descrito las fases y asociadas a ellas las actividades fundamentales, los documentos generados y el proceso de revisión y validación de cada una. Obsérvese que este proceso de validación se realiza a partir de los documentos generados; sólo en el caso del código es posible automatizar el proceso de revisión (por ejemplo, existen programas que pueden leer de forma inteligente un código fuente con el fin de extraer ciertas estadísticas que permiten a los implementadores conocer la calidad del mismo).

Son múltiples las desventajas asociadas al modelo convencional; entre ellas citamos:

a) La **visibilidad** del proceso es muy limitada siendo el código generado el único producto con el que el usuario puede validar sus requisitos. Las entradas y salidas intermedias son documentos internos al equipo de desarrollo no pensadas para su validación por los usuarios. El único objeto formalizado es el código y aparece al final cuando las modificaciones (caso de afectar a las fases anteriores) son muy costosas.

b) No permite manejar fácilmente **cambios de requisitos** una vez iniciado el desarrollo.

   Es típico cuando el usuario no conoce exactamente lo que desea que cambie de opinión sobre sus necesidades o, simplemente, que no indique todas sus necesidades. A este problema se le conoce como de **inestabilidad** de los

FASES / ASPECTOS	DEFINICIÓN DE REQUISITOS DE USUARIO	DEFINICIÓN DE REQUISITOS SOFTWARE	DISEÑO ARQUITECTÓNICO	DISEÑO DETALLADO Y PRODUCCIÓN	TRANSFERENCIA AL CLIENTE
PRINCIPALES ACTIVIDADES	Determinar el entorno operativo Identificación de requisitos de usuario	Construcción del modelo lógico Identificación de requisitos del sistema	Construcción del modelo físico Definición de los principales componentes	Diseño de módulos Codificación Pruebas unitarias Pruebas de integración Pruebas de sistema	Instalación Pruebas de aceptación
ENTREGABLES	Documento de requisitos de usuario (DRU)	Documento de requisitos software (DRS)	Documento de diseño arquitectónico (DDA)	Documento de diseño detallado (DDD) código Manual de usuario	Documento de transferencia SW Historia del proyecto
REVISIONES	Revisiones técnicas	Revisiones de documentos	Revisiones de documentos	Revisiones de documentos y	
PRINCIPALES hitos		aprobación DRU	aprobación DRS	aprobación DDA	aprobación DDD

Figura 6 - MODELO DE CICLO DE VIDA CONVENCIONAL DE LA ESA -

requisitos y es uno de los más graves en el proceso de diseño. Como consecuencia, impide realizar el seguimiento de los requisitos a lo largo del desarrollo de forma tal que facilite la evolución posterior del sistema.

c) Las **actividades de prueba** se realizan sobre el código cuando la relación con las decisiones de diseño se han perdido. Durante las pruebas de los módulos la situación es tolerable pero en las pruebas de integración surgen dificultades crecientes con la complejidad del sistema.

Casi todos los demás modelos han surgido como variaciones de éste. Una conocida variación es el **modelo en V** en el que los aspectos de prueba aparecen claramente ligados a cada una de las fases.

En este modelo se asocia una fase de prueba a cada una de las fases del ciclo de vida estableciendo métodos de gestión para la validación de cada una de ellas (conformidad). La Figura 7 describe esquemáticamente la idea de este modelo.

Visto globalmente, el modelo en V implica que las pruebas no se consideran parte de la implementación sino de forma separada relacionadas con cada una de las fases del desarrollo.

El problema de aplicación de este modelo reside precisamente en la capacidad que tengamos de asegurar la conformidad mediante pruebas sobre el código. Esta capacidad está ligada a la tecnología de software empleada en cada una de las etapas y que presentaremos en los Capítulos 3 y 4.

La aparición de tecnologías de software que permiten formalizar parcialmente la fase de análisis de requisitos de software o del diseño arquitectónico, está permitiendo completar la visión de este modelo combinando las pruebas mencionadas (a diferentes niveles pero sobre el código) con otras sobre los documentos de requisitos

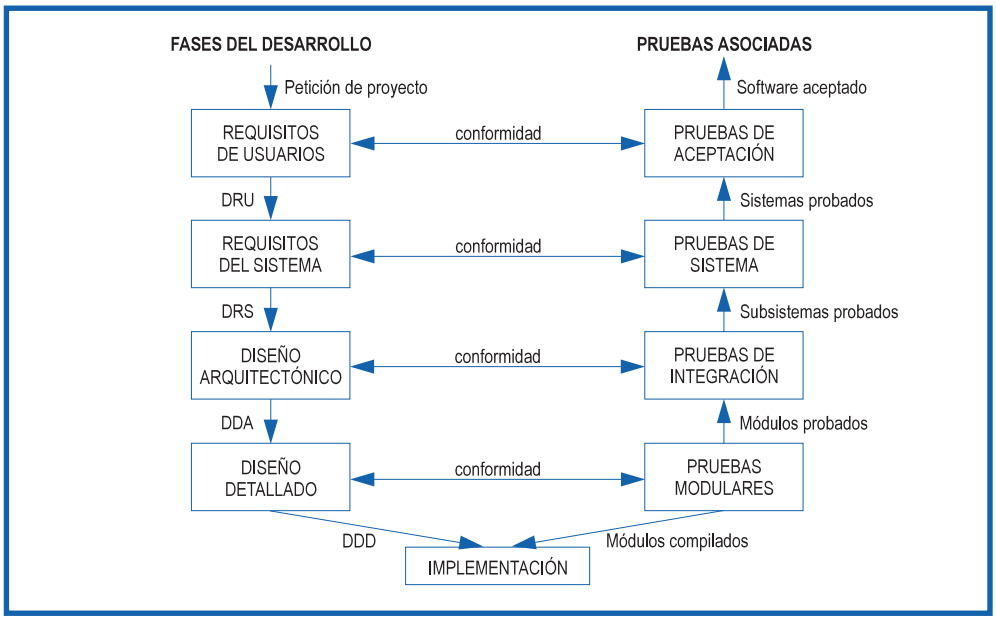


Figura 7 - MODELO DE CICLO DE VIDA EN V -

o diseño que mejoran globalmente la capacidad de validar el producto resultante.

2.4. Modelo incremental

El modelo de ciclo de vida en cascada obtiene el código como resultado de un proceso de refinamiento a partir de las especificaciones y diseño. Cuando el código es entregado al usuario es cuando éste dispone de un producto sobre el que puede evaluar la satisfacción de sus necesidades. En caso de que ésto no sea así, es necesario volver a las fases anteriores (costoso en tiempo y esfuerzo, sobre todo, cuando los problemas surgen de una mala interpretación de los requisitos de usuario).

Existe otro enfoque posible en el que al usuario se le van exponiendo productos intermedios denominados **prototipos** que le acercan

al sistema final. Estos productos intermedios sirven para validar con el usuario el sistema que se está construyendo antes de realizar la implementación.

Un **prototipo** puede definirse como un modelo parcial ejecutable de un sistema de software. Por **modelo del sistema** entendemos una descripción del sistema bajo una cierta perspectiva (por ejemplo, arquitectura, datos, comportamiento) empleando notaciones no necesariamente similares al código final; lo definimos como **parcial** porque no es necesario que cubra todo el sistema sino aquellas partes o perspectivas que se pretenden analizar; finalmente, debe ser **ejecutable** para que la validación del sistema pueda hacerse a partir de la experimentación con el prototipo por parte de usuarios y analistas.

Si es parcial, debemos conocer qué funcionalidad debe ser incluida en el prototipo. Para ello, existen dos enfoques básicos: construcción de un prototipo **vertical** en el que se presta atención a una parte pequeña del sistema pero prácticamente en su forma final y prototipo **horizontal** en el que la idea es obtener una visión global del sistema sin detallar o refinar ninguna de sus partes.

El prototipo vertical es importante cuando la especificación de requisitos demuestra problemas en conocer qué es lo que se desea sobre un aspecto muy concreto. El prototipo horizontal, por el contrario, pretende conocer mejor la estructura general de la interacción entre el usuario y el sistema de software a diseñar. Ambos son complementarios y pueden aparecer en un caso concreto de aplicación de las técnicas de prototipado.

Si pensamos en un prototipo de un avión y deseamos analizar la forma global que tendría, sin preocuparnos de detalles, estamos ante un prototipo horizontal; si, por el contrario, estamos interesados en la ergonomía (acceso a los mandos, por ejemplo) de la butaca del piloto, necesitamos el máximo detalle de ella y poco nos importa el resto del avión; en este caso, estamos ante un prototipo vertical.

También la técnica de prototipado puede clasificarse en función del uso que se va a hacer del prototipo a lo largo del desarrollo. Existen dos tipos básicos de **uso** del prototipo que seguidamente abordamos: modelo basado en prototipos desechables y modelo de prototipado incremental.

2.4.1. Modelo basado en prototipos desechables

El modelo con **prototipo desechable** aborda el problema de la inestabilidad de los requisitos generando un prototipo lo antes posible que sirva de base al mejor conocimiento de los requisitos de usuario. Este prototipo es desechado cuando usuarios y desarrolladores acuerdan un documento de requisitos de usuario. La Figura 8 describe este modelo. Podemos ver cómo apoya a la fase de análisis de requisitos (tanto de usuario como de sistema) permitiendo incrementar la confianza en que los requisitos identificados son los que realmente

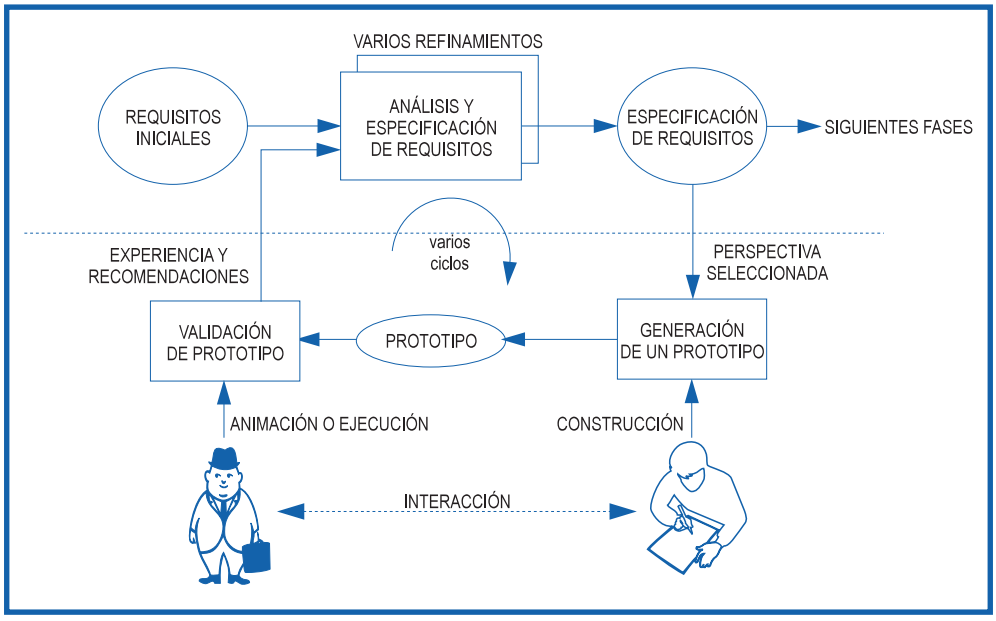


Figura 8 - MODELO DE CICLO DE VIDA CON PROTOTIPOS DESECHABLES -

desea el usuario y reducir el riesgo de problemas en la aceptación final del sistema de software.

De la Figura 8 se desprende que el prototipado puede entenderse como una mejora de la fase de análisis de requisitos con objeto de conocer mejor éstos y ayudar a usuarios y diseñadores a su definición. Concretamente, este modelo se ha demostrado muy útil para detectar **requisitos ocultos** (aquellos que el usuario no manifiesta explícitamente pero que al interaccionar con el sistema surgen de forma espontánea) y eliminar inconsistencias entre ellos.

Para algunos autores este modelo también es conocido como de **prototipado rápido** y podría tratarse independientemente del modelo incremental. No obstante, el que sea rápido es ortogonal al objetivo de uso (implica solamente que el usuario no puede esperar tanto tiempo como en el modelo en cascada para interaccionar con el sistema ya que si es así, se perderían las ventajas del prototipado) y la idea de desechable está ligada más a limitaciones de la tecnología empleada que a un interés real del usuario en «tirar» el trabajo realizado. Es justamente la aparición de nueva tecnología de software la que posibilita aprovechar parte del prototipo desarrollado y, de paso, analizarlo rápidamente.

2.4.2. Modelo basado en prototipado incremental

Si bien con el modelo basado en prototipos desechables podemos tener mayor confianza en que el producto que le entreguemos al usuario responda a los requisitos deseados por éste, es aún cierto que el producto, desde el punto de vista del diseñador, puede presentar muchos problemas que sólo serán evaluados cuando el producto se haya implementado. Concretamente, algunos requisitos funcionales no pueden evaluarse en un prototipo desechable porque la estructura del sistema final no es directamente extrapolable de la del prototipo (como puede ocurrir con los requisitos temporales o aspectos de

prestaciones). En ciertos casos, sí se pueden extrapolar tiempos de ejecución de determinados algoritmos que ayudan al diseñador en la toma de decisiones.

Más dificil es evaluar requisitos no funcionales porque la calidad del sistema final no está directamente relacionada con la del prototipo. Por ejemplo, la mantenibilidad es dificil de analizar en un prototipo porque dependerá de la forma en la que finalmente se implemente el sistema.

Un enfoque que está gozando de creciente popularidad dada la disponibilidad de tecnologías de software que lo facilitan es el uso del **prototipado incremental**. Se basa en la generación de varios modelos parciales ejecutables del sistema antes de proceder a la implementación (durante la especificación y durante el diseño) con el fin de evaluar sus características y poder obtener al final el sistema implementado. En estos prototipos no se pretende implementar partes del sistema final sino razonar sobre modelos ejecutables no implementados todavía.

Un **modelo de especificación** es una descripción de las funciones identificadas en el sistema y sus interrelaciones empleando generalmente una notación gráfica. En un **modelo de diseño** la descripción se completa incluyendo información sobre la descomposición en tareas y el uso de los recursos del S.O. para intercambiar información entre ellas y acceder a información común.

Los modelos ejecutables son «prototipos intermedios» generados para poder razonar sobre ellos. El concepto de ejecución en este caso es el de **animación del modelo**; por ello, entendemos la **visualización dinámica** de la evolución del mismo. Algunas herramientas software permiten, además, generar automáticamente interfaces gráficas con las que el usuario pueda interaccionar. Una vez que usuarios y analistas aceptan el sistema se continúa el desarrollo hacia las siguientes fases.

Otra línea de trabajo complementaria con la anterior consiste en partir de una funcionalidad mínima que se considera estable (base estable) y, apoyándose en ella, se realizan extensiones hasta llegar al producto final. En cada refinamiento se ejecuta la parte realizada para que el usuario pueda validarla (mediante ejecución de código real). La Figura 9 representa esquemáticamente el desarrollo de acuerdo con este modelo.

Una de las posibilidades que este proceso de ejecución de modelos tiene es la de permitir que un prototipo pueda describirse a diferentes niveles de abstracción combinando partes que estén descritas a un nivel de detalle cercano al código (o incluso código) con otras que aún corresponden a modelos más abstractos. Un prototipo que combine todas ellas en un sistema ejecutable se denomina **prototipo heterogéneo**.

A la técnica de desarrollo basada en la construcción de prototipos heterogéneos se le denomina **prototipado incremental**. En la

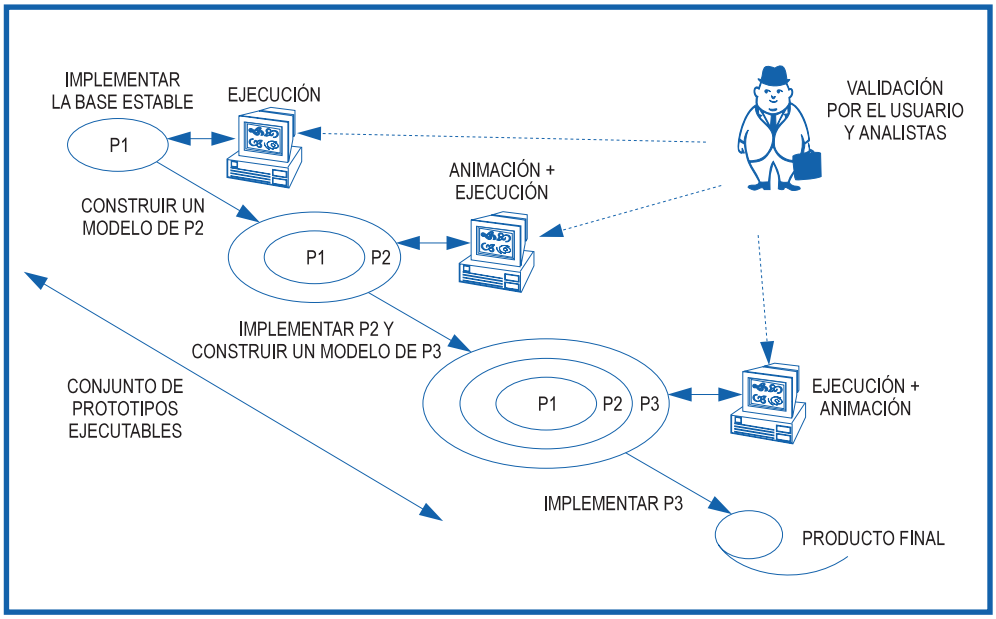


Figura 9 - MODELO DE CICLO DE VIDA INCREMENTAL -

Figura 9 las partes internas pueden estar ya implementadas (por ejemplo, P1) aunque las otras aún no lo estén. P2, por ejemplo, puede ser un modelo de diseño ejecutable que utiliza, a su vez, el código correspondiente a P1.

La Figura 10 representa esquemáticamente la estructura de un prototipo heterogéneo. Como vemos en la figura es posible tener prototipos heterogéneos combinando modelos de implementación (código) con modelos de diseño y otros que combinan modelos de diseño con modelos de especificación.

Obviamente, el concepto de «ejecución» en unos y otros es diferente. Para los modelos de especificación y diseño, la ejecución implica la **animación gráfica** del modelo mientras que la parte ya codificada requiere su **ejecución real** en una máquina. Las herramientas software de soporte deberán proveer de mecanismos para el intercambio de información entre unos y otros componentes.

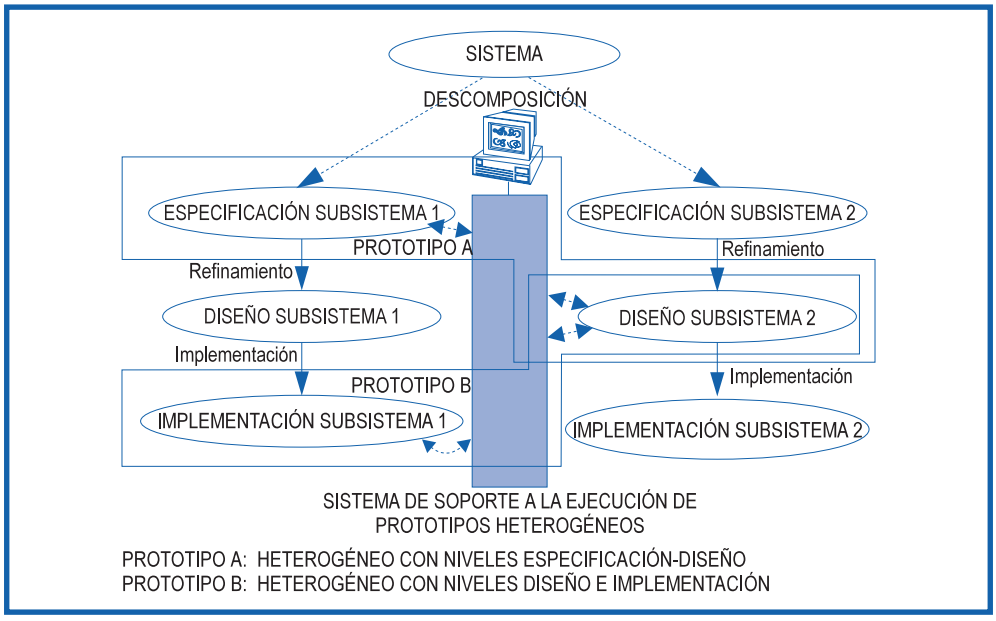


Figura 10 - PROTOTIPO HETEROGÉNEO -

Las ventajas del modelo incremental pueden resumirse de la siguiente manera:

a) Permiten incrementar la visibilidad del proceso de desarrollo mediante la experimentación con prototipos ejecutables intermedios.

b) La animación de modelos gráficos permite entender mejor el comportamiento dinámico del sistema y las interfaces hombre-máquina.

c) La documentación de las fases de análisis y diseño queda reforzada por los resultados del proceso de animación de los modelos facilitando las pruebas de aceptación.

d) No se «tira» nada. Los modelos realizados siguen siendo empleados en el siguiente prototipo o se convierten en un modelo con un nivel de detalle mayor (o incluso código).

No todo son ventajas. El disponer de prototipos intermedios ejecutables a partir de las notaciones empleadas en las fases de especificación y diseño tiene un coste que no podemos ocultar:

a) Requieren el empleo de métodos formales o semi-formales para ejecutar la descripción de la especificación y el diseño con sofisticadas herramientas gráficas.

b) Siguen existiendo dificultades para la evaluación de requisitos temporales. En el caso mejor, facilitan la obtención de estimaciones de tiempos de ejecución de código final con el fin de analizar aspectos de prestaciones.

c) Se mantienen las dificultades en la evaluación de requisitos no funcionales igual que en los prototipos desechables. En

este sentido, no es fácil definir y mantener una arquitectura del sistema a lo largo de los diferentes pasos incrementales.

2.5. Modelo de síntesis automatizada

Un modelo de ciclo de vida de síntesis automatizada es aquél que está basado en el empleo de una tecnología de software que permite generar automáticamente una implementación a partir de la especificación detallada de los requisitos de software del sistema (denominado especificación).

Este es el modelo de ciclo de vida que subyace bajo la utilización de métodos formales. Sin entrar en detalles sobre las tecnologías de software basadas en métodos formales (serán mencionadas en el siguiente Capítulo) vamos a indicar los aspectos básicos de este modelo. La Figura 11 representa esquemáticamente las fases seguidas en el desarrollo.

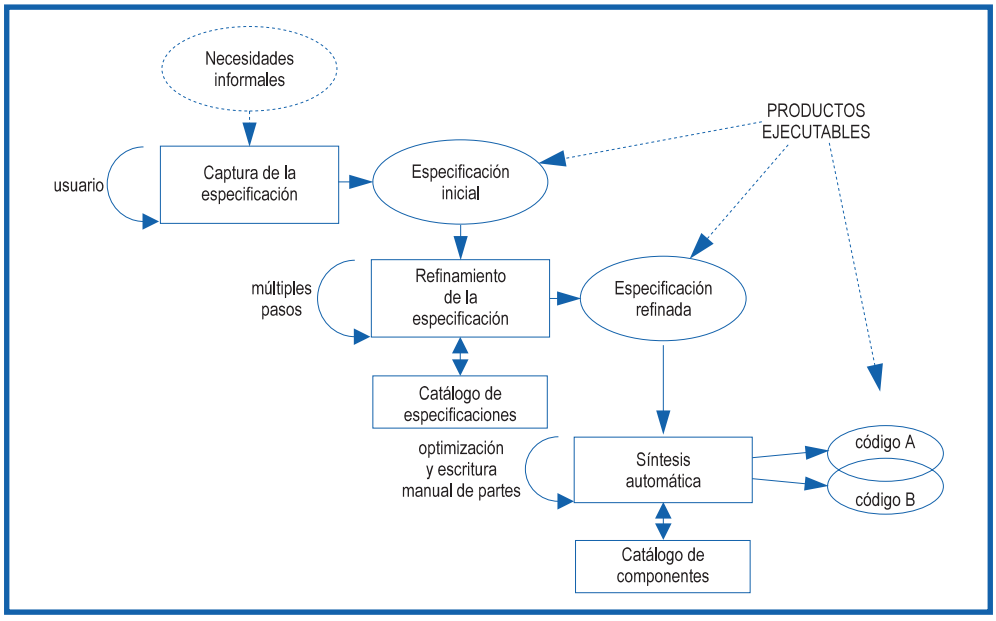


Figura 11 - MODELO DE CICLO DE VIDA BASADO EN SÍNTESIS AUTOMÁTICA -

En la primera de las fases, **captura de la especificación**, se genera una especificación formal definiendo el «servicio» que debe prestar el sistema de software y a partir de él se refina esta especificación introduciendo detalles progresivamente. En cada paso de refinamiento es posible comprobar que se mantienen determinadas propiedades con respecto al paso anterior; de esta manera, existe un alto grado de confianza en la validez del sistema final.

Cuando se ha obtenido una especificación suficientemente detallada, es posible generar el código final automáticamente (con ciertas ayudas manuales) mediante el empleo de herramientas software adecuadas. De esta forma, la fase residual de implementación es realmente una fase de optimización del código generado automáticamente.

Desgraciadamente, los métodos formales han tenido «mala prensa» en el mundo industrial provocando un cierto rechazo a su utilización. Varias han sido las causas que explican esta actitud. Por un lado, los promotores de los métodos formales han intentado promover su uso en el mundo industrial cuando la tecnología de software en la que se basaban era todavía inmadura. Por otro lado, los desarrolladores y gestores no se encontraban con los conocimientos y soporte adecuado como para adaptar sus bien conocidos procedimientos de trabajo (estimaciones de costes y recursos, mecanismos de control, etc.) al empleo de métodos formales.

Creemos que la situación se ha modificado profundamente en los últimos años. Los métodos formales ya existen de forma más o menos oculta soportando muchas de las herramientas y métodos comercializados hoy día haciendo que la polémica sea superada parcialmente por los hechos. Por otro lado, la formación de muchos de los ingenieros software actuales hacen más factible la incorporación de estas técnicas al mundo industrial. En el Capítulo 6 se analizará brevemente como la innovación en tecnologías de software depende entre otros factores de la formación de nuestros ingenieros.

A pesar de esta polémica podemos resumir las ventajas del modelo de ciclo de vida de síntesis automatizada de la siguiente manera:

a) Permiten incrementar fuertemente la confianza en el diseño fundamentalmente de aquellas partes críticas en las que un esfuerzo adicional bien vale la pena.

b) Facilitan las tareas de mantenimiento y evolución del sistema al poder actuar sobre las decisiones de diseño y no sobre el código generado. El mantenimiento se realiza sobre la especificación y no sobre el código.

c) Permiten documentar los sistemas de software de una manera más completa y precisa. Esta posibilidad está llevando a que las especificaciones constituyan productos en sí mismos. Organismos internacionales de normalización (por ejemplo, en telecomunicaciones) publican las especificaciones formales de un servicio o protocolo que los fabricantes deben satisfacer a la hora de desarrollar sus productos junto con la descripción textual convencional.

Si a pesar de estas ventajas, su uso industrial es limitado, ello es debido a problemas entre los que citamos:

a) Si bien parte del código puede generarse automáticamente, éste no cubre todo el sistema de software por lo que una parte no despreciable (fundamentalmente la interacción con el exterior) debe seguir realizándose mediante métodos convencionales.

b) No es posible obtener una seguridad absoluta en el desarrollo porque hoy día las técnicas de verificación no pueden aplicarse a sistemas grandes. Esto implica que los métodos formales deben coexistir con los convencionales. En esa coexistencia se pierden parte de las ventajas.

c) Se requieren conocimientos y herramientas no totalmente soportados desde el punto de vista comercial. Muchas de ellas son prototipos generados en centros de investigación (o pequeñas empresas relacionadas con ellos) que requieren esfuerzos no despreciables para su uso práctico en el desarrollo de sistemas de software de tamaño grande.

d) La formación requerida es también muy diferente de la que se encuentra ampliamente extendida entre los ingenieros software actuales. La mayor parte de ellos no ha visto estas técnicas en su formación reglada.

2.6. Meta-modelo en espiral

Hace unos años, Boehm propuso un modelo de desarrollo software capaz de acomodar muchos de los modelos definidos anteriormente [8]. Debido a ello se le suele conocer como meta-modelo en espiral.

La Figura 12 representa la estructura general de este modelo. En él podemos ver cuatro cuadrantes ligados a actividades genéricas de planificación, desarrollo, evaluación y selección de alternativas.

El desarrollo del sistema pasa por una serie de ciclos en los que tanto el conocimiento del sistema a realizar como el propio sistema van avanzando hasta obtener el producto final. En la última espiral se prosigue con un desarrollo convencional al haberse eliminado las incertidumbres en las espirales anteriores.

La utilización de la espiral sugiere un avance en el tiempo del desarrollo del producto y también un incremento paulatino del coste. Las espirales son progresivamente más costosas (como lo es también el coste acumulado).

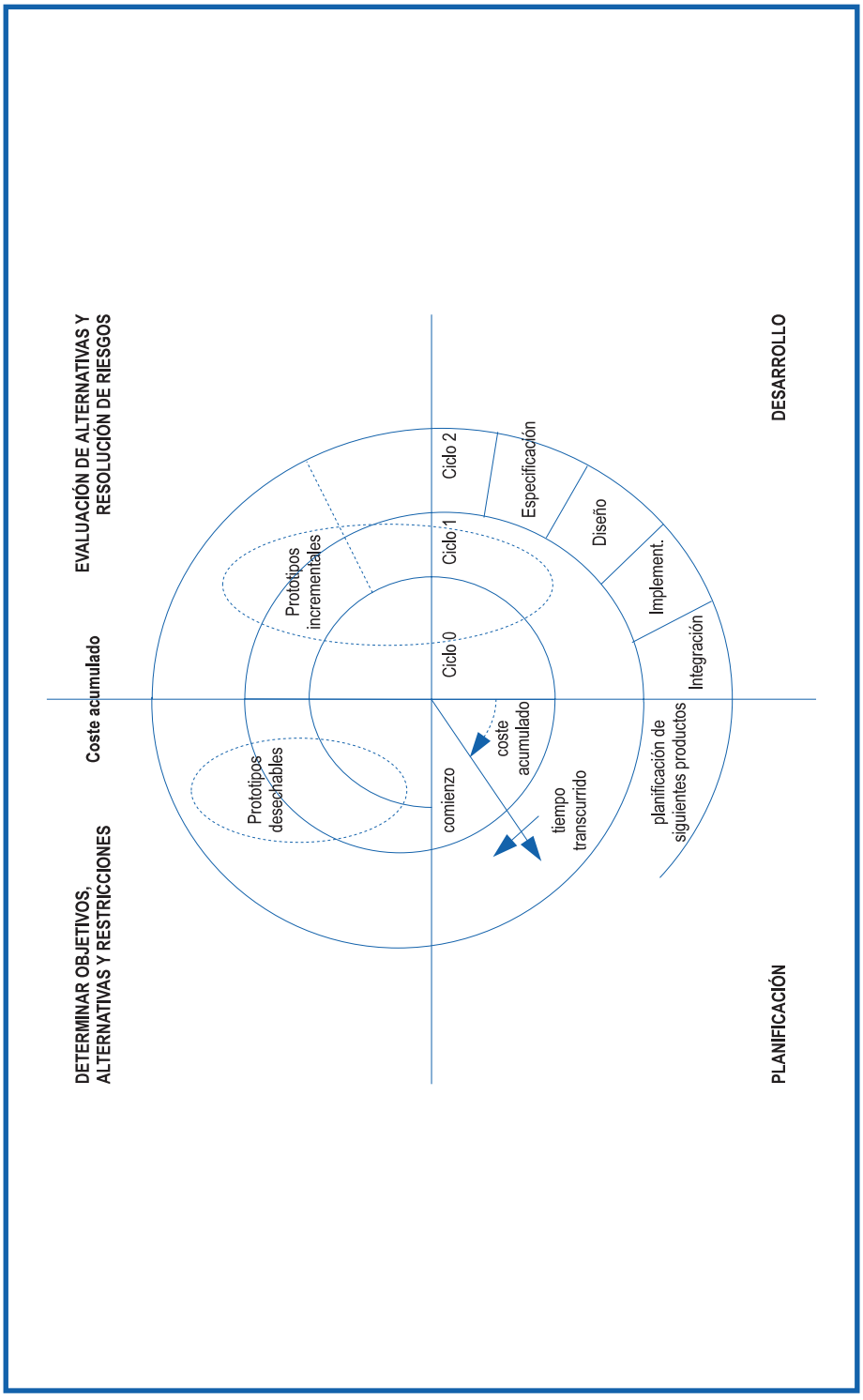


Figura 12 - MODELO DE CICLO DE VIDA EN ESPIRAL -

La idea clave detrás del modelo es asegurar que los aspectos menos conocidos o más críticos sean realizados antes, en la suposición de que de poco sirve completar partes poco críticas si éstas pueden verse modificadas o invalidadas por las partes críticas. El uso de prototipos es fuertemente promovido (aunque basado en prototipos desechables). En este sentido, el mismo Boehm indica que la característica esencial es la orientación hacia la identificación y resolución de riesgos [9].

Los primeros ciclos están pensados para asegurar una correcta comprensión del sistema y de sus requisitos, relegando cualquier implementación al momento en el que todos los factores de riesgo hayan sido eliminados. En el Capítulo 5 volveremos sobre la gestión de riesgos dentro de las técnicas de gestión.

No ha sido fácil utilizar en la práctica industrial este modelo. Las dificultades estriban en las implicaciones resultantes de una gestión del desarrollo muy diferente de la convencional y no sólo en la disponibilidad de tecnologías de software asociadas. Por otro lado, fue pensado para proyectos muy grandes y no es evidente su utilidad para sistemas pequeños dados los grandes recursos en gestión que requiere.

En la Figura 13 podemos ver un despliegue del modelo en espiral en el que tres secuencias de desarrollo se solapan en el tiempo. Comenzando con la parte más arriesgada, los diseñadores relegan hasta los siguientes ciclos las partes con riesgo moderado o bajo. Podemos observar también cómo en ciertos momentos es posible crear algún prototipo heterogéneo que incorpore elementos con diferentes niveles de abstracción.

Hemos representado en la Figura 13 dos posibles puntos de interacción con los usuarios. El primero implica la ejecución de un modelo parcial del sistema referente a las especificaciones de la parte de menor riesgo del sistema de software; el usuario podría conocer hasta que punto sus requisitos están contemplados en lo realizado hasta el momento. El segundo es un prototipo heterogéneo en el que

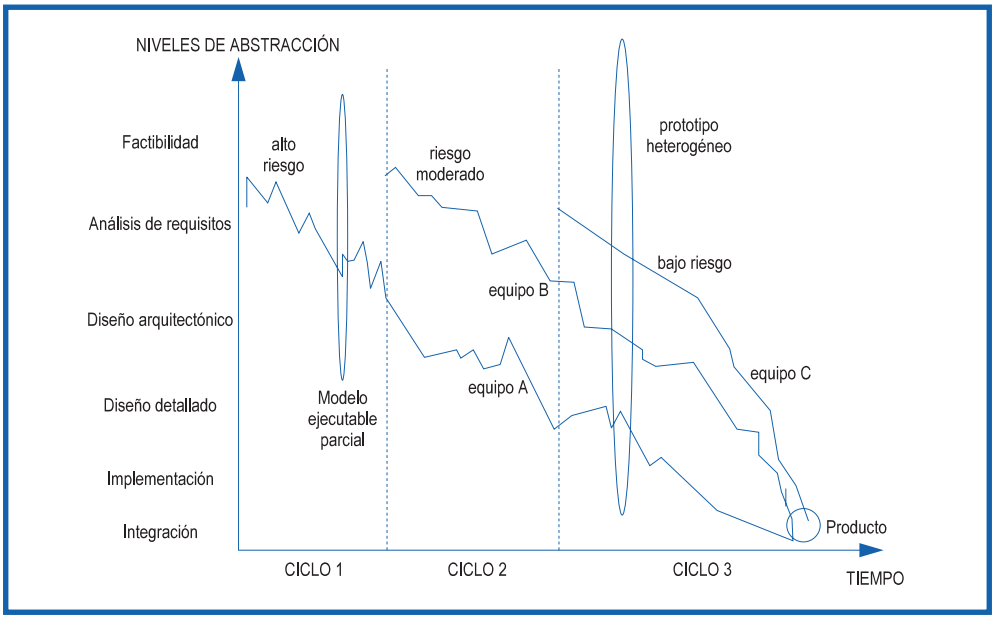


Figura 13 - ESPIRALES -

se ha integrado partes ejecutables de todo el sistema y puede obtenerse una idea global.

La figura también sugiere que el desarrollo se realiza mediante varios equipos de trabajo que no actúan sobre la misma fase del desarrollo (como sucede en una organización convencional). Esto permite encajar el concepto de prototipado incremental (con el empleo de prototipos heterogéneos) con el modelo en espiral. Con la extensión de las técnicas de prototipado incremental y de gestión de riesgos es previsible un incremento del uso del modelo en espiral incluso en proyectos pequeños.

2.7. Resumen

Los modelos de ciclo de vida analizados en este Capítulo suponen un marco de referencia fundamental para gestionar y controlar el proceso de desarrollo y evolución.

Los diferentes tipos de sistemas de software pueden requerir diferentes modelos de ciclo de vida. Podemos afirmar que todos los modelos del proceso de desarrollo mencionados en el presente Capítulo se han empleado en la práctica. Parece, por tanto, necesario que el diseñador seleccione el modelo más adecuado a sus intereses. Pero: ¿cuál es el más adecuado para un sistema de software concreto?

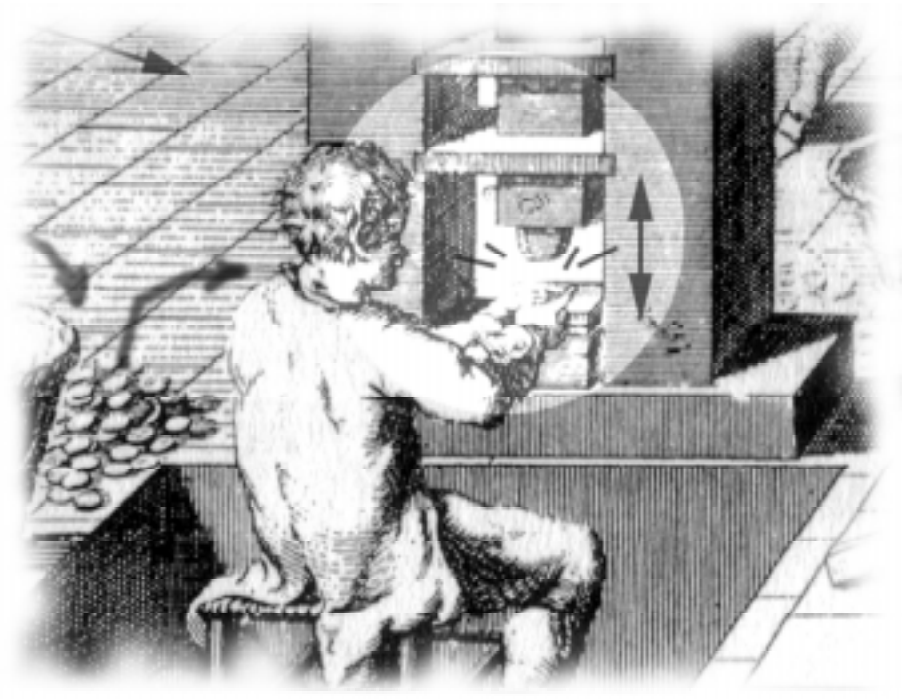
La pregunta es inadecuada porque no depende sólo del tipo de sistema a desarrollar sino también de la organización encargada de ello. Si la organización no posee procedimientos estables de gestión algunos modelos de ciclo de vida pueden ser inadecuados. Un ejemplo de este problema puede aparecer en el caso del prototipado si la organización no acepta la interacción con el usuario durante el proceso de desarrollo; otro ejemplo lo tenemos en el modelo en espiral si no se dispone de procedimientos de control de riesgos.

Podemos resumir diciendo que la selección de un modelo de ciclo de vida depende de la importancia relativa que tienen para la organización de desarrollo diferentes conceptos como el conocimiento previo sobre el dominio del problema (si es alto puede no requerirse el prototipado), la inestabilidad de los requisitos (que haría arriesgado un modelo en cascada), el ámbito de aplicación (que puede requerir técnicas formales), etc..

Queremos indicar, finalmente, que la elección de un modelo concreto no es gratuita; tanto la tecnología de software (descrita en los Capítulos 3 y 4 de esta monografía) como los procedimientos de gestión (desarrollados en el Capítulo 5) están íntimamente ligados al modelo elegido. Y cambiar de un modelo a otro (ver el Capítulo 6) no puede hacerse sin coste en tiempo y recursos.

3

Tecnologías de software



3.1. Introducción

Hemos dedicado los dos Capítulos anteriores a ofrecer un marco general del desarrollo de sistemas de software; es hora de que nos adentremos en las técnicas que nos permiten su desarrollo con una calidad adecuada.

Ni el tipo de sistema de software a desarrollar (ver Capítulo 1) ni los modelos de ciclo de vida (ver Capítulo 2) son tan similares como para permitir desarrollar cualquier sistema de software, empleando el mismo tipo de tecnología. Una de las principales funciones del ingeniero de software es la de seleccionar las tecnologías que mejor se adecúan al producto a realizar y al proceso de desarrollo utilizado.

En este Capítulo nos será imposible describir en detalle las decenas de tecnologías (lenguajes, herramientas, métodos, etc.) que se han desarrollado; ni siquiera podremos mencionar todas ellas. Las limitaciones de espacio y, más importante, la necesidad de presentar grandes tendencias como única forma de no perderse en los detalles, nos aconseja seguir un esquema matricial; esto es:

1) Describir la tecnología de software por medio de sus componentes (ver Sección 3.2) y

2) Tomar como referencia la fase del ciclo de vida considerado dentro de un modelo de ciclo de vida en cascada.

Deliberadamente, no consideraremos el tipo de sistema de software como una coordenada adicional; no obstante, en el siguiente Capítulo, utilizaremos los sistemas de tiempo real como un marco de referencia sobre el que ofrecer ejemplos concretos de aplicación de tecnología de software concreta a este tipo de sistemas dada su importancia industrial.

3.2. Concepto de tecnología de software

Definimos **tecnología de software** como un conjunto integrado de notaciones, herramientas y métodos, basados en unos sólidos fundamentos, que permiten el desarrollo de un producto software en un contexto organizativo dado.

Una tecnología de software puede considerarse constituida por los siguientes componentes (ver Figura 14):

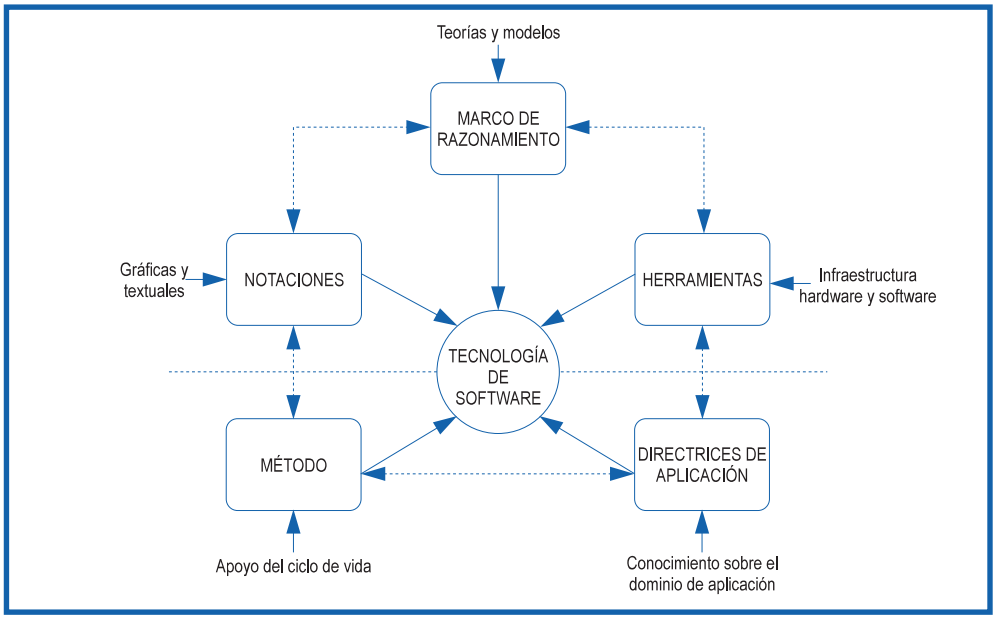


Figura 14 - LA TECNOLOGÍA DE SOFTWARE Y SU CONTEXTO -

A) **Marco de razonamiento**. Por marco de razonamiento nos referimos al conjunto de conceptos y mecanismos que una tecnología de software posee para asegurar que el sistema en desarrollo satisfaga las propiedades que se deseen. Se basa en la existencia de unos conceptos rigurosos y bien relacionados o, mejor aún, de un modelo matemático para representar la ejecución de un programa. Este modelo matemático, convenientemente manipulado, permite obtener información sobre el sistema en construcción.

Idealmente, una tecnología de software debería permitir asegurar a lo largo de todo el proceso de desarrollo que el sistema satisfaga los requisitos exigidos, tanto funcionales como no funcionales. Desgraciadamente, esto no sucede así con las tecnologías actuales. Por un lado, el razonamiento en las fases iniciales del ciclo de vida exige una precisión en la descripción del sistema de la que se carece con las técnicas usualmente utilizadas (en todo caso, se dispone de descripciones funcionales); por otra, las decisiones tomadas en la etapa de diseño arquitectónico se pierden en la maraña de decisiones de bajo nivel que es necesario adoptar en la fase de implementación (unas por razones de eficiencia, otras por las propias limitaciones de los lenguajes empleados).

Pocos diseñadores hacen uso explícito del marco de razonamiento que les ofrece la tecnología que utilizan. Casi siempre se relega en la funcionalidad de las herramientas software empleadas.

B) **Notaciones**. Lenguajes para poder describir el sistema en desarrollo. En el desarrollo de un sistema complejo coexisten diversas notaciones empleadas en las diferentes fases del modelo de ciclo de vida seleccionado dado que no es posible con una única notación cubrir las necesidades de cada una de las fases.

Contar con la notación adecuada constituye además el vehículo para poder razonar sobre el sistema en desarrollo. Esta es la razón por la que ambas cosas (notación y marco de razonamiento) se confunden en la práctica del desarrollo de software. Todos los lenguajes ejecutables empleados poseen una definición semántica con la que es posible determinar si una descripción concreta es correcta.

C) Herramientas. Un sistema implementado implica un contrato con la máquina sobre la que se ejecuta. Este contrato fuerza a disponer de sistemas software que traduzcan la descripción efectuada por el diseñador en otra adaptada para la máquina y generada automáticamente a partir de una descripción de más alto nivel.

Esta necesidad, surgida desde los comienzos de la informática (y motivada por la necesidad de alejarse de los lenguajes de la máquina) no sólo requiere del desarrollo de un traductor (compilador o intérprete) sino también de un conjunto de herramientas software que soporten diferentes actividades durante el desarrollo de un sistema: editores sensibles al lenguaje, depuradores, optimizadores, generadores de casos de prueba, etc. Estas herramientas no son tampoco elementos aislados puesto que sus entradas y salidas son también salidas y entradas de otras herramientas llegando a constituir **entornos integrados** de herramientas. Estas herramientas son dependientes de la infraestructura de ejecución (hardware/software) como se expuso en el Capítulo 1.

D) Método de desarrollo. Disponer de notaciones y herramientas no implica que los diseñadores conozcan cómo diseñar un sistema de relativa complejidad. Ello implica también disponer de procedimientos para pasar de los

requisitos al diseño y de éste a la implementación, aprovechando las notaciones y herramientas disponibles y sabiendo cómo dividir el trabajo entre los componentes del equipo humano de desarrollo.

Los métodos proporcionan una disciplina en el proceso de refinamiento que guía al diseñador a lo largo de varias fases, desde la descripción de los requisitos en lenguaje natural hasta su completa especificación. Posteriormente, a esa especificación se le añaden las decisiones de diseño continuando el proceso de refinamiento hasta poder implementar el sistema en un lenguaje convencional.

En este sentido, los métodos no son independientes del modelo de ciclo de vida elegido ya que tienen que soportar el desarrollo a lo largo de algunas de sus fases y proporcionar los heurísticos de refinamiento asociados.

E) Directrices de aplicación industrial. Las peculiaridades de un dominio de aplicación quedan reflejadas en conjuntos de soluciones probadas y difundidas entre la comunidad de diseñadores para aspectos parciales de los sistemas requeridos. El conocimiento del dominio de aplicación se concreta en conceptos, elementos, interconexiones y datos que solucionan aspectos concretos. Estas soluciones toman la forma de módulos ampliamente utilizados y, por tanto, reutilizables, patrones de diseño de gran aceptación e incluso formas de utilizar los lenguajes y herramientas adaptados al dominio de aplicación considerado.

En muchos dominios de aplicación se han consolidado bibliotecas de componentes (por ejemplo, funciones matemáticas o para realizar interfaces gráficas) reutilizables que simplifican y reducen el tiempo de

desarrollo. Podemos decir que el saber-hacer de una organización en un determinado dominio de aplicación se manifiesta en su capacidad para buscar soluciones eficientes para los productos en desarrollo dentro de ese dominio.

Obsérvese que son las relaciones entre las componentes las que permiten hablar de **una** tecnología de software. Sus componentes son como piezas de un «puzzle» que deben encajar unas en otras. En otras palabras, la relación entre los componentes implica la existencia de una imbricación entre ellas basada en la integración conceptual de los elementos de cada componentes y no simplemente del uso que cada diseñador le dé; globalmente deben contribuir al desarrollo del sistema de software de una forma coordinada.

En el lenguaje cotidiano, algunas veces se habla de una tecnología de software cuando sólo afecta a una parte del desarrollo. Esta situación es común cuando las notaciones, herramientas, etc., sólo permiten abordar el desarrollo en una de las fases. Para el resto de las fases deberían utilizarse otras «tecnologías». Esta visión de ámbito reducido de la tecnología implica que para el desarrollo completo de un sistema será necesario utilizar diversas tecnologías de software de ámbito reducido encadenadas.

De la discusión anterior se desprende que el **impacto** de una tecnología de software (o su alcance) no siempre es el mismo. Con independencia de las consecuencias que eso tendrá para la adopción de nuevas tecnologías (Capítulo 6), el impacto nos permite analizar la complementariedad entre tecnologías de software.

El impacto de la tecnología se puede entender desde tres perspectivas complementarias: fases del ciclo de vida soportadas, rango de sistemas en los que es aplicable y perfiles técnicos o equipos de desarrollo afectados por la misma.

La Figura 15 refleja esta visión en dónde se representan tres tecnologías diferentes. La tecnología A sólo soporta las fases finales del ciclo de vida (implementación y pruebas asociadas), su utilización está ligada a un único componente del equipo de trabajo y su utilización es válida para un tipo muy concreto de sistemas (por ejemplo, sistemas de información (SI)). Por el contrario, la tecnología C tiene un impacto mucho mayor al afectar a varias fases, implicar a personas con diferentes perfiles técnicos y poder aplicarse a un rango mucho mayor de sistemas (por ejemplo, sistemas distribuidos (DIST)).

Finalmente, la tecnología B se encuentra en una posición intermedia enfocada a las fases de diseño e implementación (por ejemplo, para Sistemas de Tiempo Real (STR)).

Si consideramos el proceso de desarrollo en su conjunto, puede ser necesario emplear diferentes tecnologías adaptadas a diferentes fases del ciclo de vida. De esta manera, se utilizan progresivamente notaciones, herramientas, métodos y formas de razonar sobre el sistema dentro de un dominio de aplicación dado aportados por las diferentes tecnologías consideradas. Cada actividad, asimismo se detalla en un conjunto de procesos cuya imbricación, controles, entradas y salidas permiten obtener el sistema deseado.

La Tabla 1 relaciona estos elementos con las fases de un ciclo de vida. En cada una de las celdas de la tabla indicamos el objetivo del componente.

3.3. Panorama de los componentes tecnológicos

Antes de analizar en el Capítulo 4 el empleo de algunas tecnologías concretas para el caso de los Sistemas de Tiempo Real, vamos a describir la situación actual en cada una de las componentes indicadas. Ello nos permitirá analizar la situación en la que se encuentran y su posible evolución futura.

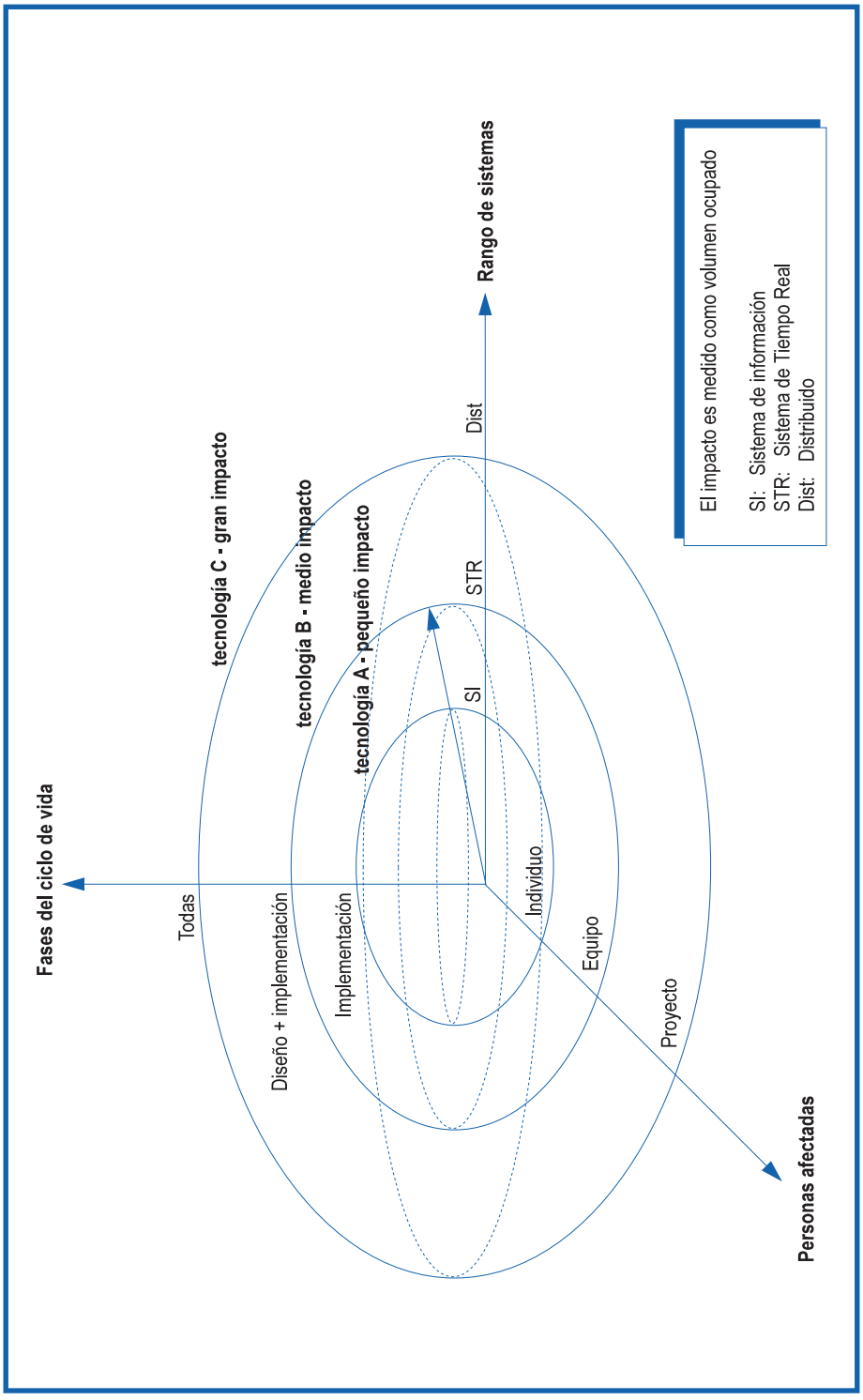


Figura 15 - IMPACTO DE LA TECNOLOGÍA DE SOFTWARE -

COMPONENTE	ESPECIFICACIÓN DE REQUISITOS	DISEÑO ARQUITECTÓNICO	DISEÑO DETALLADO	IMPLEMENTACIÓN
NOTACIONES	COMPLETAR REQUISITOS	DESCRIBIR LAS INTERACCIONES	DESCRIBIR LA SOLUCIÓN	SOLUCIÓN EJECUTABLE
MARCO DE RAZONAMIENTO	VALIDAR LA CONSISTENCIA	VALIDAR REQUISITOS NO FUNCIONALES	TRAZAR REQUISITOS	VERIFICAR PROPIEDADES
HERRAMIENTAS	DESCRIBIR LOS REQUISITOS EN LA NOTACIÓN SELECCIONADA	SOPORTAR REFINAMIENTOS Y ANIMAR LOS MODELOS	GENERACIÓN Y PRUEBA DEL DISEÑO	OBTENER CÓDIGO EJECUTABLE PARA UNA MÁQUINA
MÉTODO DE DESARROLLO	OBTENER REQUISITOS DE USUARIO Y SISTEMA	INCLUIR DECISIONES DE DISEÑO	USO DE COMPONENTES GENÉRICOS	INCLUIR REQUISITOS DE PRESTACIONES
DIRECTRICES INDUSTRIALES	COMPRENDER EL DOMINIO DE APLICACIÓN	IDENTIFICACIÓN DE MÓDULOS E INTERFACES	PROPORCIONAR MÓDULOS GENÉRICOS	REUSABILIDAD DE COMPONENTES

Tabla 1 - OBJETIVOS DE LOS COMPONENTES TECNOLÓGICOS -

3.3.1. Notaciones

Casi siempre, la visión más próxima al desarrollo de sistemas de software la ofrecen los lenguajes de programación. Todo lo demás está subordinado a ellos puesto que el objetivo último es disponer de una descripción ejecutable (entendida por una máquina física) y eficiente. Podríamos decir que la evolución de la informática durante muchos años perseguía encontrar lenguajes cada vez más cercanos al problema del usuario y más alejados de la estructura de la máquina en la que se ejecuta sin perder eficiencia en esa ejecución.

No deseamos resumir aquí la evolución de los lenguajes de programación sino destacar aquellas propiedades de interés general desde el punto de vista de la ingeniería de sistemas de software. Fijaremos nuestra atención fundamentalmente en construcciones ideadas para el desarrollo de sistemas grandes [2,10].

Si bien cualquier lenguaje de programación puede permitirnos describir cualquier sistema, no todos son adecuados para el desarrollo de sistemas grandes. Tal y como se ha expuesto en el Capítulo 1, el desarrollo de un sistema de gran tamaño requiere un grupo numeroso de personas. La distribución del trabajo entre ellos se realiza a partir de la descripción de las interfaces y de las bibliotecas de componentes utilizadas.

Aunque las notaciones empleadas en las fases iniciales del ciclo de vida son muy diferentes de las que se emplean en la fase de implementación, algunas características son comunes a todas ellas. Mencionaremos seguidamente las más importantes:

A) **Abstracción**. Inicialmente considerada una característica útil de los lenguajes para poder describir un sistema sin necesidad de tener que conocer detalles de la máquina en la que se va a ejecutar, su utilización en la ingeniería de sistemas de software es aún más importante en las primeras fases del ciclo de vida.

Las notaciones (muchas de ellas gráficas) empleadas durante la especificación de requisitos no pretenden definir cómo se hace algo (aunque sea de una forma muy abstracta) sino expresar qué es lo que el sistema debe hacer. Se abstrae, por tanto, no sólo desde el punto de vista de la estructura de la máquina sino de la solución adoptada.

B) Generalización. Los mecanismos de abstracción son necesarios para soportar las diferentes fases del ciclo de vida. En muchos casos, este soporte se logra reduciendo el ámbito de aplicación o las características del sistema que son fácilmente descritas. Para que el lenguaje sea útil en diferentes dominios de aplicación y para diferentes tipos de sistemas debe disponer de construcciones de propósito general.

No obstante, cuando el dominio de aplicación es muy importante (industrial o económicamente) como sucede con los Sistemas de Tiempo Real, es normal encontrarse con construcciones especializadas que faciliten su descripción.

C) Potencia expresiva. Un tipo determinado de sistema para un dominio de aplicación dado, utiliza un conjunto de conceptos (objetos, operaciones o relaciones) característicos de ese dominio. Si las notaciones (lenguajes de programación o de diseño) con las que se describen estos sistemas poseen construcciones cercanas a las que los sistemas necesitan, el diseñador puede expresar fácilmente estos conceptos en el lenguaje elegido. Si un lenguaje permite describir fácilmente sistemas en un dominio de aplicación dado, se dice que posee **potencia expresiva** suficiente en ese dominio.

Pero hay otra visión de la potencia expresiva muy importante desde la perspectiva de ingeniería de sistemas de software:

el lenguaje debería facilitar el desarrollo de sistemas grandes por un equipo de trabajo numeroso.

Desde esta perspectiva, el lenguaje debería facilitar la descomposición en unidades modulares (módulos) lo más independientes posibles (para que la inteacción entre los diseñadores sea mínima), separar la interfaz (lo que otros módulos pueden conocer) del cuerpo de los módulos (determinar por tanto lo que es propio de cada uno y oculto a los demás), permitir la incorporación de bibliotecas de funciones (comunes a muchos módulos), permitir operaciones sobre datos persistentes (aquellos que deben mantenerse después de la ejecución), etc.

D) **Eficiencia**. Una notación debe poseer construcciones que permitan a los compiladores generar código ejecutable eficiente para que en la ejecución del programa se puedan aprovechar las características del ordenador sobre el que se ejecute.

Así, la eficiencia puede abordarse con un buen diseño de los compiladores y la forma en la que aprovechan el conocimiento de la máquina (en la fase de optimización del código). Los diseñadores de lenguajes deben preocuparse, no obstante, de que éste sea fácilmente traducible (los primeros compiladores de Ada, por ejemplo, fueron muy ineficientes porque el tratamiento de «tareas (tasks)» no era sencillo). Desde este punto de vista, poco puede hacer el diseñador y está, por tanto, fuera de la ingeniería de sistemas de software.

3.3.2. Marco de razonamiento sobre el sistema en desarrollo

Utilizar una u otra notación no implica disponer de procedimientos para conocer que el sistema descrito es correcto. Generalmente, lo

único que podemos saber asociado a la notación (cuando el programa se compila o procesa) es que la descripción es correcta con respecto a la definición sintáctica (está bien escrito) y semántica (por ejemplo, los tipos de datos se corresponden adecuadamente) del lenguaje.

¿Qué debemos conocer de un lenguaje concreto? La mayor parte de los diseñadores o programadores (dependiendo de las notaciones empleadas) conocen la sintaxis y una descripción semántica informal. No obstante, el conocimiento profundo de un lenguaje permitiría comprobar ciertas propiedades del sistema descrito manipulando la descripción a través de herramientas de software apropiadas.

Es evidente que los lenguajes de implementación así lo hacen puesto que en caso contrario no podrían construirse compiladores. Sin embargo, no es tan evidente para los lenguajes de especificación de requisitos y diseño; en ellos, se dispone de una notación gráfica y unas reglas más o menos detalladas de cómo utilizarlas incluidas en el método de desarrollo elegido.

Mencionaremos seguidamente algunos de los aspectos más importantes ligados al razonamiento sobre un sistema que un diseñador debería poder hacer con el fin de incrementar la confianza en el sistema que está diseñando.

A) Mantenimiento de alguna propiedad deseable entre dos pasos de refinamiento consecutivos. Esta sería la gran aportación para incrementar la confianza en el proceso de desarrollo. Actualmente, esa comprobación se hace al final de un largo proceso de refinamiento.

B) Comprobación de la consistencia de datos (nombres, tipos, uso) entre niveles de refinamiento. Recuérdese que si se realiza la descripción por un equipo de personas, la consistencia ha de comprobarse entre todos.

C) Análisis temporal a lo largo del desarrollo. Implica la capacidad de asegurar la satisfacción de restricciones temporales sin necesidad de esperar a conocer tiempos de ejecución del código.

D) Análisis de la corrección en la sincronización y comunicación de información entre procesos concurrentes (por ejemplo, ausencia de bloqueo). Útil para sistemas concurrentes en los que las relaciones entre tareas (o procesos del sistema operativo) deben asegurarse con independencia de la velocidad relativa de ejecución de las mismas.

E) Análisis de prestaciones con anterioridad a disponer del código con el fin de identificar elementos críticos y actuar sobre ellos.

Todas estos aspectos deben ser abordados mediante métodos formales o semi-formales. En el caso del empleo de técnicas convencionales sólo es posible hacer algo en la fase de implementación y confiar para las fases iniciales en la disciplina que un método concreto puede imponer al equipo de desarrollo. En los últimos años, no obstante, se ha producido un esfuerzo considerable en dotar a las notaciones empleadas en las primeras fases de la suficiente formalización para poder cubrir los objetivos anteriores aunque solo sea parcialmente.

3.3.3. Métodos de desarrollo

Disponer de un lenguaje o de un conjunto de ellos (incluso cuando poseen un marco de razonamiento que permita manipular con seguridad las descripciones, no implica que se pueda abordar el desarrollo de un sistema complejo. En la década de los sesenta la mayor parte de las construcciones básicas de los lenguajes actuales (o antecesores de los mismos) estaban disponibles; sin embargo, el desarrollo de sistemas de software grandes resultó caótico al no existir

procedimientos claros que permitiesen generar los requisitos del sistema o derivar un diseño concreto que los satisficiese. Comenzar a programar pronto no aseguraba nada. Se necesitaba un método, una disciplina en el trabajo. Este requisito era tanto más necesario cuanto más complejo fuese el sistema y mayor fuera el número de personas implicadas.

Un método aporta una forma sistemática de refinar las especificaciones de un sistema haciendo que en cada paso se obtenga cierto nivel de confianza en que el refinamiento efectuado sea correcto. Es este incremento en el nivel de confianza, tanto a nivel individual como a nivel organizativo, la aportación básica de un método (otros autores la asocian con la disciplina en el trabajo que lleva aparejada).

La forma que un método tiene para lograr el objetivo de permitir incrementar la confianza del diseñador es imponer una **disciplina** en el proceso de desarrollo conjugando la utilización de una o varias notaciones y formas de razonar sobre el sistema en desarrollo con un conjunto de directrices que guían al diseñador en el proceso y generalmente apoyados por unas herramientas que soportan el método. Sus objetivos concretos son:

1) Proponer un procedimiento para capturar los requisitos del usuario y relacionarlos entre sí para facilitar la comprobación de su consistencia.

2) Distribuir el desarrollo entre un equipo de trabajo mediante la adecuada agrupación de funciones en estructuras de diseño (objetos, módulos multifuncionales, etc.).

3) Identificar interfaces claras entre los componentes del sistema a diseñar (objetos, módulos, etc.).

4) Proponer una serie de heurísticos para guiar el refinamiento en varias etapas asegurando la consistencia en cada uno

de los pasos de refinamiento basados en la experiencia de los proponentes del método en diseñar sistemas reales con el mismo.

Cada método existente pretende cubrir los objetivos anteriores generalmente limitando el ámbito de actuación ya sea por la fase o fases del ciclo de vida cubiertas, por el tipo de sistema para el que esté destinado, o combinación de ambos. Así, podemos encontrar métodos que cubren desde el análisis hasta la implementación de un sistema y otros que restringen el ámbito de actuación.

Se ha prestado mucha atención a métodos que cubren el análisis de los requisitos funcionales de un sistema y que apoyan al diseñador en el proceso de refinamiento hacia las fases de diseño. Desgraciadamente, estos métodos ofrecen mucho menos soporte para las fases de diseño detallado por lo que el paso hacia la implementación siempre implica un esfuerzo considerable que debe ser cubierto por el diseñador.

La difusión en la práctica de un método suele implicar también la de las notaciones diseñadas con él (o el uso de notaciones preexistentes) que, aunque consideradas componentes tecnológicos diferenciados, aparecen ligadas en el mercado a la difusión de los sistemas CASE (Ingeniería Software Ayudada por Ordenador).

3.3.4. Herramientas de soporte: entornos de desarrollo

Este es uno de los campos en el que más se ha trabajado últimamente debido a dos circunstancias coincidentes: la disponibilidad de estaciones gráficas de bajo coste que ha posibilitado la utilización de las mismas de forma general por los diseñadores y, por ello, de lenguajes gráficos, y la tecnología de soporte al trabajo en grupo que ha hecho de ellas unas herramientas necesarias para el desarrollo de proyectos grandes.

Podemos definir una **herramienta** como un sistema de software cuya finalidad es la de ayudar a construir otros sistemas. Desde este punto de vista lo que permite es mejorar la capacidad del ingeniero de software en diversas fases del desarrollo.

Las herramientas requeridas a lo largo del desarrollo son muy dispares. Históricamente, las primeras que aparecen son editores para generar las descripciones de los sistemas en algún lenguaje, compiladores para generar código, depuradores para analizar las posibles fuentes de error, etc. Todas ellas dedicadas a soportar la fase de implementación. Recientemente, han surgido otras para soportar las fases iniciales del ciclo de vida.

En este contexto surge el concepto de entorno de programación o diseño y que en la literatura se conoce como sistema **CASE** (Ingeniería Software Ayudada por Ordenador). Más concretamente, los sistemas CASE solían limitarse al soporte de las primeras fases del desarrollo, relegándose la denominación **Entorno de Programación** a los que soportaban la fase de implementación. Hoy día, podemos hablar de **Entornos Integrados** (o CASE integrado) a los que soportan todas ellas.

La **integración** se refiere al grado en el que las herramientas están relacionadas entre sí. Si las herramientas no están integradas, es responsabilidad del ingeniero software elegir y utilizar una de ellas, obtener los resultados, interpretarlos, y, si fuese necesario, convertirlos en el formato adecuado para otras herramientas. Al menos, conseguir que el intercambio de datos sea posible traduciendo los formatos de forma automática. Para solventar este problema se debe disponer de mecanismos de integración a diferentes niveles que faciliten la relación entre herramientas (ver Figura 16).

Diferenciamos entre **niveles de integración**: visual, de datos, de control o de proceso. Adicionalmente a estos niveles de inte-

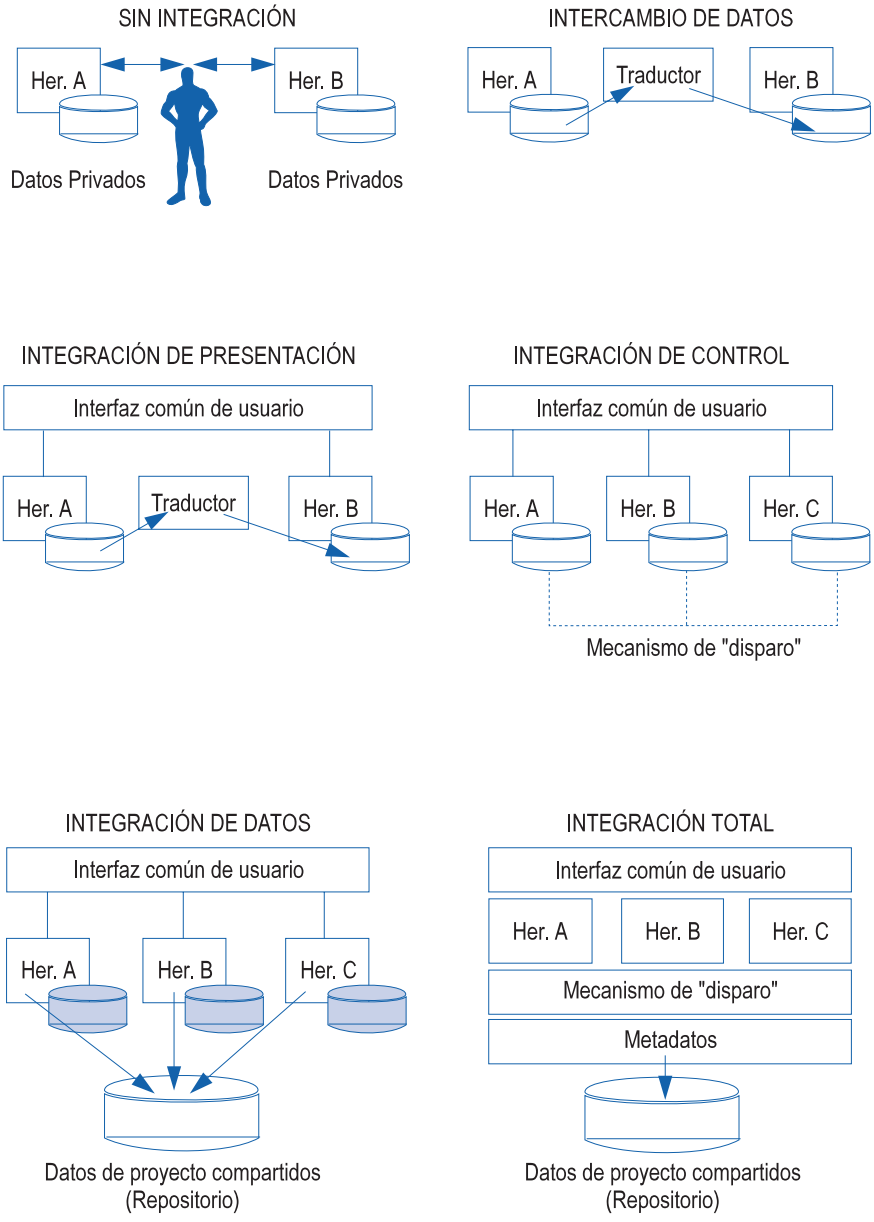


Figura 16 - COMPARACIÓN ENTRE NIVELES DE INTEGRACIÓN -

gración hablamos de **integración conceptual** cuando las herramientas están aisladas pero juegan un papel perfectamente definido dentro del soporte a un método concreto. No olvidemos que, generalmente, un sistema CASE soporta un conjunto de métodos concretos:

A) Integración visual o de presentación. La integración visual se refiere al hecho de que las herramientas se presentan al usuario con una interfaz única desde la que puede acceder a cualquiera de ellas. Típicamente se apoyan en entornos de ventanas normalizados («de facto») con una apariencia común (botones, barras de herramientas, formas de navegar por la información, etc.). Estos entornos de ventanas permiten gestionar la creación, movimiento, borrado, e intercambio de información entre ventanas.

A partir de estos entornos de ventanas se diseña la interfaz gráfica del entorno de desarrollo. Desde la interfaz gráfica se puede llamar a todas las herramientas del entorno. Existen herramientas software que facilitan la creación casi automática de estas interfaces gráficas.

Téngase en cuenta que por disponer de integración visual no tiene por qué existir ninguna relación entre las herramientas a las que se llame. La interfaz gráfica sólo conoce el nombre y algunos parámetros de configuración de la herramienta pero no su función ni si es posible usarla en ese estadio del desarrollo.

B) Integración de datos. Por integración de datos se entiende la capacidad de las herramientas para acceder a una estructura de datos común. Los datos comunes contienen la información asociada al sistema en desarrollo. Los datos se albergan en un elemento denominado

repositorio y que puede considerarse como una base de datos especializada; este elemento es básico para mantener y actualizar la información relativa al sistema en desarrollo.

Como ejemplo de este nivel de integración, un editor gráfico genera una descripción del sistema en desarrollo en un formato interno. Esta descripción puede utilizarla posteriormente un generador de documentos para obtener una representación en algún formato externo requerido. Cada herramienta puede tener también sus datos privados.

La Figura 17 resume la estructura de un sistema CASE genérico en el que hemos representado las diferentes herramientas compartiendo una estructura de información común sobre el sistema en desarrollo.

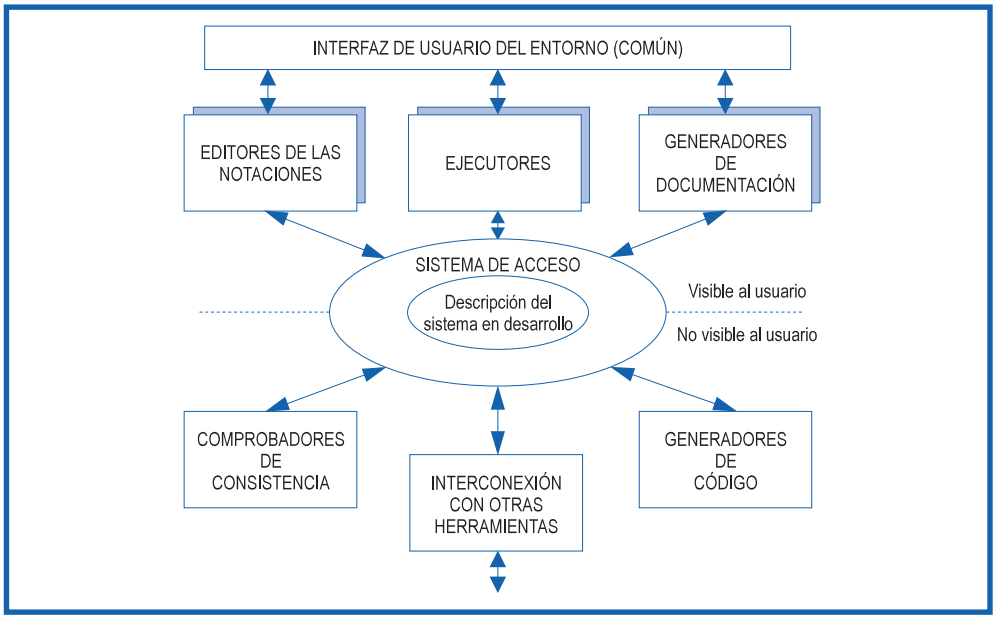


Figura 17 - ESTRUCTURA DE UN SISTEMA CASE GENÉRICO -

No obstante, las herramientas no pueden intercambiar información durante su ejecución; para ello, es necesario un nivel de integración superior.

C) **Integración de control**. La integración de control provee de mecanismos para que las herramientas intercambien información mediante mensajes o se activen y desactiven durante una sesión de desarrollo del sistema. Este nivel de integración es también la base para el soporte al trabajo cooperativo en grupo de un equipo de desarrollo.

Un ejemplo de la necesidad de este nivel de integración surge cuando diversos ejecutores de un prototipo heterogéneo necesitan coordinarse para su ejecución simultánea.

D) **Integración de proceso o total**. El nivel más elevado de integración entre herramientas la proporciona la integración de proceso. Con él, las herramientas conocen el modelo de desarrollo elegido y, en función de la fase en la que se encuentra, la actividad a realizar y los perfiles de las personas que intervienen, permite la utilización de determinadas herramientas y el acceso a datos comunes o el intercambio de información entre ellas impidiendo el acceso a otras y facilitando el «disparo» de las actividades del sistema. Esta información se contiene en lo que se denomina «meta-datos» (datos sobre los datos del sistema en desarrollo; ver la Figura 16).

Con estos conceptos básicos de integración es posible desarrollar diversos tipos de entornos muy diferentes entre sí. Desde el punto de vista de la ingeniería de sistemas de software, son piezas bastante complejas y costosas (en muchos casos con un coste superior al del ordenador en la que se ejecutan) pero que se están

haciendo imprescindibles. Nadie aborda hoy día el desarrollo de un sistema de software sin disponer de herramientas.

Desde el punto de vista del usuario, no sólo interesa la funcionalidad que posea en el momento de la adquisición sino hasta qué punto el entorno puede evolucionar. Los usuarios de entornos sofisticados son conscientes de que viven en un contexto muy cambiante en el que tanto la funcionalidad de los ordenadores como las propias plataformas de ejecución (sistemas operativos, bibliotecas de funciones u objetos, etc.) van a modificarse sustancialmente en los próximos años. Como consecuencia de ello se desea que el entorno sea **abierto** para poder incorporar nuevas herramientas a través de interfaces normalizadas.

La Figura 18 describe un modelo genérico de entorno CASE que está siendo utilizado como marco de referencia por diversas organizaciones de normalización para generar productos CASE en

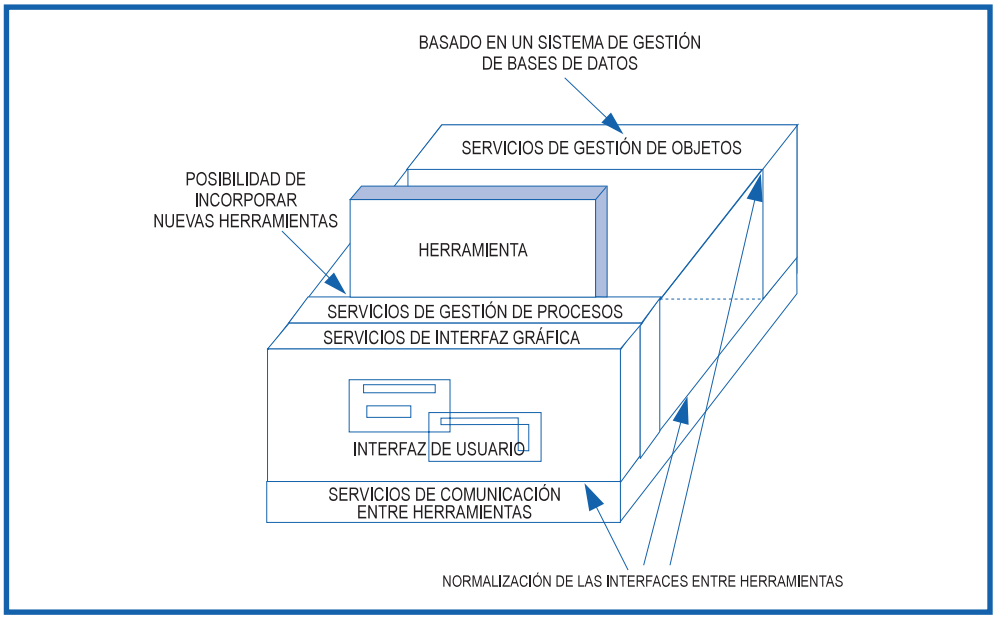


Figura 18 - MARCO DE REFERENCIA DE SISTEMAS CASE -

el mercado. Obsérvese que el entorno proporciona **servicios** para cubrir las necesidades de los diferentes niveles de integración.

Podemos ver también cómo las herramientas individuales pueden introducirse o extraerse (como tostadas en un tostador de pan; de ahí el nombre de «modelo de tostador» con el que algunas veces se le conoce) según las necesidades.

Estas herramientas pueden ser **verticales** cuando están orientadas a una actividad (o actividades) concreta del ciclo de vida (por ejemplo, un compilador) y **horizontales** cuando pueden soportar la gestión del proceso en todas sus fases (por ejemplo, un controlador de versiones).

No consideramos en esta clasificación de herramientas las que suelen servir para aspectos de gestión (como control de versiones y configuraciones, generadores de documentación, soporte al trabajo en grupo, planificación de proyectos etc.), actividades que veremos en el Capítulo 5 y que, sin embargo, forman parte sustancial de un entorno de desarrollo comercial.

3.3.5. Directrices de aplicación industrial

El último de los componentes tecnológicos mencionados anteriormente que cada vez está tomando más importancia es el denominado **directrices de aplicación industrial**. Este componente aporta al desarrollo de un sistema de software el conocimiento concreto sobre el dominio de aplicación. Con él es posible emplear soluciones, reutilizar componentes, y aprovechar la experiencia que, sobre ese dominio de aplicación, ha acumulado la organización y los individuos implicados en su desarrollo.

Todos los componentes anteriores están basados en el uso de elementos relativamente genéricos; pueden emplearse en el diseño

de muchos tipos distintos de sistemas de software. Lo que ahora buscamos son soluciones particulares en un dominio concreto.

Desde esta óptica, es responsabilidad de los desarrolladores, a partir del conocimiento del problema a resolver, emplear de la mejor manera posible los componentes de la tecnología de software seleccionada adaptándolos a las necesidades de nuestro problema concreto.

La experiencia que una empresa posee en el desarrollo de sistemas se consolida en un conocimiento estructurado en directrices o guías de aplicación industrial. Este conocimiento y experiencia se posibilita aún más cuando se integra en los métodos, notaciones, herramientas y formas de razonar sobre el sistema, es decir, complementa y da un sentido práctico al resto de los componentes de una tecnología de software. Esta es una de las razones que justifican el tiempo necesario para que una tecnología de software madure.

En resumen, este componente está ligado a dos aspectos básicos: aprovechar componentes previamente diseñados y útiles para una aplicación concreta (ligado a la reusabilidad) y aprender de la experiencia anterior. Seguidamente, veremos cómo se aplican.

3.3.5.1. Componentes reutilizables

Cada dominio de aplicación posee una serie de características que fuerzan el uso de notaciones o métodos concretos (las herramientas deben soportar ambas).

Los componentes reutilizables son módulos genéricos que pueden componerse para construir un sistema. No ha sido hasta muy recientemente cuando estos componentes han empezado a ser útiles de forma real para el diseño de sistemas de software complejos aunque en la fase de implementación todos hemos empleado las bibliotecas

de funciones (por ejemplo, matemáticas o de entrada/salida) llamadas desde un lenguaje determinado (por ejemplo, en FORTRAN).

Un diseño basado en componentes de un catálogo conlleva, además, un problema de confianza en la corrección de los módulos a utilizar; de ellos dependerá la corrección del sistema final.

Es interesante comentar en este caso la aparición en la ingeniería de sistemas de software de un fenómeno bien conocido en la ingeniería de sistemas: el control de calidad de las piezas es básico para obtener un producto de calidad. Cada vez será más importante disponer de bibliotecas de calidad porque de ellas se derivará la calidad del producto final.

3.3.5.2. Consolidación del conocimiento previo

Aprender de la experiencia anterior implica reconocer aspectos comunes en el desarrollo continuo de sistemas cuya evaluación, problemas y soluciones pueden aplicarse posteriormente.

Consolidar el conocimiento previo no sólo implica utilizar un catálogo de componentes para el diseño de un nuevo sistema. La mayor parte de las empresas necesita mantener sistemas de software ya anticuados que es necesario modificar sustancialmente. Si únicamente se dispone de la documentación ligada al código fuente, la única manera de reconstruir el producto es mediante la extracción del diseño a partir del código; este es el concepto de **Ingeniería Inversa** en ingeniería de sistemas de software y para la que empiezan a aparecer métodos y herramientas específicos.

3.4. Ejemplos de tecnologías de software

En la presente sección pasaremos revista a dos grandes grupos de tecnologías de software analizando las principales características

de sus componentes en el momento actual y señalando las tendencias observadas. Las tecnologías identificadas son:

a) Tecnologías de desarrollo estructurado.
b) Tecnologías orientadas a objetos.

Se han seleccionado estas dos porque representan dos estadios distintos de la evolución tecnológica en la ingeniería de sistemas de software. Deliberadamente, no entraremos en profundidad en ninguna de ellas. El lector interesado deberá remitirse a la información contenida en la bibliografía de cada una de ellas. En el Capítulo 4 podrá verse la aplicación de alguna de ellas para el desarrollo de sistemas de tiempo real.

Las limitaciones de espacio de esta monografía y su carácter no especializado nos impiden abordar un tercer grupo como es el de los métodos formales. Al lector interesado le recomendamos [11] para hacerse una idea del estado actual.

3.4.1. Tecnologías de desarrollo estructurado

Las tecnologías de desarrollo estructurado son las más convencionales de las empleadas hoy día. Han surgido de la evolución de las ideas de programación estructurada (hace más de veinticinco años) hacia las fases iniciales del ciclo de vida.

En su formulación actual, las notaciones empleadas en las primeras fases del ciclo de vida (especificación de requisitos de usuario y sistema) suelen estar constituidas por lenguajes gráficos que permiten: identificar el sistema y el entorno; representar el flujo de información entre los elementos; y, describir los datos y las actividades del sistema [12].

La idea base de esta tecnología es que es posible estructurar el modelo de un sistema de software en base a funciones que procesan información que reciben de otras funciones (o del exterior) y dirigen la información

procesada a otros módulos funcionales (o al exterior). El enfoque seguido, por tanto, es el de pensar en las funciones del sistema necesarias (extraídas de los requisitos del sistema) y luego en los datos que requieren.

Entre las más utilizadas para análisis y especificación de requisitos se encuentra SA/RT (Análisis Estructurado con extensiones para tiempo real) [13]. Surgió como un lenguaje gráfico capaz de representar las actividades que deberá realizar el sistema, los intercambios de información entre ellos, etc. La descripción del comportamiento se realiza mediante diagramas de transición de estados.

Existen otras notaciones basadas en conceptos muy similares y el utilizar una u otra es más bien un problema de gusto. Las diferencias entre ellos provienen más de la forma de usarla que de la potencia expresiva del lenguaje.

Como evolución de las técnicas de análisis estructurado, en la fase de diseño se han utilizado variantes de SA/RT: SD/RT (Diseño Estructurado con extensiones para Tiempo Real). Al igual que SA/RT consta de un lenguaje gráfico no ejecutable e incorporan conceptos tales como: tarea, procesador, colas de mensajes, mecanismos de sincronización entre tareas, etc. que son conceptos necesarios en la fase de diseño.

En una línea diferente y para evitar los problemas de la explosión de estados se definieron por Harel [14] los «statecharts» (variante de los diagramas de estado). Con ellos, se lograba compactar el espacio de estados que resultaba al describir sistemas de gran complejidad al permitir jerarquización de estados y descomposición en componentes. En base a ellos se ha desarrollado una tecnología estructurada adaptada a sistemas de control denominada Statemate [15].

Para la fase de análisis y especificación de requisitos, las herramientas están asociadas a la construcción de modelos del sistema (modelos lógicos con diagramas de estado asociados). Estas herramientas no son genéricas sino que soportan métodos concretos. Suelen constar de:

A) Editores gráfico-textuales de la notación asociada a un método (tanto para describir las funciones como para describir el comportamiento mediante diagramas de estado).

B) Comprobadores de consistencia en la información relativa a refinamientos del modelo (nombres, tipos, uso, etc. de los elementos definidos en los diagramas).

C) Sistema de gestión de la información almacenada (en ocasiones basada en bases de datos relacionales u orientadas a objetos para gestionar el acceso a la información).

D) Generadores de prototipos (normalmente de interfaz gráfica) con objeto de evaluar los modelos lógicos o de diseño.

En las fases de diseño del sistema se dispone del mismo tipo de herramientas aunque en este caso se suele disponer también de: analizadores temporales y estimadores de tiempos de ejecución, generadores de código (más o menos completos) o facilidades para la utilización de componentes genéricos contenidos en bibliotecas menos comunes pero cada vez más conocidas son herramientas como las de **animación gráfica de modelos**. Estas herramientas aparecen como extensión de las que permiten editar y validar modelos de especificación y diseño estructurado de sistemas de software.

Finalmente, las herramientas que soportan la fase de implementación son las más conocidas dado que han estado en su mayor parte presentes desde los comienzos de la programación: editores (conociendo la sintaxis del lenguaje en algunos casos), compiladores e intérpretes, generadores/optimizadores de código, ejecutores de casos de prueba, depuradores simbólicos, etc.

En resumen, la Figura 19 representa esquemáticamente los componentes de la tecnología de software estructurada.

Figura 19 - COMPONENTES
DE UNA TECNOLOGÍA DE DESARROLLO ESTRUCTURADO -

Aunque este tipo de tecnologías de software aún se utilizan y sufren rejuvenecimientos periódicos, se está produciendo un desplazamiento de los usuarios hacia tecnologías orientadas a objetos que abordaremos seguidamente. Únicamente en el caso de sistemas de tiempo real existe una inercia a su abandono puesto que aún no se dispone de tecnologías orientadas a objetos validadas industrialmente en ese dominio.

3.4.2. Tecnologías orientadas a objetos

Las tecnologías de desarrollo estructurado han demostrado sus limitaciones a la hora de organizar y facilitar la evolución de sistemas de software complejos. La descomposición en funciones hace dificil al diseñador mantener la relación con los objetos del mundo real sobre los que se modifican generalmente los requisitos del usuario.

Los métodos de descomposición orientada a objetos constituyen la tendencia más influyente observada en la ingeniería de sistemas de software en los últimos años. Con ellos nos referimos a un conjunto de métodos (aún en fase de desarrollo o evolución) que permiten al analista y diseñador concebir su sistema identificando clases de objetos, operaciones permitidas y relaciones entre ellos como base para la estructura del sistema a diseñar.

En ellas, un **objeto** es un conjunto de datos y funciones de manipulación de los mismos encapsulados en una unidad que es posible tratar como un todo (crear, copiar, destruir, etc.). Un objeto posee unas operaciones visibles a otros objetos aunque éstos no conocen cómo están implementadas. El diseñador reconoce inicialmente **clases de objetos** de las que se derivan los objetos concretos que utilizará en el diseño.

Un objeto puede construirse jerárquicamente empleando, a su vez, a otros objetos más simples. Una **clase** implica una generalización del concepto de objeto (identificando similitudes entre objetos similares) y constituye la base a partir de las cuales se construye el sistema.

Existen varias tecnologías orientadas a objetos que, aunque similares en su potencia expresiva, ofrecen algunas diferencias que las hacen más adecuadas para algún tipo concreto de sistemas. Podemos mencionar como una de las más representativas a **OMT**.

OMT está soportada por muchas herramientas CASE comerciales. Corresponde a una notación gráfica que permite representar las clases de objetos, sus relaciones y la creación de ejemplares de los mismos. Aunque básicamente empleada para la fase de análisis de requisitos del sistema puede también emplearse para las primeras fases del diseño.

La descripción del comportamiento se realiza generalmente asociando a los objetos diagramas de transición de estados similares a los empleados en las tecnologías de software estructuradas (con los

mismos problemas de la explosión de estados). En Booch puede verse una idea general de su tecnología orientada a objetos [16].

Los métodos de diseño orientados a objetos suelen facilitar el desarrollo de una implementación en un lenguaje de programación orientado a objetos (C++, Ada95 o Eiffel). No obstante, la elección del lenguaje de implementación no es realmente importante y esta elección está condicionada por muchas otras razones. Justo es reconocer, sin embargo, que ha sido la **Programación Orientada a Objetos** la que ha impulsado también la difusión de estas técnicas.

Las herramientas que acompañan a las tecnologías orientadas a objetos y disponibles en sistemas CASE comerciales no se diferencian en esencia de las que aparecen en las tecnologías estructuradas. El único aspecto destacable es la proliferación de catálogos de clases para aplicaciones determinadas y los mecanismos de recuperación y personalización asociados.

La Figura 20 representa esquemáticamente los componentes típicos de una tecnología de software orientada a objetos.

3.5. Resumen

El análisis efectuado en este Capítulo por cada uno de los componentes de una tecnología de software nos ha permitido obtener una idea global del estado en el que se encuentra cada una de ellas. Estos componentes, obviamente, no son independientes. La situación puede describirse de la siguiente manera:

A) Las notaciones empleadas comúnmente (lenguajes de especificación, diseño, codificación, prueba, etc.) no permiten describir el sistema de manera progresiva a lo largo del ciclo de vida. En la práctica, es necesario combinar varias de

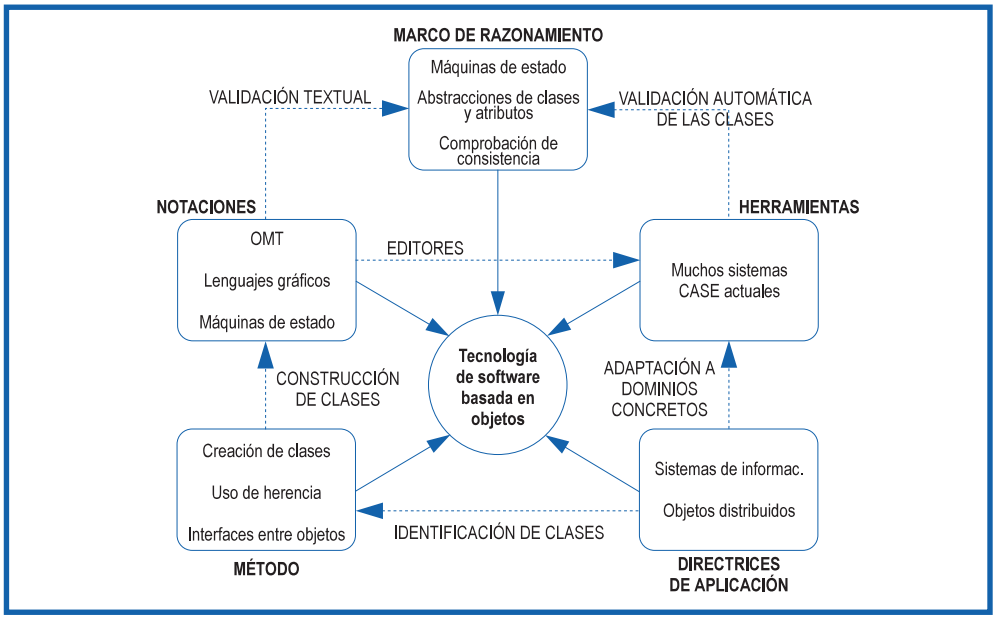


Figura 20 - COMPONENTES DE UNA TECNOLOGÍA ORIENTADA A OBJETOS -

ellas, con el inconveniente de tener que trasladar las decisiones y conceptos de una a otra.

B) Toda tecnología requiere que exista una forma de refinar el sistema y llegar desde la definición de requisitos hasta la implementación. Existen muchos métodos posibles con variada penetración en el mercado. La elección del más adecuado depende de múltiples factores no exclusivamente técnicos.

C) En el proceso de refinamiento con un método concreto es necesario comprobar que las decisiones tomadas no vulneran las propiedades deseadas del sistema. La capacidad de razonar sobre el sistema (a través de las descripciones del mismo) es la base del modelo de razonamiento empleado. Solo los métodos formales pueden realmente asegurar un nivel alto de confianza.

D) Las herramientas no son más que un vehículo para soportar la descripción y el razonamiento sobre un sistema. De poco sirve que sean muy agradables de usar, robustas o eficientes si soportan notaciones poco claras o desligadas entre sí, marcos de razonamiento pobres o muy difíciles de usar.

E) Su principal rol es, justamente, posibilitar que pueda aprovecharse toda la potencialidad de las notaciones y la capacidad de razonamiento sobre un sistema. Si, además, las herramientas están integradas se puede soportar el desarrollo de un sistema complejo por un equipo de trabajo. Es interesante destacar que las mismas notaciones y herramientas pueden usarse de muchas maneras ... y unas son más eficientes que otras.

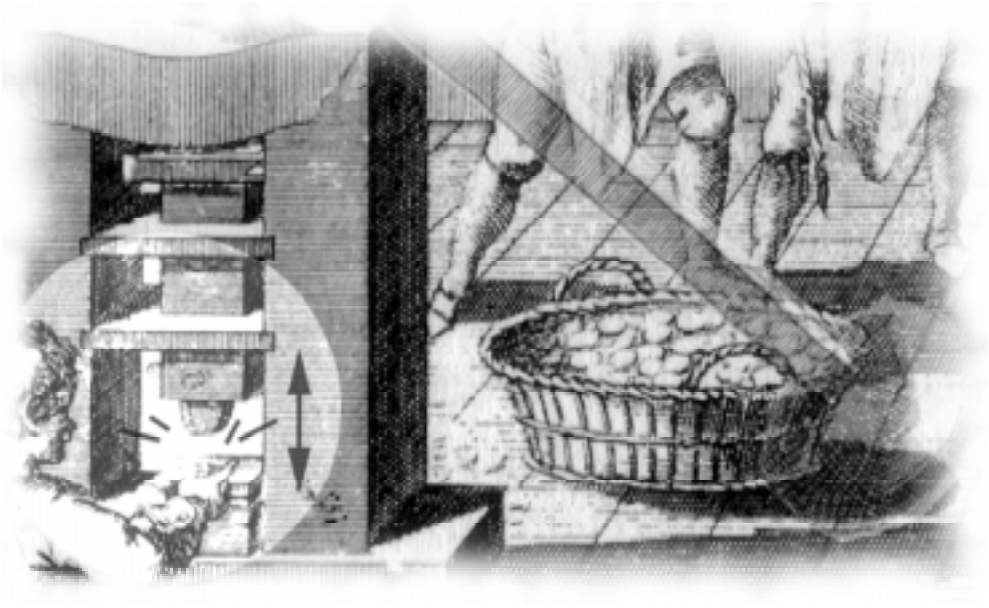
F) Finalmente, como parte de la tecnología de software hemos desgajado el conocimiento relativo a un dominio de aplicación. Esto implica que, en cierta forma, los problemas y los esquemas de soluciones ligados a un dominio concreto son independientes de la notación utilizada (no evidentemente su formulación, que depende de los otros componentes).

Generalmente, el desarrollo de un método no se hace a espaldas de los demás componentes. En la práctica es el desarrollo de sistemas reales el que alimenta los propios heurísticos del método y las mejores notaciones y herramientas. Esta realimentación es la base del progreso de la tecnología de software.

Buscar una solución genérica asociada a un problema tipo, expresarla en las notaciones empleadas, comprobar que satisface las propiedades deseadas, emplear las herramientas para ello, y utilizarla de forma consistente con un método de desarrollo que guíe el proceso de refinamiento de la descripción del sistema desde los requisitos hasta la implementación es la manera en la que los componentes de la tecnología interaccionan.

4

Tecnologías para desarrollo de sistemas de tiempo real



4.1. Introducción

Aunque a lo largo del Capítulo 3 hemos presentado diversas tecnologías aplicables a diferentes tipos de sistemas de software, queremos en este Capítulo dedicar especial atención a los sistemas de tiempo real. Ello nos servirá también para entender la relación entre los diferentes componentes de una tecnología de software y así poder profundizar en alguna concreta.

Los **Sistemas de Tiempo Real** están teniendo una importancia creciente utilizándose en muchos campos de la actividad humana y constituyendo uno de los elementos básicos para el control de actividades críticas [17]. Esta situación ha hecho que muchos de los desarrollos de nuevas tecnologías de software tengan como objetivo facilitar el desarrollo de Sistemas de Tiempo Real.

Revisaremos en este Capítulo la problemática general de su diseño, las características generales de las técnicas que se han ideado para su desarrollo y la posible evolución de las mismas. Siendo conscientes de la necesaria brevedad del Capítulo, animamos al lector interesado a profundizar en estas técnicas a partir de las referencias contenidas en el presente Capítulo.

4.1.1. Definiciones básicas

Un Sistema de Tiempo Real (STR) puede definirse como aquél que debe completar sus actividades en plazos de tiempo predeter-

minados. Como consecuencia, su ejecución debe satisfacer restricciones temporales cuyo incumplimiento supone el funcionamiento incorrecto del sistema.

En otras palabras, cuando se recibe algún evento (mensaje o dato) desde el exterior del sistema (por ejemplo, desde una línea de comunicación) o se lee algún valor de un dispositivo externo (por ejemplo, un sensor de un sistema físico), el STR debe reaccionar ejecutando algunas acciones en un intervalo de tiempo predefinido. La corrección, por tanto, de un STR no depende sólo del valor de los datos de salida sino también del instante en el que se producen.

Si estas restricciones no se satisfacen, sus resultados empiezan a perder validez para el usuario o se llega incluso a consecuencias catastróficas en el sistema sobre el que se pretende actuar.

Un campo clásico de aplicación de los Sistemas de Tiempo Real es el de los sistemas de control. Su importancia justifica que les dediquemos especial atención en este Capítulo. No obstante, muchos sistemas de tiempo real no pertenecen a los denominados sistemas de control. Hay otro gran grupo de STR cuya misión es monitorizar un sistema físico externo (por ejemplo, capturando datos de él como ocurre en los sistemas de adquisición de datos). En muchos casos, nos encontramos con sistemas de tiempo real híbridos en los que se monitoriza y controla un sistema físico externo.

El objetivo de un **Sistema de Control** es hacer que la evolución dinámica de un sistema físico externo (sistema controlado) evolucione según nuestros deseos. El sistema de control se considera de tiempo real en el sentido de que la actuación sobre el sistema controlado debe efectuarse dentro de restricciones temporales estrictas.

La actuación del STR sobre el sistema controlado se efectúa leyendo datos de algunas magnitudes físicas del mismo y generando a partir de ellas señales de actuación que modifican el comportamiento

del sistema controlado. En función de estas señales, el sistema controlado evoluciona con lo que las nuevas lecturas que haga el STR serán diferentes haciendo que el sistema de control actúe de nuevo sobre el sistema controlado repitiéndose el ciclo.

El ciclo de lectura de señales, cálculo y actuación se repite cíclicamente. El comportamiento del sistema controlado evolucionará según su propia dinámica y de la actuación del STR. Es importante tener en cuenta que no basta repetir el ciclo mencionado: debe repetirse en plazos dependientes de la dinámica del sistema; de otra forma, no se podría garantizar que el sistema evolucione según nuestros deseos.

La Figura 21 representa la forma genérica de un Sistema de Tiempo Real en el que podemos ver cómo las salidas (en función del tiempo) son funciones de las entradas (eventos externos recibidos en un instante anterior) y salidas (generadas también en otro instante anterior). En la figura se ha representado también un caso concreto de STR: un Sistema de Control. Obsérvese la existencia del sistema controlado que es el motivo de la existencia del sistema de control de tiempo real.

En la Figura 21 podemos ver cómo el sistema de control obtiene del sistema controlado unos datos de monitorización y algunas variables empleadas para el control y, posiblemente, información del operador. Con ello, elabora las señales de actuación y, eventualmente, otros datos de salida para poder representar la evolución dinámica.

Un ejemplo típico de un sistema de control es un controlador remoto de la temperatura de un horno de fundición de aceros especiales que actúa en función de la temperatura del mismo. El control de la temperatura debe ser muy estricto no sólo en los valores permitidos sino en los tiempos en los que esta temperatura debe mantenerse dado que, en caso contrario, se alterarían las propiedades del acero resultante.

El sistema de control (controlador) puede leer los sensores de temperatura del horno cada cierto tiempo. éste, en función de los valores

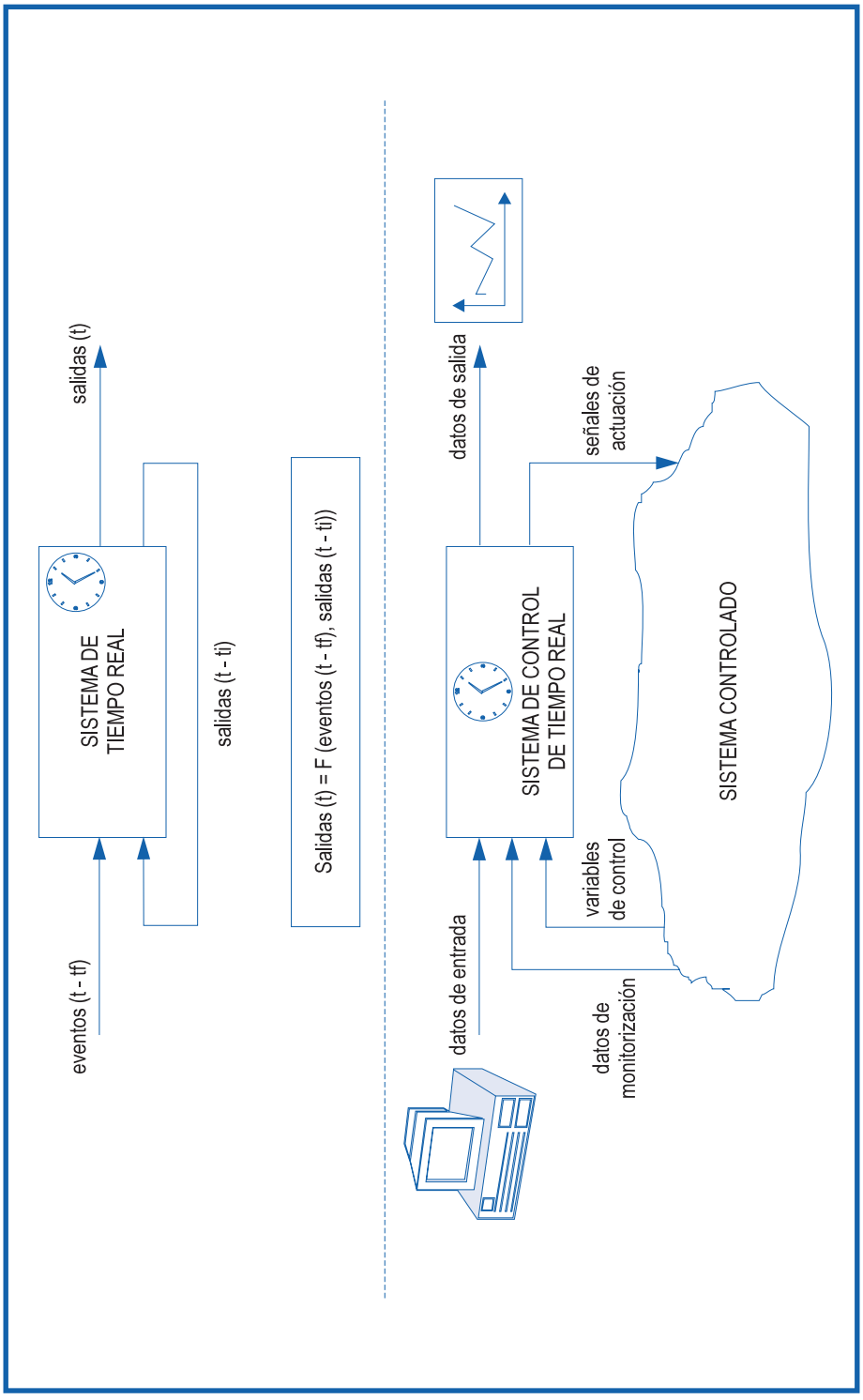


Figura 21 - ESTRUCTURA DE UN SISTEMA DE TIEMPO REAL -

recibidos, del instante de tiempo considerado y de la situación del horno decide incrementar o decrementar la temperatura actuando sobre el sistema de calentamiento.

El sistema es de tiempo real dado que la actuación sobre el sistema de calentamiento es función de los valores recibidos dentro de un margen temporal estricto.

Si no se atiende a los datos suministrados por los sensores o se atienden muy lentamente, puede ocurrir que la temperatura del horno no mantenga la curva de temperatura predefinida o peor aún, que ésta ascienda por encima del margen de seguridad del horno y se produzcan fisuras en las paredes del mismo (consecuencias indeseables sobre el sistema controlado).

En el caso de que el incumplimiento de una restricción temporal lleve a un fallo global del sistema (con consecuencias que pueden llegar a ser muy graves) estamos ante un caso de **tiempo real crítico**. En otros casos, en los que el efecto es simplemente una degradación de las prestaciones se denomina de **tiempo real acrítico**.

Continuando con el ejemplo anterior, si el sistema de control no mantiene la temperatura del horno dentro de los márgenes preestablecidos, puede ocurrir que haya una degradación paulatina de las propiedades del acero o que esta degradación sea muy brusca; igualmente, el margen para sobrepasar la temperatura de peligro puede ser muy estricto o no. En un caso, estamos en presencia de un sistema de tiempo real crítico y en otro caso acrítico.

Un ejemplo similar al anterior sería un controlador de temperatura de un edificio. También aquí el STR lee valores de sensores de temperatura. La diferencia es que en este caso el sistema no será crítico puesto que variaciones no controladas no llevan a una inutilización del edificio. Por otro lado, el margen de actuación es de minutos.

Imaginemos, sin embargo, el caso de un brazo robotizado empleado para asir un objeto. En este caso, las señales de actuación sobre el robot deben aparecer en el momento adecuado para que, dada la inercia del movimiento del mismo (no puede detenerse en cero segundos), se detenga sobre un objeto sin romperlo. Si la señal de actuación se produce más tarde podría chocar violentamente sobre el objeto inutilizándole.

Muchos STR son **empotrados** (no pueden verse aisladamente del sistema sobre el que actúan). Por ejemplo, el sistema de control de temperatura del horno mencionado anteriormente puede ser empotrado y no visible directamente a ningún usuario externo (éste sólo observará las consecuencias).

4.1.2. Restricciones temporales

De acuerdo con lo anterior, un Sistema de Tiempo Real suele tener un conjunto de **restricciones temporales** marcadas por la dinámica del sistema físico externo. Estas restricciones se asocian a las especificaciones funcionales cuyo cumplimiento es esencial para otorgar validez a la ejecución del sistema. Las restricciones temporales pueden ser de tres tipos diferentes:

1) **Planificación de los instantes de activación de las actividades del sistema**.

   Es típico encontrarse con requisitos del tipo de que una actividad debe iniciarse a una hora dada (por ejemplo, el lanzamiento automático de una señal de aviso a las 9:45) o que se ejecute con una determinada periodicidad (por ejemplo, el muestreo de una señal de entrada cada 10 segundos). Para ello se requiere que las actividades del sistema sean controladas por un reloj con la precisión adecuada.

2) **Plazos de tiempo (máximos o mínimos) en los que debe completarse una actividad**.

Como ejemplo de este tipo, podemos indicar el tiempo máximo o mínimo de procesamiento de una actividad (por ejemplo, el tiempo de consulta a una base de datos debe ser inferior a X segundos) para que los datos consultados tengan validez.

Aunque estos plazos puedan derivarse de un requisito de usuario, durante el desarrollo pueden aparecer otros muchos asociados a las actividades del sistema.

3) **Intervalos de tiempo entre eventos del sistema**.

Como ejemplo, nos podemos encontrar con restricciones del tipo de que el tiempo máximo o mínimo entre dos eventos determinados (por ejemplo, dos disparos consecutivos de un sistema de armas) no debe ser superior a X segundos.

Este tipo de restricciones temporales implica que todos los eventos del sistema o actividades internas deben tener asociado un valor temporal, **sello temporal**, con el valor del instante de tiempo en el que se han producido. El código del STR utilizará estos valores junto con el valor del reloj para tomar las decisiones apropiadas.

Tampoco podemos pensar que la mera formulación de un requisito temporal (por muy clara que ésta sea) implica que sea realizable. Si un usuario nos exige que un mensaje debe ir a Marte y volver en menos de 3 ms, el requisito no es realizable en ninguna tecnología (aceptando las limitaciones de la Fisica clásica).

Es interesante diferenciar bien lo que suponen las restricciones temporales de otros requisitos de prestaciones. Si lo que buscamos es un sistema de software más rápido (esto es, que realice sus actividades en menor tiempo) tendremos quizás un sistema con mejores presta-

ciones y puede que con ello, sea más aceptado por el usuario final. En un STR es necesario asegurar el cumplimiento de todos los plazos siempre y no «casi siempre». Asegurar que estos plazos se cumplen puede llevar a sacrificar prestaciones globales.

En algunos sistemas de software se suelen determinar las prestaciones requeridas en base a estadísticas del tiempo de respuesta de determinadas actividades (por término medio, las actividades requerirán un determinado tiempo de ejecución por debajo de un cierto umbral). Sin embargo, en un STR lo importante es que se cumplan las restricciones temporales para todas las actividades y no sólo determinados valores estadísticos.

Como ejemplo, un programa de cálculo numérico que para la resolución de sistemas de ecuaciones lineales necesite realizar inversiones de matrices de 1.000 x 1.000 números complejos, puede que requiera los menores tiempos de ejecución posibles para que pueda abrirse paso en el mercado; para ello, puede ser necesario disponer de algoritmos especiales para tratamiento de matrices u obligar al usuario a ejecutarlo sobre un supercomputador. Pero la validez de los datos de salida (la solución del sistema) no depende del instante en el que se producen ni por ello pierden validez.

4.1.3. Evolución dinámica

Para controlar o monitorizar un sistema externo, el diseñador del STR deberá construir un **modelo** del sistema a controlar en el que incluya la información necesaria para poder controlar la evolución dinámica del mismo. A esta información se le denomina **estado**. En la Figura 22 se puede ver esta idea en la que un mismo evento puede dar lugar a dos estados distintos en función de alguna condición (relacionada con algún valor de una variable o del reloj). El cambio de estado puede provocar algún otro evento o acción sobre otra parte del sistema. En el nuevo estado, el sistema reaccionará a otros eventos.

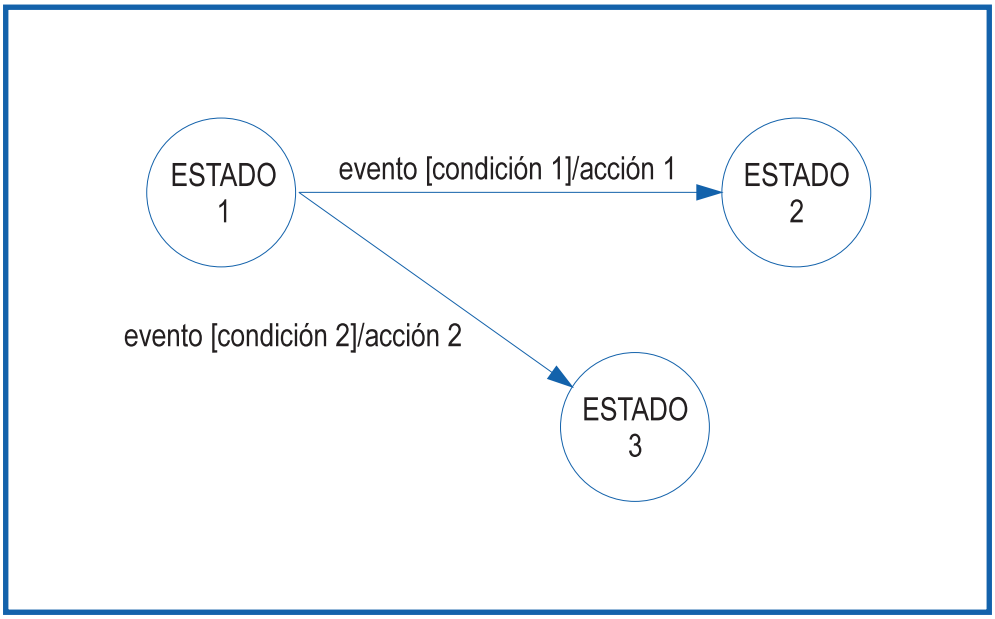


Figura 22 - TRANSICIÓN ENTRE ESTADOS EN UN SISTEMA DE TIEMPO REAL -

Las actividades que realiza un STR son función del estado del mismo. Un STR se encuentra en un estado definido por los valores de sus variables internas, actividades en curso, valor del reloj, etc.

El STR pasa,por tanto, por una serie de estados siendo los eventos generados o recibidos, internos o externos, los responsables de las transiciones entre estados. Como ejemplo, la llegada de un mensaje por una línea de entrada es un evento externo mientras que la finalización de una temporización es un evento interno al sistema.

Característico de los STR es que algunas de las transiciones entre estados estén motivados por eventos temporales (finalización de un temporizador, llegada del reloj del sistema a una hora programada, etc.). La «máquina» que maneja los estados del sistema constituye el corazón del control de las actividades del mismo.

4.2. Aspectos críticos en el desarrollo de sistemas de tiempo real

El desarrollo de sistemas de tiempo real podría hacerse como cualquier otro sistema de software empleando cualquier modelo de ciclo de vida. No obstante, la importancia que los aspectos temporales tienen en este tipo de sistemas hace que la atención del diseñador deba concentrarse en algunos aspectos desapercibidos en sistemas de información [10].

Los problemas más comunes al aplicar las técnicas de desarrollo convencionales vistas en el Capítulo 3 son:

A) **Descripción de los requisitos temporales**. La mayor parte de las técnicas de desarrollo estructurado u orientadas a objetos se han ideado para sistemas de información para los que la expresión de los requisitos temporales no es básica. En ellos, el aspecto fundamental que debe soportar es la transformación de los datos de entrada.

El objetivo de una tecnología adaptada al desarrollo de STR en este aspecto es poder describir los requisitos temporales y asociarlos a las especificaciones funcionales desde las primeras fases del ciclo de vida.

B) **Descripción del control**. Como hemos indicado, el control sobre el sistema físico externo se realiza en función del estado. Será necesario, por tanto, describir éstos mediante diagramas de estado que permitan conocer los estados y las transiciones entre ellos.

Desde esa perspectiva, el problema está asociado a la necesidad de manejar un número elevado de estados que hace imposible en la práctica el empleo de diagramas de estado planos (los convencionales, en un único nivel) que son los que soportan la mayor parte de los métodos de desarrollo disponibles.

A este problema se le conoce generalmente como de **explosión de estados** y ha motivado la aparición de técnicas de manejo de diagramas de estado jerárquicos (en varios niveles).

C) La planificación de los recursos del sistema. El uso de la capacidad de procesamiento disponible (una o varias CPUs) debe repartirse entre todas las actividades que intervienen. Esta distribución deberá asegurar el cumplimiento de las restricciones temporales establecidas.

Este problema conlleva la necesidad de determinar cuándo una actividad debe comenzar o reanudar su ejecución y cuál es el orden entre todas ellas. Si es posible encontrar una ordenación que respete los requisitos temporales, decimos que el sistema es **planificable**.

D) La prueba del sistema. Probar que un STR satisface sus requisitos temporales es algo que se hace clásicamente actuando sobre el código. Es en ese momento en el que podemos asegurar que sobre una plataforma de ejecución concreta (hardware y software), la ejecución y planificación de las actividades se realiza cumpliendo los requisitos temporales establecidos en la especificación.

Desgraciadamente, las técnicas de prueba habituales obligan a incluir código adicional al sistema que se va a probar afectando a los tiempos de ejecución reales y, por tanto, a la validación del sistema. Además, las tecnologías de software convencionales no permiten trasladar esta prueba a las fases iniciales.

E) Adecuación de la infraestructura de ejecución. La ejecución de cualquier sistema de software requiere de una plataforma de ejecución (recuérdese el Capítulo 1). Para

la ejecución de un STR no son válidas las habitualmente disponibles.

Centrando la atención en los sistemas operativos (S.O.) requeriremos que los mecanismos de reparto de la CPU entre las actividades (a este nivel tareas o procesos) y el manejo del reloj del sistema sea adecuado para un STR. Esto ha hecho necesario el diseño de sistemas operativos de tiempo real muy diferentes de los convencionales de tiempo compartido.

Como ejemplo, un algoritmo de planificación de procesos en la CPU que reparta la CPU asignando a cada proceso un intervalo (cuanto de tiempo) y seleccionando a uno de ellos en función de prioridades dinámicas (variables en función del tiempo de ejecución en la CPU acumulado por cada proceso) no permite garantizar la satisfacción de los requisitos temporales.

Los aspectos mencionados no están aislados. La Figura 23 resume los problemas encontrados al utilizar el modelo de ciclo de vida en cascada y las relaciones entre ellos cuando se trata de diseñar un STR.

El manejo de los requisitos temporales a lo largo del desarrollo implica ser capaces de describirlos de manera no ambigua, poder razonar sobre la consistencia de los mismos, y finalmente disponer de mecanismos que permitan definir los recursos necesarios para su implementación en una plataforma hardware o software predefinida. Ello dependerá de las tecnologías de software que podamos utilizar y que seguidamente analizamos brevemente.

4.3. Tecnologías de software para sistemas de tiempo real

Para poder cubrir los aspectos críticos que hemos mencionado, se han ideado o adaptado diversas tecnologías de software espe-

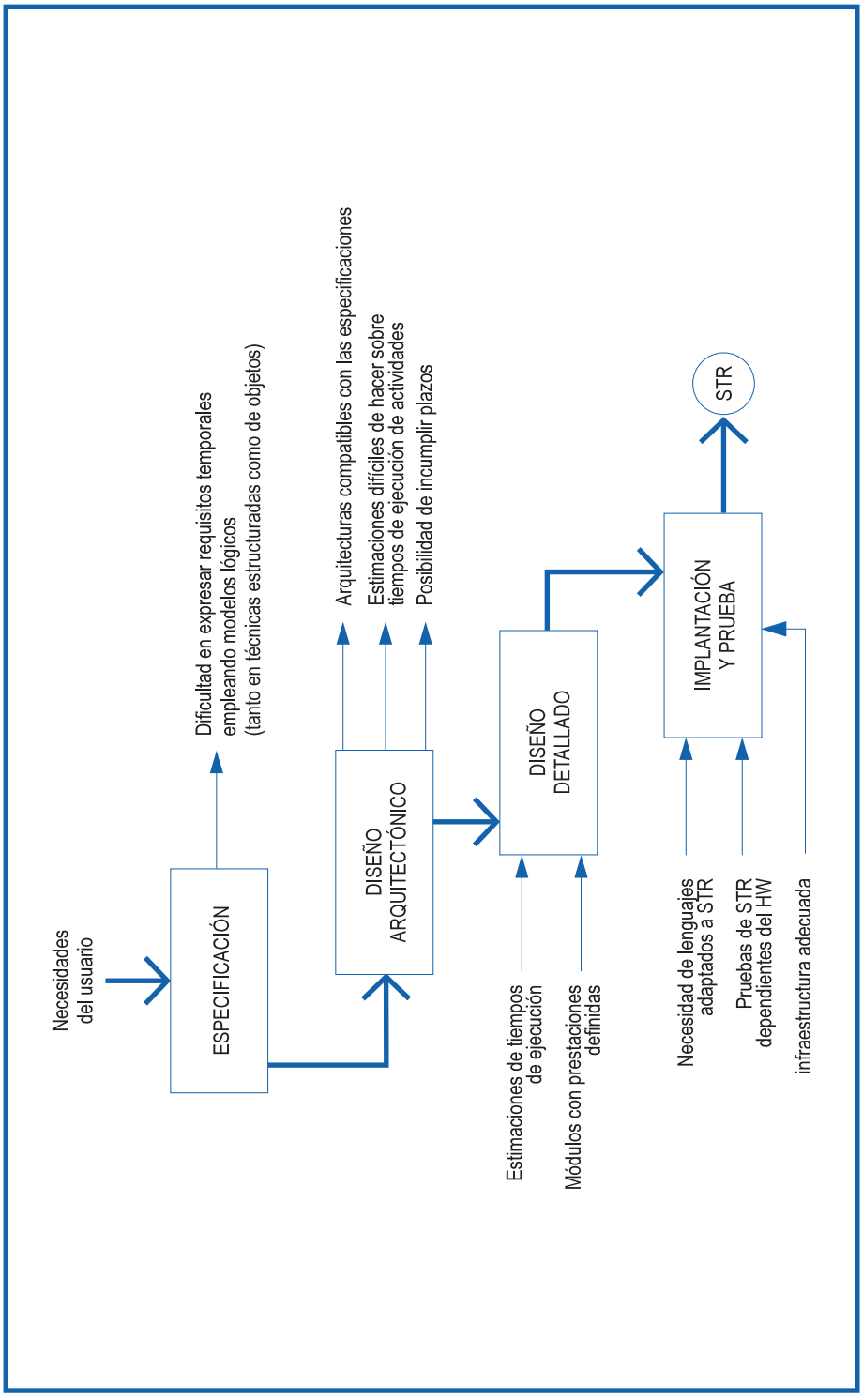


Figura 23 - LOS SISTEMAS DE TIEMPO REAL EN EL CICLO DE VIDA CONVENCIONAL -

cialmente adecuadas para el desarrollo de sistemas de tiempo real. Existen muchas tecnologías de software para STR y no es posible presentarlas todas en este Capítulo. Reduciremos la panorámica a dos de ellas: SA/RT y Statemate [15] cuya presentación se hará a través de las diferentes componentes tecnológicas. Comenzaremos por la componente del método por ser la más importante y la que, de hecho, ha condicionado a las demás.

4.3.1. Métodos para el desarrollo

Desde el punto de vista metodológico, el objetivo fundamental de un método de desarrollo de STR es ayudar al diseñador a refinar el sistema teniendo en cuenta aspectos temporales a lo largo del ciclo de vida.

Para lograrlo, se han propuesto diversas técnicas que, empleando unas notaciones, formas de razonar y métodos de descomposición permiten asegurar un nivel de confianza adecuado en el sistema en desarrollo.

A) Método de Análisis y Diseño Estructurado para Tiempo Real.

Es una extensión de las técnicas estructuradas de análisis y diseño de sistemas enunciadas en el Capítulo 3 a las que se ha incorporado una serie de elementos que parecen necesarios para la descripción de un sistema de tiempo real.

Desde el punto de vista metodológico, se basa en la creación de un modelo lógico (o «esencial» por utilizar la terminología de [13]) en el que se incluyen las actividades que debe realizar el sistema y los datos que debe almacenar y un modelo físico (o de «implementación») que indica cómo el sistema se desarrolla en una tecnología determinada.

A lo largo del proceso de refinamiento no sólo cambia la notación y el nivel de detalle de la descripción sino que el enfoque del usuario también lo hace. La Figura 24 resume los conceptos empleados en una tecnología de desarrollo estructurada. Podemos observar en la figura cómo los modelos lógicos (independientes de la implementación) y los físicos (dependientes de la implementación) se subdividen en submodelos, cada uno de ellos enfocado hacia aspectos concretos. Podemos ver también el uso de diversas notaciones para acabar finalmente en el código

El proceso de construcción del modelo (lógico o físico) se realiza en varias etapas. En cada una de ellas se refina la información de los transformadores de datos o de los almacenamientos de información. En la Figura 25 se pueden ver tres niveles de descomposición de un modelo. El primero, diagrama de contexto, representa la relación con el exterior, los demás desarrollan los transformadores hasta que su actividad sea comprendida totalmente.

Históricamente, estos métodos han surgido junto con unas notaciones gráficas que permitían describir el sistema en desarrollo. Revisaremos estas notaciones en la siguiente Sección.

B) Método Statemate.

Harel desarrolla una técnica que ha tenido aplicación en el desarrollo de STR y que ya se está comercializando soportada por un producto CASE (Statemate) [15].

La idea base es poder describir y razonar sobre el comportamiento del sistema de tiempo real combinando simultáneamente diversas perspectivas. El refinamiento del modelo

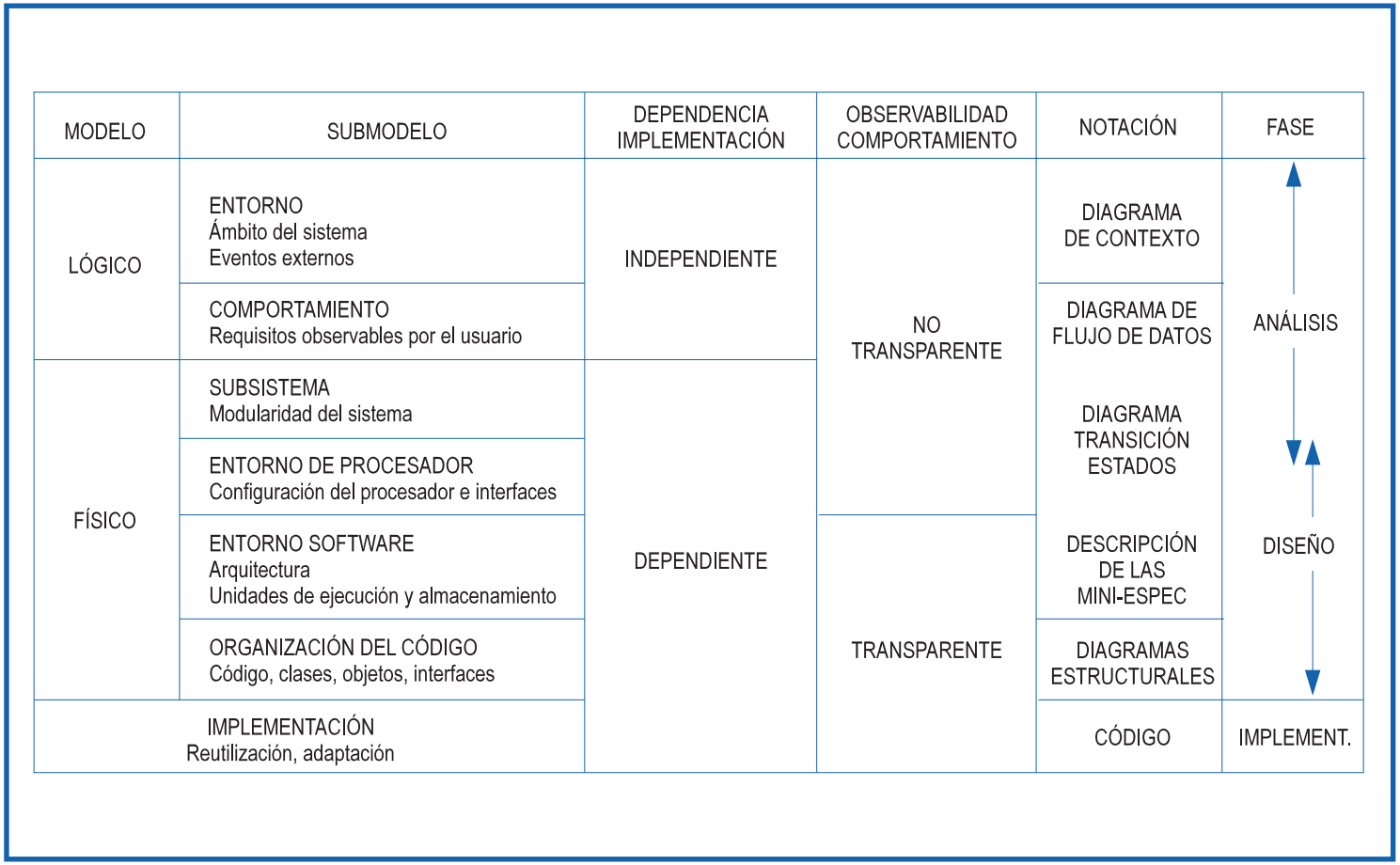


Figura 24 - NIVELES DE ABSTRACCIÓN EN EL DESARROLLO DE UN STR -

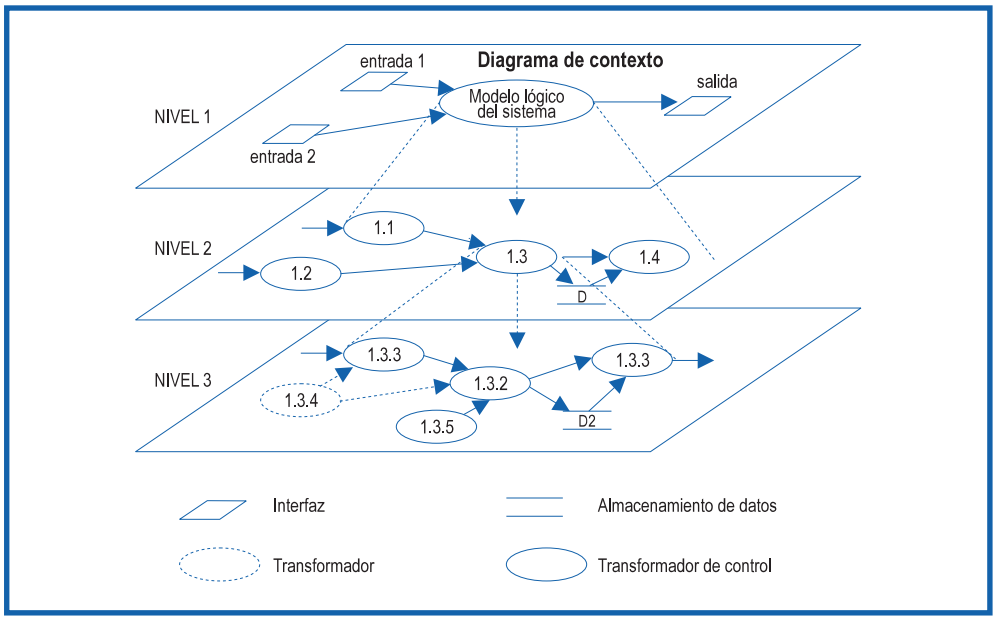


Figura 25 - REFINAMIENTO
DE UNA TECNOLOGÍA DE SOFTWARE ESTRUCTURADA -

del sistema se hace en un ciclo de trabajo en el que es posible ejecutar los modelos y prototipar el STR a realizar.

La descripción del STR se basa en el empleo de tres notaciones de forma combinada (fuertemente implicadas en el método): **diagramas de actividad** (similares a los diagramas de flujo de datos) empleados para representar las actividades del sistemas y sus interrelaciones, **diagramas de estado extendidos** que representan la evolución del control de un diagrama de actividad (cada nivel de descripción de un diagrama de actividad puede tener asociado un diagrama de estado) y **diagramas de módulos** para indicar los elementos de la arquitectura del sistema.

Lo más significativo de Statemate es el uso de diagramas de estado extendidos ya que los diagramas de actividad son similares a los diagramas de flujo de datos. Statemate utiliza

una extensión de los diagramas de estado denominados «Statecharts» para conseguir resolver parcialmente los problemas derivados de la explosión de estados. Para ello, propone utilizar un diagrama de estados con las siguientes características:

– **Jerarquización**. Los estados se descomponen en subestados a diferentes niveles mejorando la comprensión del comportamiento del STR.

– **Ortogonalidad**. En un nivel dado, los estados se agrupan en conjuntos de tal forma que el estado del sistema es una constelación de estados (uno por cada subconjunto).

– **Difusión instantánea de cambios de estado**. Cualquier evento que puede producir un cambio de estado es comunicado instantáneamente a todos los estados a los que afecta (necesario en el caso de que existan componentes ortogonales).

La gran ventaja de Statemate para la descripción de STRs es que permite describir aspectos concurrentes sin perder modularidad. La Figura 26 representa esquemáticamente la forma en la que se relacionan las actividades para la construcción de un STR.

Todo lo que hemos indicado hasta ahora es utilizado en la fase de análisis para poder expresar todos los requisitos del STR. A la hora de llegar a la fase de diseño, es necesario, sin embargo, incluir otros aspectos que aparecen en sistemas de tiempo real y que son menos empleados en otros tipos de sistemas. Estos aspectos no se tienen en cuenta en la fase de análisis de requisitos pero en la de diseño adquieren toda su importancia. Destacamos los siguientes:

1) Control de dispositivos hardware (líneas de comunicación, terminales, recursos hardware).

2) Caracterización de los mensajes. Llegada de mensajes a intervalos regulares, con tasas de llegada fluctuantes y con diferentes prioridades.

3) Control de condiciones de fallo con diferentes mecanismos de recuperación.

4) Dimensionamiento de memorias tampón (bufferes) y control del uso concurrente de los mismos.

5) Interfaz con relojes de tiempo real con posibilidad de definir y activar temporizadores.

6) Análisis de prestaciones para determinar cuellos de botella.

7) Garantizar plazos de respuesta a partir de estimaciones de tiempos de ejecución.

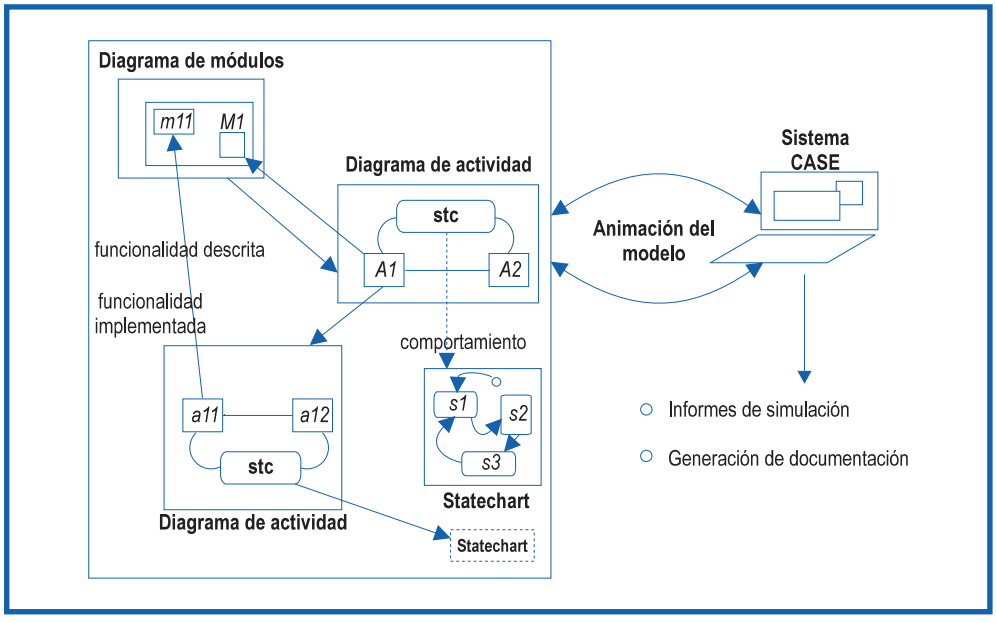


Figura 26 - EJEMPLO DEL MÉTODO STATEMATE -

Para cubrir este conjunto de aspectos de diseño se requiere modelar algunas funciones del sistema operativo tales como el manejo de memoria compartida y la sincronización en su acceso. No entraremos en técnicas concretas para la implementación de sistemas de tiempo real. En Laplante pueden verse algunas técnicas concretas [19].

4.3.2. Notaciones para la descripción de los sistemas de tiempo real

Las notaciones empleadas para la descripción de sistemas de tiempo real suelen ser extensiones de las utilizadas para la descripción de los requisitos de sistemas de información (sin restricciones temporales) a las que se han incorporado nuevos elementos relacionados con aspectos de control de la evolución dinámica del sistema: básicamente la descripción de los requisitos temporales del sistema y los cambios de estado asociados. Comentaremos las características de las notaciones empleadas en las dos técnicas que estamos presentando.

1) Notación SA/RT.

La Figura 27 describe los elementos básicos de una extensión de las notaciones clásicas de análisis estructurado adaptada a STR conocida como SA/RT.

La figura describe uno de los primeros niveles de descripción del modelo lógico de un STR (en una situación real no aparecerían todos los elementos gráficos representados en el mismo nivel; las interfaces con el exterior sólo aparecen en el diagrama de contexto) en el que podemos distinguir los siguientes elementos gráficos:

a) **Flujos continuos y discretos**. En los STR es necesario representar flujos de información **discretos** en los que la información sólo está presente en un momento (por ejemplo, un mensaje enviado desde una entidad a otra) y flujos

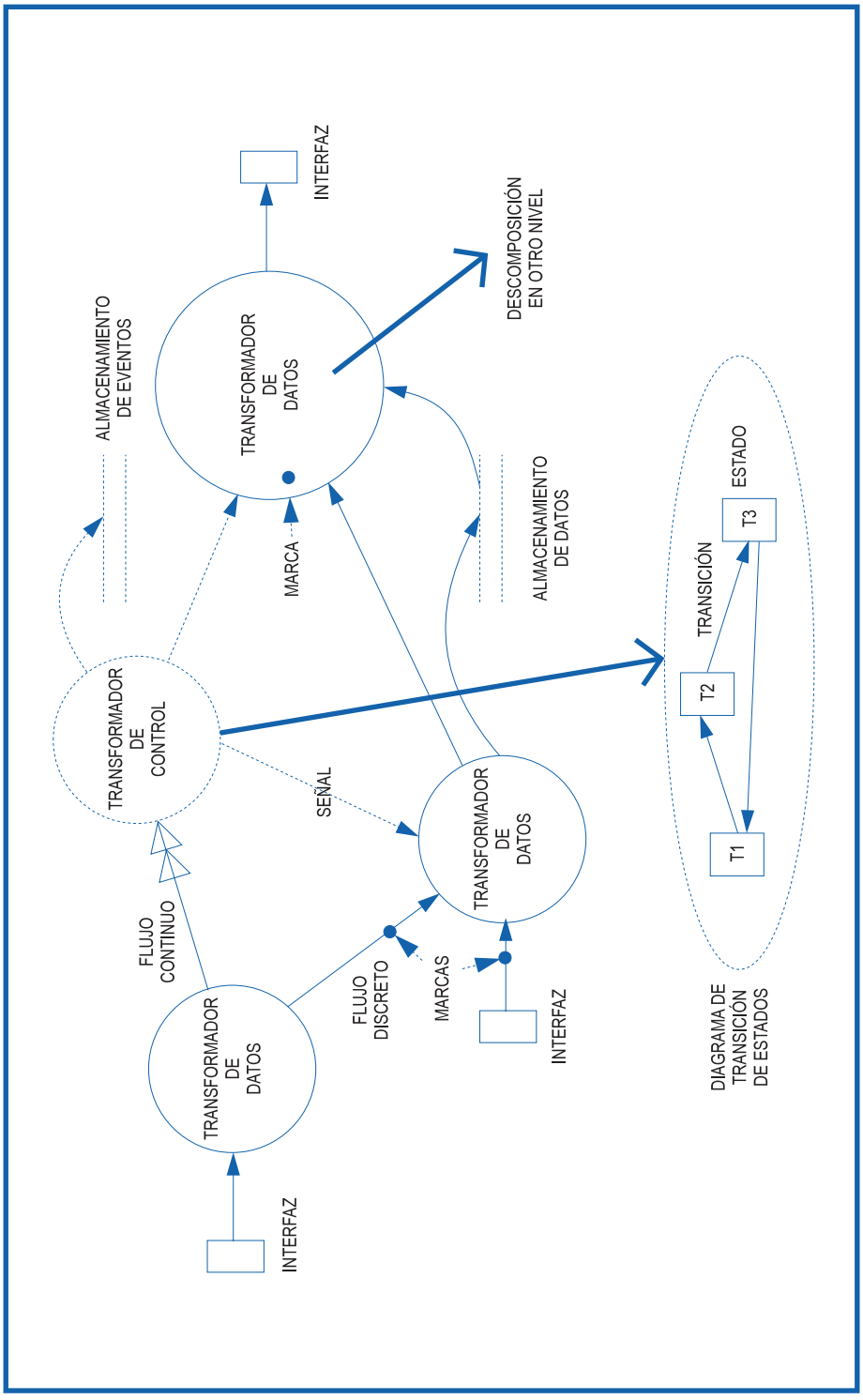


Figura 27 - EJEMPLO DE USO DE SA/RT -

continuos en los que la información está siempre presente (por ejemplo, en la medida del peso de una cabina de un ascensor o la posición de un avión). Ambos tipos de flujo son necesarios para construir el modelo lógico de un sistema de control de tiempo real.

b) **Señales**. Los sistemas de control suelen generar **señales** (no llevan información) utilizadas para activar o desactivar tareas, avisar de la existencia de algún problema, generar una alarma, etc. Es útil distinguirlas de los flujos de información porque generalmente la generación de señales está asociada a cambios de estado y, por tanto, a la evolución dinámica del sistema.

c) **Transformadores de datos**. Son los que representan el procesamiento de la información recibida y la generación de los datos de salida.

Un transformador de datos lee información a través de su flujos de entrada, la procesa y genera una información que aparece en sus flujos de salida. Si volvemos a la Figura 27, su interpretación no depende sólo del conocimiento de los símbolos gráficos. Es necesario entender también la interpretación de los diferentes transformadores. Éstos tienen asociada una «mini-especificación» en la que están contenidas las restricciones temporales y las transformaciones de los valores de los datos recibidos.

d) **Transformadores de control**. Se utilizan para representar la generación de señales (de activación o desactivación) y cambios de estado. Por ello, tienen asociado un diagrama de estados.

En cada nivel de descripción del sistema existe un transformador de control que regula el intercambio de

información y que permite controlar la dinámica del sistema controlado.

e) **Almacenamiento de datos o eventos**. El estado del sistema es función de los valores de entrada y de los valores almacenados. Estos valores almacenados pueden ser datos o eventos que han sucedido en el sistema. Los cambios de estado pueden también depender de ellos.

Se suele distinguir entre almacenes de datos y almacenes de eventos en función del tipo de información que alberguen.

f) **Diagramas de transición de estados**. Representación de los estados del sistema y transiciones entre ellos. Las transiciones implican determinar el evento que la produce (o combinación de eventos) y las acciones que se generan debido al cambio de estado.

Esta relación entre los diagramas de estado y la representación del flujo de información es muy importante y característica de los sistemas de control.

2) Notación Statemate.

Ya hemos mencionado que Statemate emplea tres notaciones inter-relacionadas: diagramas de actividad, diagramas de módulos y diagramas de estado extendidos.

La Figura 28 representa un diagrama de actividad y un diagrama de estados asociado muy sencillo correspondiente a un modelo de un cajero automático. En ella podemos ver la estructura. Como vemos, existen dos grandes estados: conectado y desconectado. Cuando está en estado conectado puede estar en espera o en servicio (uso de la estructura jerárquica entre estados) y así continúa la descripción a otro nivel.

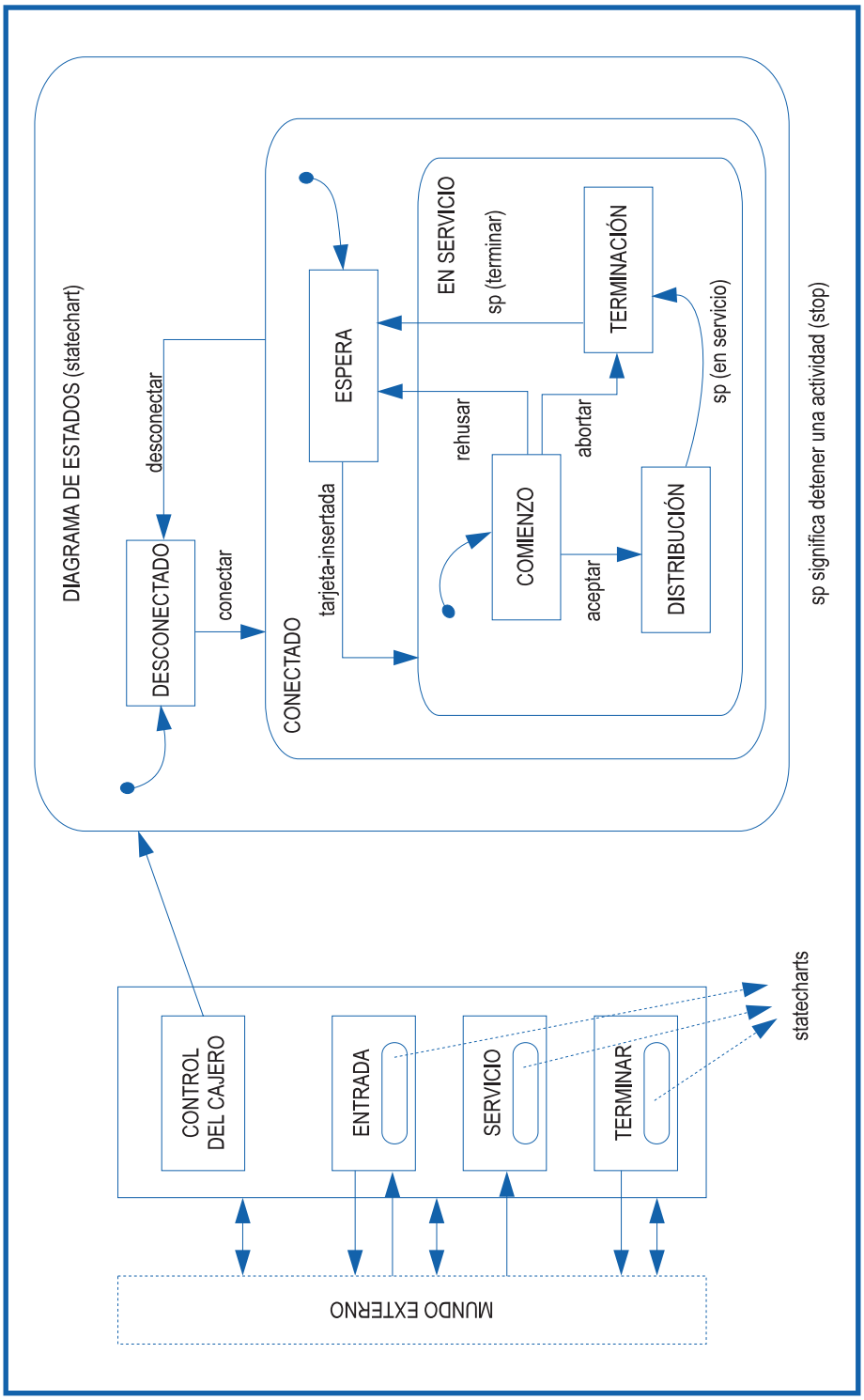


Figura 28 - EJEMPLO DE USO DE STATEMATE -

Centremos ahora la atención en la forma en la que se pueden expresar los requisitos de tiempo real. Statemate utiliza dos mecanismos: temporizadores y planificación retardada de tareas.

El **temporizador** es un evento que aparece transcurrido un plazo temporal contado a partir de otro evento (por ejemplo, el que se genera automáticamente cuando el sistema entra en un estado). Un ejemplo es:

tm (en (estado1), 14 seg.)

En este ejemplo, se define un temporizador que se activa cuando han transcurrido 14 segundos desde que el sistema ha entrado en el estado 1.

La **planificación retardada** se utiliza para asociar la activación de una tarea a un evento (puede ser también un temporizador) y un tiempo. Un ejemplo sería:

sc (tm (en (estado), 14 seg.), 10 seg.)

En el ejemplo, la tarea se activará 10 seg. después de haber finalizado el temporizador anterior.

Para la fase de diseño se suelen emplear notaciones muy *ad hoc* (en el caso de SD/RT derivadas de la empleada en SA/RT descrita anteriormente).

4.3.3. Razonamiento sobre sistemas de tiempo real

4.3.3.1. Razonamiento temporal en sistemas de tiempo real

Con independencia de la necesidad de razonar sobre la corrección de las transformaciones de datos como en cualquier otro tipo de sistema de software, lo que es específicamente propio de los sistemas de tiempo real es el **razonamiento temporal**.

Por razonamiento temporal se entiende la capacidad de asegurar que los requisitos temporales expresados en el modelo lógico y en el modelo de implementación del sistema se han satisfecho mediante análisis de la descripción del STR.

Lo que es más dificil de representar a la hora de construir el modelo lógico son las restricciones temporales. Usualmente, eso se hace asociando a los transformadores de datos una mini-especificación y haciendo que ésta contenga las restricciones temporales deseadas por el usuario. De esa forma no son visibles en el lenguaje gráfico sino en la notación empleada para la mini-especificación.

Asociar una restricción temporal no implica que se satisfaga en el sistema final; el que eso se logre o no dependerá de la forma en la que ese modelo lógico del sistema se convierta en un modelo físico incorporando restricciones de diseño y, finalmente en una implementación. El razonamiento temporal se efectúa, por tanto en tres fases diferenciadas:

1) En la construcción del modelo lógico se trata simplemente de asociar a las transformaciones adecuadas los requisitos deseados por el usuario y asegurar la consistencia de los mismos.

2) En la fase de diseño se trata de asociar a las tareas en las que se ha decidido organizar el sistema los requisitos temporales y generar, a partir de ellos, estimaciones de tiempos de ejecución de las mismas. Obviamente, se pretende que las estimaciones sean compatibles con los requisitos temporales establecidos previamente.

3) Finalmente, la fase de implementación debe asegurar que los tiempos de ejecución reales están dentro de los márgenes establecidos en la estimación. Con el conjunto de tiempos

de cómputo de todas las tareas del sistema será posible analizar los plazos de respuesta del sistema.

La Figura 29 resume el razonamiento temporal que es posible en las diferentes fases de un diseño de un STR. Es interesante observar el creciente papel que juegan las estimaciones en la fase de diseño. A partir de las estimaciones de tiempos de ejecución puede analizarse el cumplimiento de las especificaciones del producto previamente a la implementación del sistema.

Hay otro aspecto importante en un STR que es el de poder asegurar formalmente que un STR satisface aspectos concretos de interés en un STR. Existen dos líneas de actuación en las que han existido resultados de aplicación industrial: la comprobación de propiedades concurrentes y la planificabilidad del sistema. A pesar de la existencia de diversas técnicas surgidas del campo académico [17], su uso industrial es aún escaso. Al final, la validación de un STR depende de los métodos de prueba.

4.3.3.2. Prueba de sistemas de tiempo real

Si bien los métodos descritos anteriormente contribuyen a mejorar la calidad de los productos obtenidos, al forzar una disciplina en el proceso de refinamiento y a demostrar la satisfacción de algunas propiedades, la validación del sistema mediante pruebas sigue siendo necesaria.

El usuario (probador del sistema) define una secuencia de acciones que debería seguir la evolución dinamica del modelo lógico para que éste sea aceptable. Una vez iniciada la animación del modelo, el usuario puede comprobar si se satisface la secuencia de acciones predefinida. Obsérvese que este mecanismo es conceptualmente similar al uso de casos de prueba en sistemas de software convencionales.

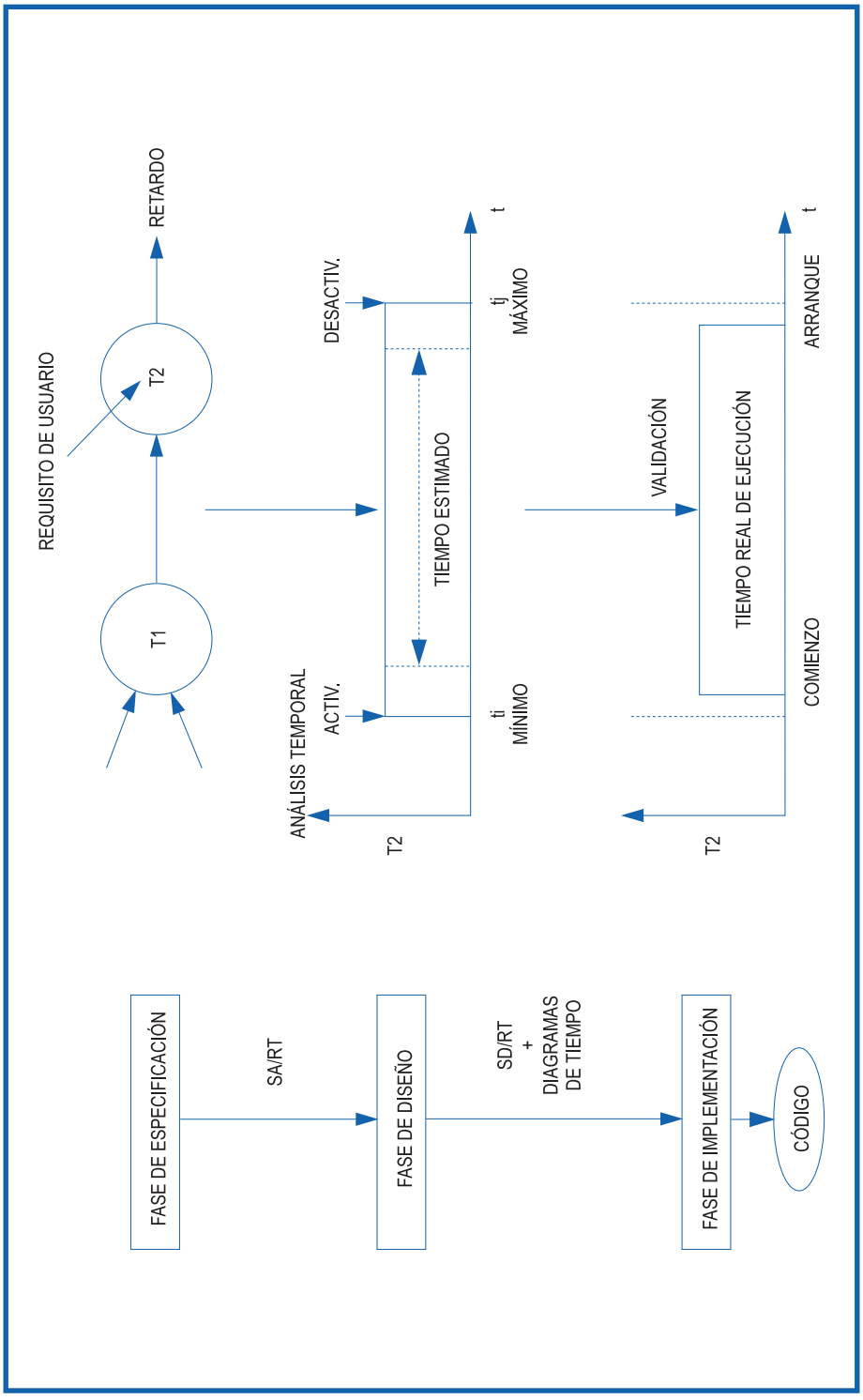


Figura 29 - FASES Y OBJETIVOS EN EL RAZONAMIENTO TEMPORAL -

Los métodos estructurados para sistemas de tiempo real permiten también analizar la evolución dinámica del sistema mediante **reglas de ejecución**. Estas reglas se basan en el movimiento de marcas sobre los transformadores de datos, los flujos o los almacenamientos en función de la evolución del sistema (dependiente de los cambios de estado). Obsérvense las marcas en la Figura 27.

Esta capacidad de «animación» del modelo ha permitido abrir un nuevo dominio de **validación dinámica** de los modelos de STR además de la validación sintáctica y estática común desde hace varios años.

Desde un punto de vista práctico, no obstante la aparición de estas nuevas técnicas de animación, la mayor parte de las técnicas empleadas hoy día se basan en la ejecución de pruebas sobre el código ya que la animación de modelos lógicos y físicos sólo puede asegurar la consistencia de los requisitos temporales. En este sentido, podemos destacar la aparición de herramientas que consisten en la simulación del entorno de ejecución del sistema con facilidades para la simulación del soporte físico final, las entradas y salidas y el sistema operativo.

4.3.4. Sistemas CASE para STR

Los sistemas CASE empleados para el desarrollo de sistemas de software cualesquiera poseen un conjunto de herramientas que también son útiles para el desarrollo de un STR; no obstante, los aspectos que debería tener un sistema CASE para el desarrollo de STRs y que no se encuentran en los sistemas CASE convencionales son los siguientes:

a) Representación de la evolución temporal de un sistema (posiblemente mediante animación de modelos y representación gráfica de su evolución).

b) Análisis de la satisfacción de los requisitos temporales para un diseño o implementación determinado.

c) Soporte a la prueba de STR con conexiones a la plataforma de ejecución real que se utilice.

d) Simulación del entorno con la generación de eventos adecuada al STR para la posible generación (interactiva o no) de casos de prueba.

La utilización de las técnicas vistas en el apartado anterior se ha visto favorecida por la aparición de sistemas CASE que las soportan. Estos sistemas incorporan algunas de las necesidades mencionadas [20].

En el caso de SA/RT es muy común encontrarse con sistemas CASE que soportan varios métodos del mismo tipo y, en algunos casos, con generación de algún tipo de código. Existe una línea de trabajo en la que se combinan los sistemas tipo SA/RT con métodos formales para poder disponer de prototipos ejecutables y animar los modelos. Este es el enfoque utilizado en la tecnología IDERS [21] en la que el entorno de soporte posee este tipo de herramientas. En la Figura 30 podemos ver cuatro componentes principales:

1) Subconjunto de herramientas para la construcción de modelos lógicos y físicos incluyendo un núcleo de ejecución de éstos con el fin de poder animar los modelos.

2) Visualizador de código objeto incluyendo el simulador del S.O. para poder ver su evolución dinámica.

3) Gestor del modelo de proceso empleado para que la activación de las herramientas y las actividades del equipo de trabajo sean conformes con ellas.

4) Mecanismo de integración de control que asegura el intercambio de información entre los diferentes ejecutores de un prototipo heterogéneo asegurando la consistencia de los valores temporales.

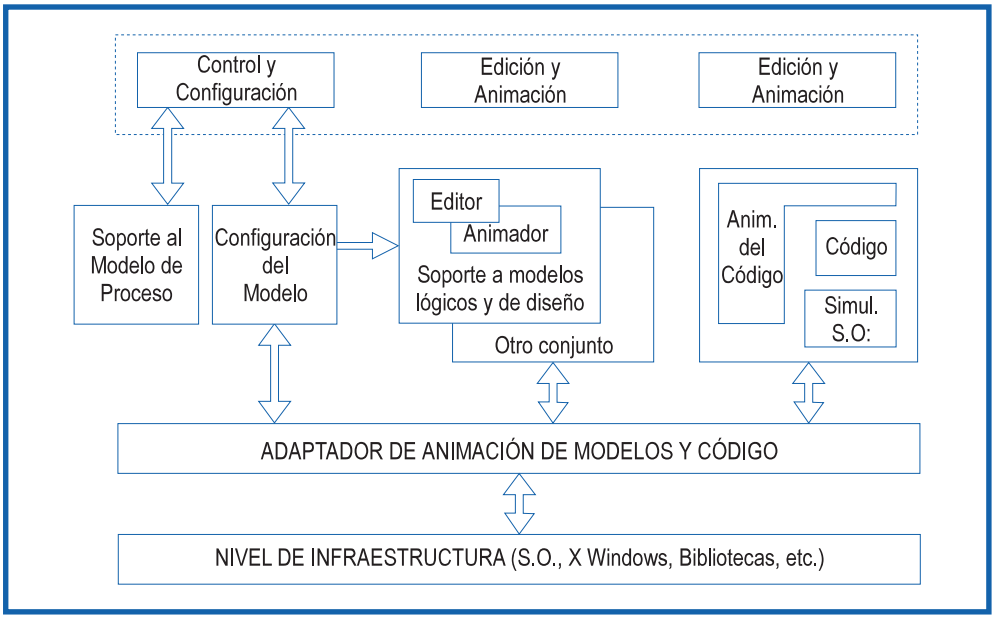


Figura 30 - ENTORNO DE HERRAMIENTAS IDERS -

En el caso de Statemate [15] la estructura del entorno de herramientas permite no sólo describir el sistema a través de ediores de los tres lenguajes existentes sino también simular el funcionamiento del sistema e, incluso, generar prototipos de la interfaz de usuario.

La Figura 31 describe los principales componentes de herramientas disponibles en Statemate. De la figura podemos ver que se trata de un sistema CASE basado en la existencia de un repositorio común en el que se encuentra la información relativa al STR que se está desarrollando (más centralizado, por tanto, que en el caso de IDERS).

4.3.5. Directrices industriales

La última de las componentes tecnológicas presentadas en el Capítulo anterior corresponde a la posibilidad de disponer de soluciones

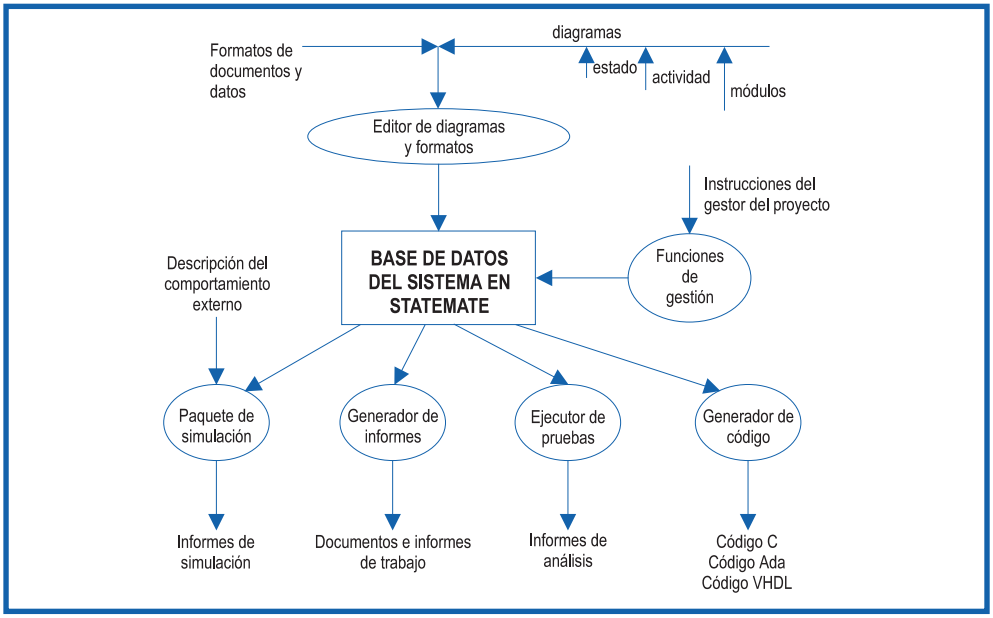


Figura 31 - ENTORNO DE HERRAMIENTAS STATEMATE -

específicas para un dominio o subdominio concreto aprovechando la experiencia acumulada en experiencias anteriores.

En el caso de STR, empiezan a surgir soluciones ensayadas y probadas que parecen válidas para su extrapolación a diferentes tipos de sistemas. Nos referiremos a tres aspectos concretos:

1) Algoritmos de planificación. Existe mucha experiencia acumulada y bien documentada sobre los tipos de planificación de procesos más adecuado a cada tipo de sistema de tiempo real.

2) Primitivas y funciones del sistema operativo. Un problema importante es determinar el conjunto de servicios del S.O. que son necesarios. El IEEE a través de la norma POSIX ha determinado conjuntos mínimos (denominados perfiles) para tipos de sistemas.

3) Arquitecturas de STR concretos. Al nivel de diseño arquitectónico se han propuesto estructuras que la experiencia dicta como las más adecuadas con el fín de reutilizarlos en diseños posteriores. Aún así, los conceptos de reusabilidad no se aplican fácilmente a STR y sigue siendo una línea de investigación abierta.

4.4. Resumen

Este Capítulo se ha centrado en las tecnologías de software que nos permiten el desarrollo de sistemas de tiempo real. En él hemos pretendido enfocar la atención en los aspectos específicos de estos sistemas y cómo influyen en el desarrollo de tecnologías de software específicas.

A lo largo del Capítulo hemos utilizado los requisitos temporales como elemento motriz de todos los aspectos planteados: su descripción, análisis, relación con aspectos de procesamiento de información, etc.

No es posible condensar en un Capítulo todas las tecnologías existentes para STR y hemos preferido apoyarnos en dos de ellas destacando dos ideas fundamentales:

1) La descripción de los requisitos temporales debe considerarse desde las primeras fases del ciclo de vida. Existen métodos y notaciones que permiten expresarlos y manipularlos.

2) La comprobación de que el sistema diseñado satisface los requisitos temporales requiere de métodos y herramientas CASE que empiezan a estar disponibles. Concretamente, el uso de técnicas de prototipado incremental y de animación de modelos y código son muy prometedoras.

5

Gestión del desarrollo del software

5.1. Introducción

Una de las consecuencias directas del incremento de complejidad de los sistemas de software (recuérdese el Capítulo 1) es que su tamaño hace prácticamente inabordable su desarrollo por una persona o incluso por un grupo reducido de éstas. Derivada de esta misma complejidad, ningún componente del equipo de trabajo puede dominar todos los detalles implicados en el desarrollo siendo la causa de la especialización del equipo humano con diferentes perfiles. Esta especialización ha estado históricamente ligada a la tecnología y al modelo de ciclo de vida con el que se desarrollaba el producto.

La utilización de un modelo de ciclo de vida (cualquiera de los descritos en el Capítulo 2) supone que el desarrollo del producto pasa por una serie de fases en las que se realizan unas actividades, se generan unos documentos y se requiere un tiempo y unos recursos para su realización. La calidad del producto final dependerá de cómo se efectúen esas actividades.

Pues bien, el objetivo de la gestión del desarrollo de un producto de software consiste en la utilización de la forma más eficaz posible de los recursos humanos y materiales asignados para conseguir el producto en los plazos temporales y con la calidad adecuada. Debemos gestionar, por tanto, tanto el producto como el proceso.

Por **gestión del producto** nos referimos a los procedimientos necesarios para convertir unas necesidades expresadas informalmente

por el usuario en un producto de software que las satisfaga. Es el dominio de las tecnologías de software genéricas presentadas en el Capítulo 3 y las específicas para los sistemas de tiempo real descritas en el Capítulo 4.

La **gestión del proceso** se refiere a los procedimientos establecidos por los gestores del proyecto de desarrollo para optimizar los recursos y controlar el desarrollo a efectos de establecer las medidas correctoras que sean precisas con el objetivo de obtener la calidad del producto deseada.

Para muchos autores, la gestión de un proyecto de desarrollo de software es básicamente una gestión de la calidad del producto que se manifiesta en diversos aspectos de éste y de los procesos implicados en las fases del desarrollo. En todo caso, la calidad final del producto no sólo depende de la tecnología de software empleada sino de cómo ésta se utiliza a lo largo del desarrollo.

Aún admitiendo el papel central que la gestión de la calidad juega, nosotros adoptaremos en este Capítulo una visión un poco más general incluyendo también la gestión de recursos humanos y la planificación general de actividades. Ambos conceptos van ligados en la gestión del desarrollo.

Desde este objetivo genérico, la gestión de **calidad de un sistema de software** implica la actuación sobre las siguientes áreas de gestión:

A) Planificación del desarrollo y del mantenimiento.

Implica la gestión de los recursos tanto humanos como materiales necesarios para obtener el producto de software incluyendo el entrenamiento requerido por los componentes del equipo de trabajo y los potenciales usuarios del sistema a desarrollar.

B) Control de la calidad técnica del producto.

Establecimiento de métricas y procedimientos de validación para asegurar un nivel de calidad determinado.

Entre las actividades contempladas consideramos las siguientes: revisiones y auditorías de las actividades, documentos y productos intermedios generados; actividades de prueba, gestión de errores y medidas correctoras y, en general, el control de la correcta aplicación de los componentes de la tecnología de software empleada para asegurar la validez del producto.

C) Control de versiones y configuraciones.

Entre los aspectos considerados está la gestión (protección, almacenamiento, control de acceso, uso y distribución) de la documentación y de los productos de software generados a lo largo de las fases del desarrollo así como su mantenimiento durante la evolución del producto.

D) Gestión de riesgos.

Entre los aspectos importantes se encuentra el control de proveedores externos y de la posible subcontratación así como el control de calidad de los componentes entregados por los mismos; el aprovisionamiento de componentes requeridos y el análisis de las soluciones técnicas existentes.

Estos componentes no están aislados. En el desarrollo de un producto de software concreto se imbrican de acuerdo a procedimientos genéricos establecidos por los gestores del proyecto.

5.2. Validación de sistemas de software

5.2.1. Conceptos básicos

Las técnicas de validación de sistemas de software están ligadas al desarrollo del producto y podrían haberse considerado como componentes de la tecnología de software. No obstante, he preferido incluirlas en este Capítulo porque afectan globalmente a todo el desarrollo (no a una fase en concreto) y porque están íntimamente ligadas a las técnicas de gestión del proyecto (en realidad se puede decir que forman la base que asegura la calidad técnica del producto).

La **validación** se refiere a asegurar que estamos haciendo el producto «bien». En otras palabras, es válido para nuestros propósitos. La *validez* está ligada no sólo a comprobar la satisfacción de la funcionalidad sino también a asegurar que satisface unas normas predeterminadas.

Hay, por tanto, dos perspectivas diferentes para definir la validez: una, subjetiva, desde el usuario como adecuación a sus necesidades; otra, desde el diseñador, más objetiva en relación con la satisfacción de determinados parámetros. En todo caso, la validez no viene fijada externamente al propio proceso de desarrollo sino que viene ligada al mismo y definida junto con la generación del producto.

Existe otro término con el que muchas veces se asocia: verificación. La **verificación** tiene como objetivo asegurar que el producto es funcionalmente correcto (su comportamiento es exactamente el deseado). La verificación es un problema de corrección matemática (adecuación entre la especificación y la implementación) y requiere el empleo de técnicas formales dado que la especificación deber ser manipulable matemáticamente.

Desde esta óptica, la verificación puede ser una de las técnicas empleadas para validar un sistema, aunque la validación es mucho

más amplia (un proceso formalmente verificado puede que no sea válido al no satisfacer otros requisitos de adecuación que hayamos establecido).

Desgraciadamente, las técnicas de verificación formal están muy lejos de poder ser empleadas de una manera sistemática para demostrar la corrección de sistemas grandes de forma completa. En el momento actual, sólo se pueden verificar aspectos muy parciales de algunos sistemas.

Si no podemos proceder de forma general verificando el programa sí podemos aplicar técnicas que nos permitan probar que lo que estamos haciendo es válido. A estas técnicas se las suele conocer como de **prueba de software** («testing»).

Comentamos en un Capítulo anterior que el objetivo de la prueba de software (tanto del código como de la especificación) es demostrar la presencia de errores (con el fin de conseguir su eliminación posterior) pero que no permiten demostrar su ausencia. Desgraciadamente, si es así, lo es debido a limitaciones de las técnicas empleadas y no como objetivo implícito de la prueba.

Una formulación complementaria es decir que la prueba de programas pretende entregar al usuario el producto con las mayores garantías de que está libre de errores (respecto de su funcionalidad) a un coste razonable en términos de recursos humanos, materiales y tiempo empleado.

Cuando se está probando un sistema de software pueden seguir errores que, poco a poco, se van espaciando. Tras la acumulación de pruebas y la consiguiente corrección del sistema, el tiempo necesario y las personas que intervienen para que se descubra un nuevo error es muy grande. Los gestores deberán decidir si conviene dar por finalizadas las pruebas o no. Obsérvese que no es un problema exclusivamente técnico sino de confianza en el sistema en desarrollo.

Una **prueba** en concreto consiste en someter al producto de software (o a una parte del mismo) a una evaluación para conocer si se comporta de acuerdo con una especificación tomada como referencia. Para realizar una prueba concreta (un caso de prueba) se requiere:

A) Una descripción de la prueba.

La descripción debe indicar el propósito de la prueba (para qué se hace), el componente de software sobre el que se realiza (podría ser un módulo, subsistema o el sistema completo), los datos de prueba que se van a entregar y los resultados esperados.

B) Una ejecución de la prueba.

Si se dispone de una descripción ejecutable del sistema (por ejemplo, código) la ejecución de la prueba implica la ejecución de ese módulo en un entorno controlado. Posiblemente, ni el sistema final ni el resto del producto están disponibles y deberán ser simulados. Esta situación obliga a disponer de un entorno específico (conjunto de herramientas que simulen a efectos de la validación el entorno de ejecución del producto de software) para la ejecución de las pruebas.

C) Una valoración de los resultados obtenidos.

Una vez obtenidos los resultados de la prueba, éstos deben compararse con otros considerados como referencia. En muchos casos, esta referencia es una especificación a partir de la cual se ha generado el programa. En otros casos son heurísticos que los diseñadores han seleccionado.

Intuitivamente, podemos decir que cuantas más pruebas se realicen mayor será nuestra confianza en el producto. Cada prueba

debe cubrir un aspecto concreto y una secuencia de pruebas nos da la **cobertura del producto** conseguida.

Conocer el grado de cobertura de las pruebas (porcentaje del código o de funciones probadas) es un elemento importante a la hora de planificar su realización.

En el Capítulo 2, al analizar el modelo de ciclo de vida en cascada y concretamente el modelo en V, asociamos un tipo de pruebas a cada fase del desarrollo: pruebas modulares, pruebas de integración, pruebas de sistema y pruebas de aceptación. Lo que no vimos en el Capítulo 2 es quienes las realizaban, cuándo y con que técnicas. Ese es el aspecto que vamos a tratar ahora ligado a la gestión del proyecto de desarrollo.

Para productos que deben interaccionar con otros y que deben ser homologados por un organismo internacional, no basta que la validación la efectúen los mismos diseñadores. En estos casos las pruebas se realizan por un laboratorio independiente, como parte del proceso de certificación. Esto sucede, por ejemplo, en muchos productos de comunicaciones (por ejemplo, un protocolo de comunicaciones) en los que la conformidad se realiza con respecto a una norma internacional publicada por un organismo internacional.

Para conseguir la homologación, el producto bajo prueba debe superar un conjunto de pruebas predefinidas y públicas; cuando lo consigue, se dice que el producto ha sido certificado.

5.2.2. Clasificación de las técnicas de prueba

Clásicamente, las técnicas de prueba se centraban en probar el código puesto que éste era el único producto disponible al que se le podía someter a unas pruebas.

La aparición de notaciones rigurosas para las fases iniciales del ciclo de vida han permitido expandir estas técnicas sobre otros productos intermedios. No obstante, centraremos la atención inicialmente sobre la prueba del código fuente y al final comentaremos su expansión a otros productos del desarrollo.

Atendiendo a *qué es lo que se prueba* podemos hablar de dos técnicas complementarias:

1) Funcional.

Pretende conocer si el código satisface la funcionalidad requerida sin preocuparse de cómo lo hace. A esta técnica se la suele conocer como de «caja negra» porque no le preocupa cómo el módulo realiza su función sino comprobar que la relación entre entradas y salidas es la deseada.

Para este tipo de pruebas se parte de la identificación de las funciones requeridas y su asociación a los módulos del sistema que se supone que las implementan. Seguidamente, se generan los datos de prueba que comprobarán si estas funciones son realmente realizadas por el software. Esta generación de datos de prueba puede hacerse ayudada por herramientas de software.

El problema básico ligado a esta técnica estriba en la dificultad de asociar las funciones incluidas en un módulo a los requisitos de usuario, dado que esa relación se suele perder en la fase de diseño. En este sentido, una de las ventajas de las técnicas de análisis y diseño orientado a objetos con respecto a las estructuradas reside en que en las primeras esa relación se mantiene mejor hasta la fase de implementación.

El segundo de los problemas es la dificultad en seleccionar un conjunto de casos de prueba que sean representativos y

en decidir a partir del resultado de su ejecución si el módulo está actuando correctamente o no. En algunos casos sabemos el resultado que debemos obtener (caso de que el módulo implemente una función matemática de la que podemos conocer realmente el resultado correcto previamente), en otros no es posible y la validación deberá hacerse en base a heurísticos y la experiencia previa en sistemas similares.

2) Estructural.

Se basa en explorar el código del programa para conocer si realiza correctamente las especificaciones del diseño detallado. A esta técnica se la conoce como de «caja blanca».

Explorar el código completo de un sistema no es sencillo en programas reales ya que las posibles combinaciones de valores con la estructura del programa impiden disponer de un conjunto de pruebas manejable. Al menos se debería asegurar que:

– Todas las sentencias del programa se ejecutan al menos una vez durante alguna de las pruebas.

– Todas las bifurcaciones del programa se ejecutan al menos una vez.

– Todos los fragmentos del programa que finalicen en una transferencia de control a otra parte del programa son ejecutadas al menos una vez.

Aunque cumplir estas condiciones no implica que se ha probado todo el código, sí que permite incrementar la confianza en el diseño efectuado. El grado de cobertura mide el porcentaje del código probado respecto del total.

Atendiendo al *método de prueba* elegido, es decir, cómo se prueba, existen dos técnicas básicas:

1) **Estática**. Implica el análisis del código fuente sin ejecutarlo. De la lectura (más o menos automatizada) del código es posible obtener bastante información sobre la calidad de código y descubrir problemas.

2) **Dinámica**. Implica la ejecución real del código bajo prueba con objeto de analizar los resultados obtenidos. Para ello, es necesario disponer de unos datos de prueba y unos heurísticos sobre los resultados.

Si bien hemos prestado atención a las pruebas del código, la utilización de técnicas formales o semi-formales permite extender las pruebas a otras fases de ciclo de vida. Concretamente, las mismas ideas de cobertura se pueden aplicar a modelos ejecutables (lógicos y físicos) cuando están descritos en una notación textual. En el caso de notaciones gráficas (que es el caso habitual) los conceptos de cobertura están asociados a los elementos del modelo y no a sentencias de un lenguaje de programación.

5.2.3. Gestión de las pruebas

La planificación y realización de las pruebas se realiza a lo largo de todo el ciclo de vida. La modificación del ciclo de vida convencional descrito en el Capítulo 2 como ciclo de vida en V, enfatiza las actividades de prueba sobre diferentes productos y documentos intermedios. Para que las pruebas se lleven a cabo es necesario establecer actividades concretas en cada una de las fases con procedimientos concretos de gestión.

Cada procedimiento de gestión de pruebas deberá fijar al menos: el responsable, los recursos necesarios, el tiempo dedicado, los

objetivos a conseguir, las técnicas a utilizar (incluyendo herramientas), los datos, código o documentos empleados o generados y las acciones a realizar en función de los resultados de las mismas.

Desde el punto de vista técnico, las pruebas se enmarcan en la revisión del diseño y del código. Los métodos más conocidos son los siguientes:

A) **Inspección de código**. Procedimiento para la revisión por un equipo humano de la implementación de un producto software.

Las técnicas de revisión de código se han revitalizado últimamente a partir de experiencias muy alentadoras en grandes industrias. Así la técnica de *sala limpia* («clean room») ha demostrado su utilidad en forma de una reducción significativa de defectos finales.

B) **Paseos por el código** («walk-throughs»). Proceso menos formal que el anterior en el que participan usuarios y desarrolladores con el objetivo de descubrir y resolver omisiones e incomprensiones de lo que hace una parte del código.

C) **Revisiones personales**. El objetivo es recuperar la «vieja» práctica de releer cuidadosamente el código antes de compilar; práctica que la aparición de estaciones de trabajo y el reducido coste en tiempo del ciclo de edición-compilación-depuración ha relegado. La consecuencia es que la estructura del código y su fácil o no comprensión queden relegadas por el «visto bueno» de la máquina dificultando la evolución posterior del sistema. Humphrey ha propuesto basar la formación del ingeniero de software en un acercamiento a la calidad del sistema vía el uso de técnicas de revisión a nivel personal que van mejorando paulatinamente con la experiencia acumulada [3].

Estas mismas ideas pueden trasladarse a la fase de diseño siempre que se cumplan dos condiciones: existan documentos de diseño susceptibles de revisión y, en lo posible, que sean ejecutables.

Disponemos de una documentación de diseño revisable cuando está descrita empleando notaciones y métodos rigurosos (no necesariamente formales). Los documentos de diseño son ejecutables cuando se basan en un modelo de computación que permite visualizar su comportamiento dinámico.

A pesar de todo lo que hemos indicado, las técnicas de pruebas disponibles han sido ideadas básicamente para sistemas secuenciales y fundamentalmente para pruebas funcionales en sistemas de información. La situación es mucho peor para aspectos relacionados con los requisitos temporales, básicos para sistemas en tiempo real o para sistemas concurrentes y distribuidos, tal y cómo hemos expuesto en el Capítulo 4.

5.3. Control de versiones y configuraciones

5.3.1. Conceptos básicos

Cualquier producto de software complejo está constituido por un conjunto de elementos que deben combinarse para formar un sistema que se entregue al usuario. Esto incluye tanto los elementos ejecutables (módulos o programas) como documentación, procedimientos de instalación, etc. para que el usuario utilice (y no sólo ejecute) el sistema en su entorno final.

Como ejemplo, un típico producto de software entregable a un usuario está constituido por un conjunto de componentes mantenidos por el sistema de ficheros entre los que podemos citar:

a) el programa (o programas) ejecutable,

b) diversos ficheros de datos (datos numéricos, texto, imágenes, etc.) necesarios para la ejecución,

c) procedimientos de órdenes del sistema operativo (p.ej. para instalación o arranque),

d) bibliotecas de módulos compilados,

e) código fuente (sólo algunas veces),

f) manual de instalación,

g) manual de usuario con los ficheros de datos, imágenes, etc. necesarios. Los manuales pueden tener una versión en formato imprimible directamente o preparados para consulta en línea,

h) un conjunto de ejemplos de uso, y

i) un fichero de explicación inicial.

En muchos casos, el sistema que se utilizará en el entorno de un usuario diferirá en algunos de sus componentes de otro (por diferencias en las versiones de la máquina, del sistema operativo, de bibliotecas, de elementos especialmente adquiridos por el usuario, de los dispositivos externos disponibles, o por otras causas). Estos elementos se definen y construyen durante el desarrollo del sistema y sirven de base para la generación del sistema final.

En productos horizontales (pensados para multitud de usuarios anónimos), el proceso de configuración debe completarlo el usuario respondiendo interactivamente durante la ejecución de un programa de configuración (éste es el sistema empleado en la instalación de la mayor parte de los sistemas de software en el mercado doméstico) que viene incluido con la distribución del producto.

Si consideramos el desarrollo y evolución futura del producto, será necesario asociar a ese producto entregable al usuario muchos elementos más. Ellos serán necesarios en la fase de mantenimiento posterior del producto o para incrementar la capacidad de la organización para realizar mejor sus futuros productos. Entre estos elementos adicionales se encuentran:

a) Documentación de análisis de requisitos (modelos lógicos).

b) Documentación de diseño.

c) Análisis de pruebas ejecutadas.

d) Diferentes versiones de implementación.

e) Lecciones aprendidas (parte de la historia del proyecto), etc.

Una de las actividades más importantes de la gestión de un proyecto de software es, por tanto, controlar la gran cantidad de componentes que se generan a lo largo del desarrollo y su interrelación. Su utilidad no sólo se circunscribe al proceso de desarrollo sino que también es necesario durante el mantenimiento del sistema en el que muchos de esos componentes evolucionan.

A la actividad relacionada con el control de la generación y evolución de estos componentes, sus relaciones y los procesos de asegurar que todo ello está asociado a las necesidades de un sistema concreto se le denomina **gestión de configuraciones y control de versiones**. Seguidamente, definiremos informalmente algunos de los términos utilizados.

Se entiende por **componente** cualquier unidad básica a partir de la cual se construye el sistema. Cada componente posee una descripción de su función, relaciones con otros componentes,

informaciones relativas al entorno de ejecución (fechas de creación, autor, etc.) así como de los ficheros que forman parte del mismo.

La Figura 32 representa esquemáticamente uno de estos componentes. La definición del componente (que puede requerir una notación especializada) debe permitir la identificación de los módulos de un catálogo que sean requeridos. El conocimiento del contexto puede ser necesario para resolver ambigüedades.

Obsérvese la existencia de un catálogo de módulos genéricos de los que puede proceder el componente en cuestión.

Configurar un sistema consiste en describir los componentes que forman parte de un sistema de software y generar a partir de ellos el sistema deseado. Por **gestión de configuraciones** se entiende el conjunto de procedimientos establecidos para controlar la configuración de un sistema y asegurar que la descripción sea correcta.

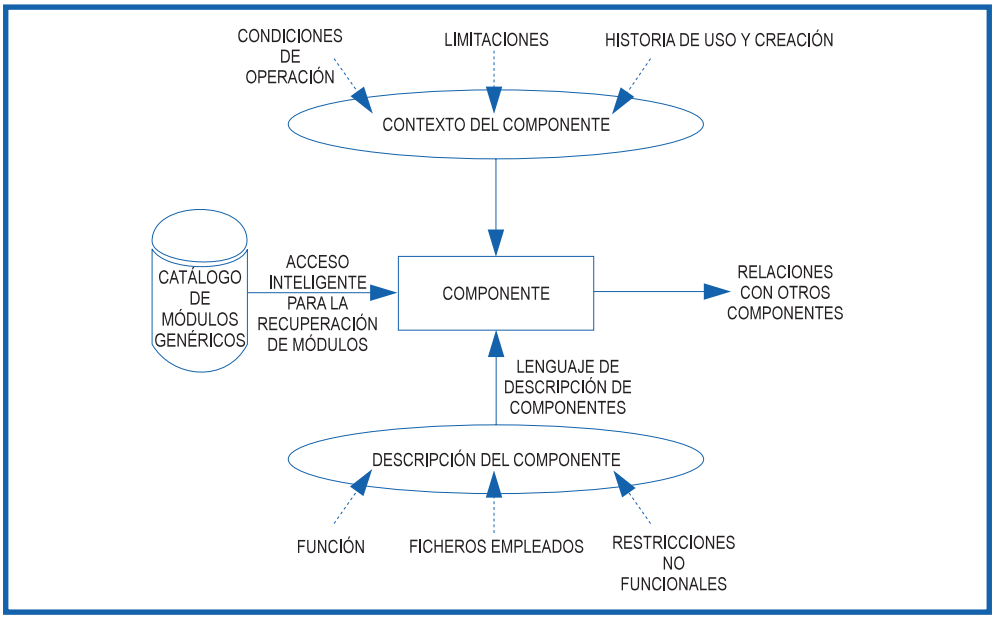


Figura 32 - ESTRUCTURA DE UN COMPONENTE GENÉRICO -

Si fijamos ahora la atención en un componente concreto y observamos su evolución durante el proceso de desarrollo y mantenimiento posterior, éste pasa por una serie de cambios debido a múltiples causas: eliminación de errores, incremento de su funcionalidad, adaptación a un entorno de ejecución modificado, etc. A estos productos intermedios se les denomina **versiones.** Los procesos necesarios para el manejo de las versiones de un componente se conocen como **control de versiones.** La Figura 33 representa una posible evolución de un componente en el que no sólo existe una única línea de evolución sino que ésta se bifurca varias veces en función de las razones que sustenten esta evolución.

No por antiguas las versiones de un componente dejan de ser útiles. En muchos casos es necesario mantener «vivas» varias versiones de un mismo componente porque ellas serán utilizadas en diversas configuraciones de un sistema de software pensadas para distintos usuarios.

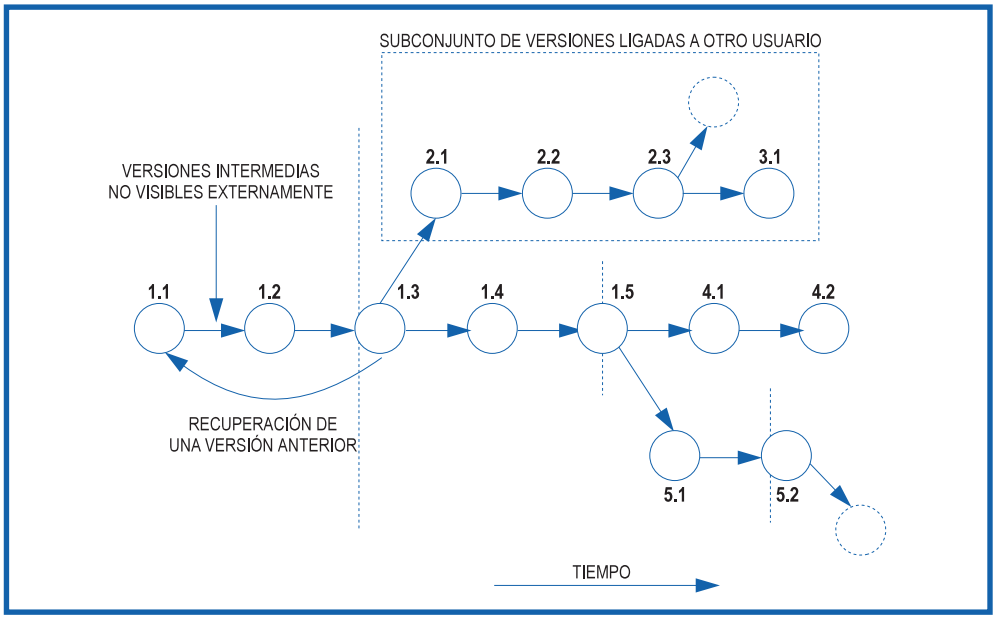


Figura 33 - EVOLUCIÓN DE LAS VERSIONES DE UN COMPONENTE -

Es interesante mencionar que las ideas de control de versiones son también útiles durante el proceso de desarrollo de un producto. En este caso la utilidad está ligada a la necesidad de conservar versiones intermedias que aún no son rechazadas o que pueden servir para otros productos. La Figura 34 describe la evolución de un producto software a través de diferentes versiones.

Como ejemplo, en el caso de esta monografía se han empleado versiones correspondientes a cada Capítulo (asociándoles ficheros de figuras) aunque podría haberse decidido emplear como componente una estructura de nivel más básico que el Capítulo. Ello hubiera permitido mayor flexibilidad para generar configuraciones (monografías diferentes) a costa de una mayor complejidad en la generación de las mismas.

Seguidamente, vamos a detallar los procesos empleados en la gestión de versiones y configuraciones a partir de la funcionalidad de herramientas software específicas.

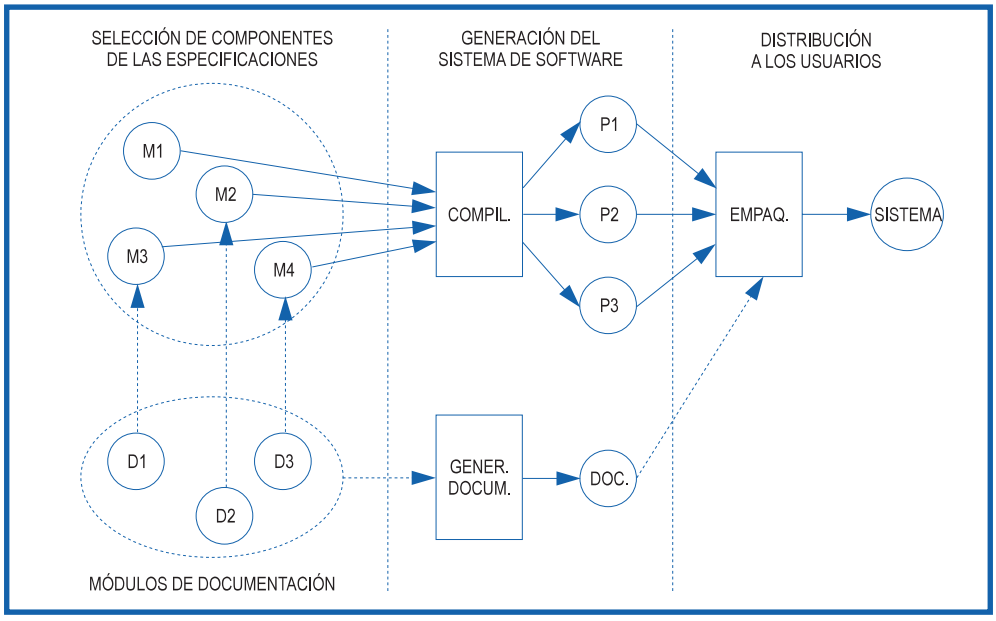


Figura 34 - PROCEDIMIENTOS DE CONSTRUCCIÓN DE UNA CONFIGURACIÓN -

5.3.2. Herramientas para control de versiones y configuraciones

Todos los procesos implicados con la gestión de configuraciones y versiones son proclives a cometer errores y, en todo caso, son tediosos; no es extraño, por tanto, que se apoyen en herramientas de software. Existen muchas herramientas para ayudar en la gestión de configuraciones y versiones y éstas han evolucionado mucho en los últimos años.

Podemos hablar de tres generaciones de herramientas:

1) La **primera generación** disponía de herramientas para control de versiones y de configuraciones separadas y no integradas con el resto del entorno de desarrollo. Por otro lado, el conocimiento que estas herramientas tenían del contenido de los ficheros era nulo y el usuario no podía verse ayudado en ello (los consideraban como ficheros de texto sin estructura); así, un programa o una documentación se trataban de la misma manera. Aunque los componentes tenían asociados números y fechas, eso no era suficiente para capturar la información contenida en el módulo. De hecho, se apoyaban en el sistema de ficheros del sistema operativo.

   Un ejemplo de estas primeras herramientas es **SCCS** (Source Code Control System, sistema de control del codigo fuente) ideada en la década de los setenta e incorporada al sistema UNIX para soportar el control de versiones y **make**, también en UNIX y sobre la misma fecha para soportar control de configuraciones. Posteriormente a SCCS, aparece **RCS** admitiendo una estructura de versiones en forma de árbol y ofreciendo una interfaz a «make». Ofrece, además, diversas ventajas adicionales para bloquear el uso de los ficheros.

2) La **segunda generación** constaba de herramientas integradas con el resto del entorno de desarrollo de software. Conceptualmente, estas herramientas consideraban los

componentes como objetos con atributos y no como ficheros. La base ya no estaba en el sistema de ficheros sino en una base de datos que controlaba estos componentes. Finalmente, la descripción de cada uno de los componentes se homogeneizaba mediante la utilización de un lenguaje de descripción (se suponía que el lenguaje de implementación no soportaba chequeo entre módulos ni podía expresar la interconexión entre módulos y eso lo debía realizar el usuario externamente).

Las ideas de versiones y configuraciones son también aplicables a otras fases del ciclo de vida en las que aún no existe ningún producto ejecutable. Como ejemplo, la documentación (texto) generada en el desarrollo de un producto también está sometida a control de versiones.

3) La **tercera generación** surge con la aparición de nuevos lenguajes de programación basados en el concepto de **módulo**. Éstos poseen construcciones separadas para la interfaz de los módulos (visible por otros módulos) y la parte ejecutable separadas; ello permite extraer automáticamente la relación entre ellos y construir la configuración requerida. Con ellos las dependencias entre las versiones se construyen automáticamente.

5.4. Métricas

El proceso de planificación del desarrollo de cualquier sistema debe hacerse partiendo de una estimación del trabajo a realizar. Sólo a partir de ello es factible conocer los recursos necesarios y el tiempo necesario para su realización.

Una **métrica** es una medida efectuada sobre algún aspecto del sistema en desarrollo o del proceso empleado que permite,

previa comparación con unos valores (medidas) de referencia, obtener conclusiones sobre el aspecto medido con el fin de adoptar las decisiones necesarias. Con esta definición, la definición y aplicación de una métrica no es un objetivo en sí mismo sino un medio para controlar el desarrollo de un sistema de software.

5.4.1. Métricas sobre el producto

Las métricas sobre el producto están orientadas a estimar las características del mismo antes de su desarrollo. Estas estimaciones se basan en el conocimiento que los desarrolladores adquieren a partir de datos obtenidos de proyectos anteriores.

A) Tamaño estimado del código.

La forma más obvia y la que se ha utilizado históricamente antes para estimar el tamaño es contar el número de líneas de código. Con ciertas normas para determinar qué es lo que se cuenta (líneas de comentario, código incluido, etc.) y siempre referido a un lenguaje concreto, lo que los valores nos dan es un valor para, comparando con otros casos, poder estimar el esfuerzo necesario en futuros desarrollos.

Los resultados obtenidos (estimaciones y valores reales) alimentan la base de datos históricos que es el fundamento para posteriores estimaciones.

Boehm desarrolló una técnica empleando el método Delphi para mejorar las estimaciones con múltiples opiniones de expertos. La idea de emplear el método Delphi es asegurar en dos o tres pasos de convergencia que las estimaciones son aceptadas por los expertos.

Ha sido muy criticada la tendencia en estimar el esfuerzo en base a las líneas de código. Una de las críticas se centra en que la complejidad del desarrollo no está directamente ligada al tamaño cuando nos movemos hacia el dominio de los sistemas concurrentes, distribuidos o de tiempo real. En ellos, las medidas deben referirse a estimaciones del mismo tipo de productos.

Otro problema surgido recientemente con la proliferación de generadores de código es que no importa demasiado el número de líneas de código generadas (excepto por problemas derivados del tamaño de la memoria para sistemas embebidos) sino el número de líneas de especificación que las han generado, porque la complejidad del problema de mantenimiento depende de ello.

B) Complejidad estimada.

Con el fin de superar el problema de las estimaciones del tamaño de código, se ha prestado recientemente atención a medidas de complejidad no basadas en estimaciones de número de líneas.

Albrecht definió en 1979 un método conocido como de **puntos de función** que está teniendo cada vez más aceptación. Su método se basa en el empleo de factores normalizados para juzgar la importancia relativa de varios requisitos funcionales.

Parte de cinco funciones básicas que suelen aparecer en muchos sistemas:

1) Entradas. Pantallas o formatos empleados para introducir datos a un programa.

2) Salidas. Pantallas o informes empleados para utilizarlos con otros programas o para lectura directa.

3) Consultas. Mecanismos para pedir ayuda o dar órdenes de ejecución.

4) Ficheros de datos. Conjuntos lógicos de información empleados por una aplicación (ya sean tablas en memoria como ficheros de disco) junto con los procedimientos de acceso a los mismos.

5) Interfaces. Ficheros compartidos con otras aplicaciones.

La idea básica del método consiste en definir unas estimaciones de complejidad para cada una de estas funciones (en forma de pesos relativos) y estimar, dadas las especificaciones del sistema, cuántos elementos de cada tipo van a ser necesarios.

El problema con los puntos de función es que no son realmente medidas sino valoraciones subjetivas y no tienen en cuenta diferencias en la implementación (al fin y al cabo, el esfuerzo del desarrollo depende también del lenguaje utilizado o del dominio de aplicación y eso no se tiene en cuenta). De nuevo, la comparación con sistemas similares permite «calibrar» las decisiones tomadas.

C) Robustez.

Por **robustez** de un programa se entiende la ausencia de fallos en su ejecución con diferentes datos de entrada durante intervalos de tiempo predeterminados.

La robustez de un programa está ligada a la aparición de problemas durante su ejecución. Generalmente, el número

de fallos encontrados durante la fase de prueba y, posteriormente, durante el mantenimiento del sistema constituye una medida de la calidad del producto de software e, indirectamente, de la calidad del proceso de desarrollo.

La importancia de conocer el número de fallos encontrados en un intervalo de tiempo no reside únicamente en obtener un valor global de la calidad del producto sino en los beneficios derivados de su análisis.

Las medidas estadísticas de fiabilidad (tiempo medio entre fallos encontrados durante la ejecución, reducción del número de recopilaciones necesarias, etc.) sirven para alimentar el proceso de desarrollo.

5.4.2. Métricas sobre el proceso

Las métricas mencionadas en la sección anterior estaban orientadas a conocer la complejidad del producto (con algún valor indirecto como el tamaño) para poder estimar los recursos necesarios para su realización. Hemos mencionado también que, según se vayan acumulando datos y se analicen estadísticamente, las estimaciones serán cada vez mejores. Esto nos servirá para planificar mejor futuros desarrollos.

Existen otros tipos de datos que se pueden tomar durante el desarrollo de un producto de software y que no están ligados al producto sino a los procesos implicados. El análisis de cómo estos procesos se realizan a partir de medidas tomadas en el desarrollo es la base para su ulterior mejora.

Algunos de los elementos a medir son:

A) Distribución del esfuerzo en cada una de las fases con objeto de poder estimar los recursos necesarios. Obsérvese que esta medida es complementaria a las de tamaño mencionadas

anteriormente; aquella nos permitía conocer los recursos globales necesarios; de lo que se trata aquí es de obtener medidas reales y extrapolarlas a futuros proyectos.

B) Productividad medida en número de líneas de código documentadas que es capaz de producir una persona en una unidad de tiempo. A título orientativo, podemos decir que los valores típicos de productividad por persona (empleando tecnologías de desarrollo convencional) están entre 30 y 50 líneas de código por día de trabajo.

5.5. Organización del desarrollo

A lo largo del proceso de desarrollo las actividades de gestión deben planificarse. No se puede hacer algo que no está planificado ni tampoco controlarlo. Una vez que disponemos de datos suficientes a partir de las medidas descritas en los apartados anteriores podemos planificar el desarrollo del producto.

5.5.1. Planificación del proceso de desarrollo

La planificación de un proyecto de software tiene como objetivo el establecimiento de los tiempos dedicados a cada fase del desarrollo y sus actividades así como a los recursos necesarios (humanos y materiales) para cada una de ellas.

Esta planificación parte de las estimaciones sobre la complejidad del sistema a realizar que hemos descrito en la sección anterior para dividir el trabajo en cada una de las tareas identificadas. Entre ellas:

a) Estimación del tiempo de desarrollo global y de cada tarea.

b) Estimación de los recursos necesarios para cada tarea.

Asimismo, la planificación tiene en cuenta las restricciones que impone las limitaciones de recursos, materiales, aprovisionamiento de software externo, etc. Dado que el control que se tiene sobre todos estos aspectos no es absoluto, se deberán definir planes de contingencia. Los aspectos de riesgos asociados y su efecto sobre la planificación serán tratados posteriormente.

Tomando como referencia un modelo de ciclo de vida en cascada, los tiempos dedicados a cada una de las fases han seguido la evolución representada en la Figura 35. En ella hemos representado también la típica curva de esfuerzo que aparece en la mayor parte de los sistemas. Es interesante comparar la curva A representativa de la situación hace diez años con la B que puede representar la situación actual.

La progresiva atención a las primeras fases del desarrollo derivada del análisis de la curva B, y el subsiguiente desplazamiento del esfuerzo

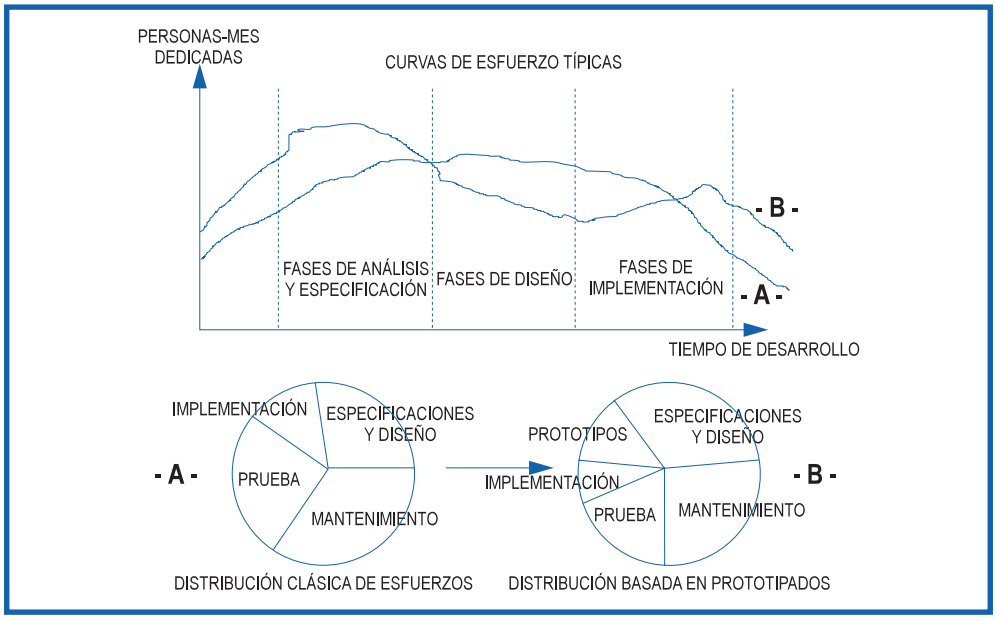


Figura 35 - ESFUERZOS EN LAS FASES DEL CICLO DE VIDA CONVENCIONAL -

hacia ellas ha modificado también los perfiles profesionales requeridos y el número de personas requeridas en cada uno de ellos.

La estimación de los tiempos dedicados a cada actividad (para un modelo de ciclo de vida determinado) debe tener en cuenta los siguientes factores:

a) Experiencias de productividad (líneas de código documentadas por persona y día) extraídas de proyectos anteriores de similar complejidad.

b) Experiencia de los componentes del equipo de trabajo en la tecnología de software que se utilizará en el desarrollo incluyendo la planificación de los aspectos de formación de las personas que intervengan.

c) Disponibilidad de los componentes software requeridos y su provisionamiento externo (paquetes software requeridos y extensiones de la infraestructura hardware de la máquina en la que se ejecuten.

d) Posibilidad de realización de actividades concurrentes de desarrollo. En esta faceta se incluyen aspectos tan importantes como la subcontratación externa de diversos componentes.

e) Mecanismos de control de calidad que se establezcan.

La Figura 36 reproduce un típico esquema de planificación de actividades tomando como referencia el modelo en cascada de la Agencia Espacial Europea (ESA). En él podemos ver el secuenciamiento de las fases, los hitos fundamentales establecidos, los productos generados, los mecanismos de revisión y los esfuerzos asignados a cada una de las fases (representada en forma del área asociada a cada una de las actividades).

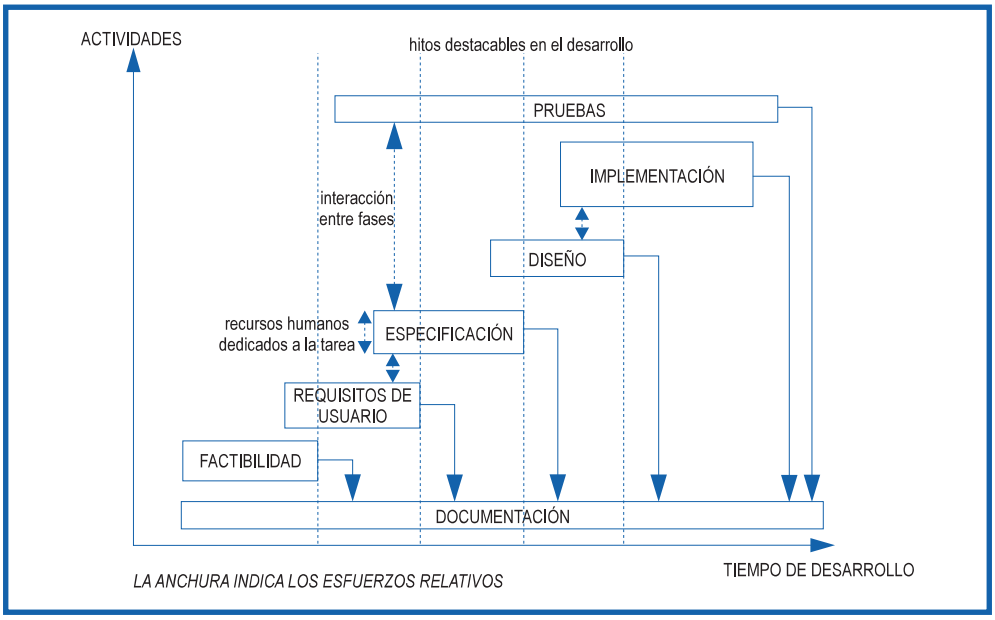


Figura 36 - ESQUEMA TÍPICO DE PLANIFICACIÓN DE UN PROYECTO DE SOFTWARE -

5.5.2. Gestión de riesgos

Toda actividad en el desarrollo de un proyecto de software conlleva riesgos. Estos riesgos, caso de que aparezcan efectivamente y produzcan sus efectos, pueden tener un fuerte impacto en la planificación «ideal» efectuada y poner incluso en peligro el desarrollo del proyecto en su conjunto.

Todos los gestores de proyectos de software han asumido la existencia de riesgos y han intentado que sus efectos sean los menores posibles. Ha sido en los últimos años cuando la gestión de riesgos ha pasado a ser una parte fundamental de la gestión de proyectos. En estos momentos, existen métodos de gestión orientados al control de riesgos.

Por gestión de riesgos se entiende el proceso de identificar riesgos y evaluar su probabilidad y potencial impacto y planificar el

proyecto a partir de ello. Esto incluye el desarrollo de estrategias adecuadas para mitigar su impacto, así como la asignación de recursos para ello, la creación de informes para la conocimiento del estado de los mismos y su seguimiento hasta que las consecuencias se hayan resuelto totalmente.

La realización de estas actividades dentro del ciclo de vida del desarrollo de un producto está generando una nueva forma de abordar los desarrollos de sistemas complejos. Antes de describir cada una de ellas vamos a definir los conceptos básicos que necesitamos.

Por **riesgo** se entiende cualquier evento que puede afectar al desarrollo de un producto de software.

El **impacto** o **exposición** a un riesgo se define como una función de la probabilidad de que ocurra el evento (indeseado) multiplicado por su impacto sobre el desarrollo del producto.

El impacto puede cuantificarse en pérdidas o ganancias asociadas en tiempo, dinero, recursos infrautilizados, reducción o expansión de la cuota de mercado, etc. Obviamente, no todas estos efectos aparecen con un riesgo concreto. Generalmente, se consideran riesgos cuando están asociados a impactos negativos pero los riesgos de cualquier signo afectan a la planificación del proyecto y todos ellos deben considerarse.

Las acciones asociadas a la gestión de riesgos son las siguientes:

A) Análisis de riesgos.

El análisis de riesgos se realiza durante las fases de planificación del producto de software con el objetivo de conocer a priori qué acontecimientos pueden poner en peligro el desarrollo. Consta de dos etapas:

1) **Identificación y documentación de riesgos**. Existen muchas causas que pueden dar origen a un riesgo. Unas externas al propio desarrollo; otras, derivadas del mismo.

   Entre los riesgos debidos a causas externas podemos citar:

   * **Político-económicos**. Difícilmente predecibles, suelen estar ligados a riesgos financieros del proyecto o a inestabilidades políticas que comprometan el desarrollo en proyectos plurianuales y multinacionales.

   * **Mercado**. Entre ellos aparecen limitaciones para el uso de algunas tecnologías o de comercialización de productos. También pueden considerase de este tipo riesgos derivados de limitaciones en la importación/ exportación de componentes en determinados mercados.

   * **Organización**. Formas de organizar los equipos de trabajo en función de los perfiles requeridos con objeto de minimizar los efectos de algunos acontecimientos.

   * **Técnicos**. Aparecen debido a las diferentes componentes de la tecnología de software (generalmente ligados a su inmadurez o falta de adaptación al sistema a desarrollar).

2) **Cuantificación de riesgos**. Cuantificar un riesgo consiste en determinar los valores de la probabilidad de aparición y del impacto (por tanto, también de la exposición) a lo largo del tiempo de desarrollo del producto. Estos datos definen el perfil del riesgo.

   La Figura 37 muestra posibles tipos de riesgos. Existen riesgos cuyo impacto se mantiene constante a lo largo del desarrollo, otros cuyo impacto aumenta, o disminuye

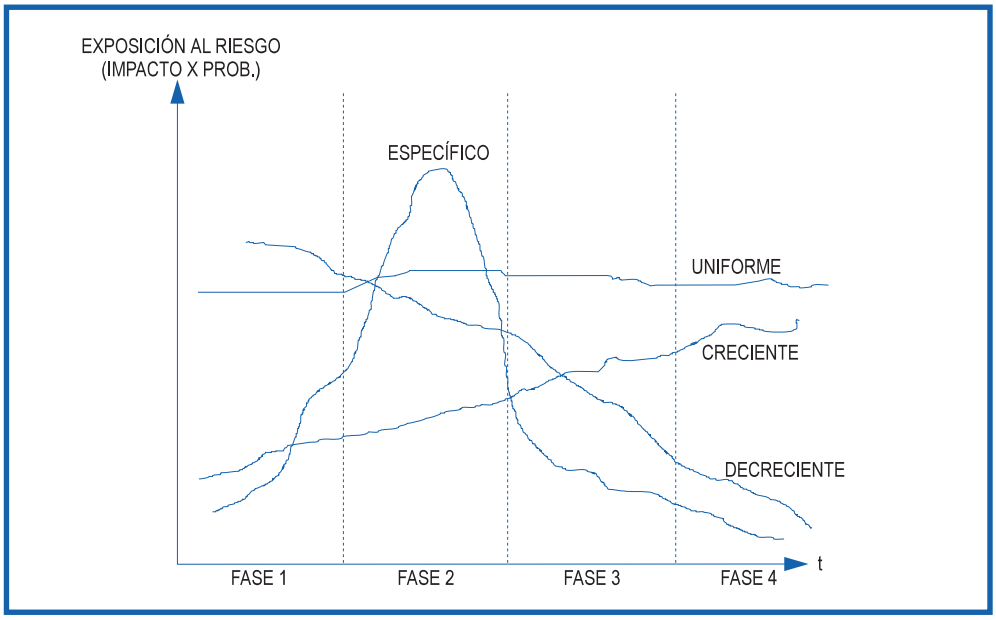


Figura 37 - PERFILES DE RIESGOS -

y otros, finalmente, cuyo impacto está localizado en el tiempo asociado a una actividad del desarrollo concreta.

3) **Modelado de riesgos**. Esta actividad pretende establecer prioridades entre los riesgos identificados que tomen en cuenta la cuantificación efectuada y el contexto del proyecto en el que aparecen

B) Resolución de riesgos.

Las actividades que genéricamente hemos denominado de análisis de riesgos nos permiten conocer, cuantificar y priorizar los posibles riesgos. Eso ha sido útil para mejorar el proceso de planificación temporal y la asignación de recursos. No obstante, el desarrollo de un proyecto es un proceso dinámico en el que la planificación inicial debe contrastarse continuamente con el desarrollo real.

La resolución de riesgos intenta eliminar o reducir la exposición al riesgo actuando ya sea sobre la probabilidad de que el riesgo se presente como sobre el impacto que éste tiene en el caso de que así suceda.

Las actividades incluidas son las siguientes:

1) **Mitigación de riesgos**. Por mitigación de riesgos se entiende las acciones requeridas para reducir o eliminar el impacto potencial de los riesgos en un proyecto. La planificación de la mitigación de riesgos debe seguir a las actividades de análisis de riesgos mencionadas anteriormente.

   Para cada riesgo priorizado debemos evaluar alternativas y seleccionar la mejor. Gran parte de estas acciones se pueden realizar en la fase de negociación antes de que el proyecto real comience.

2) **Monitorización de riesgos**. La única forma de mejorar el proceso de gestión de riesgos (y por ello mejorar la gestión del proyecto) es mantener una base de datos que pueda irse enriqueciendo dinámicamente con la experiencia derivada de la realización del proyecto.

La importancia de la gestión de proyectos se ha manifestado en la aparición de técnicas de gestión y planificación específicas. La Figura 38 representa esquemáticamente los procedimientos implicados [22].

5.5.3. Control de recursos humanos

Cualquier gestor debe asignar sus recursos humanos a las tareas a realizar intentando optimizar sus conocimientos y perfiles psicológicos

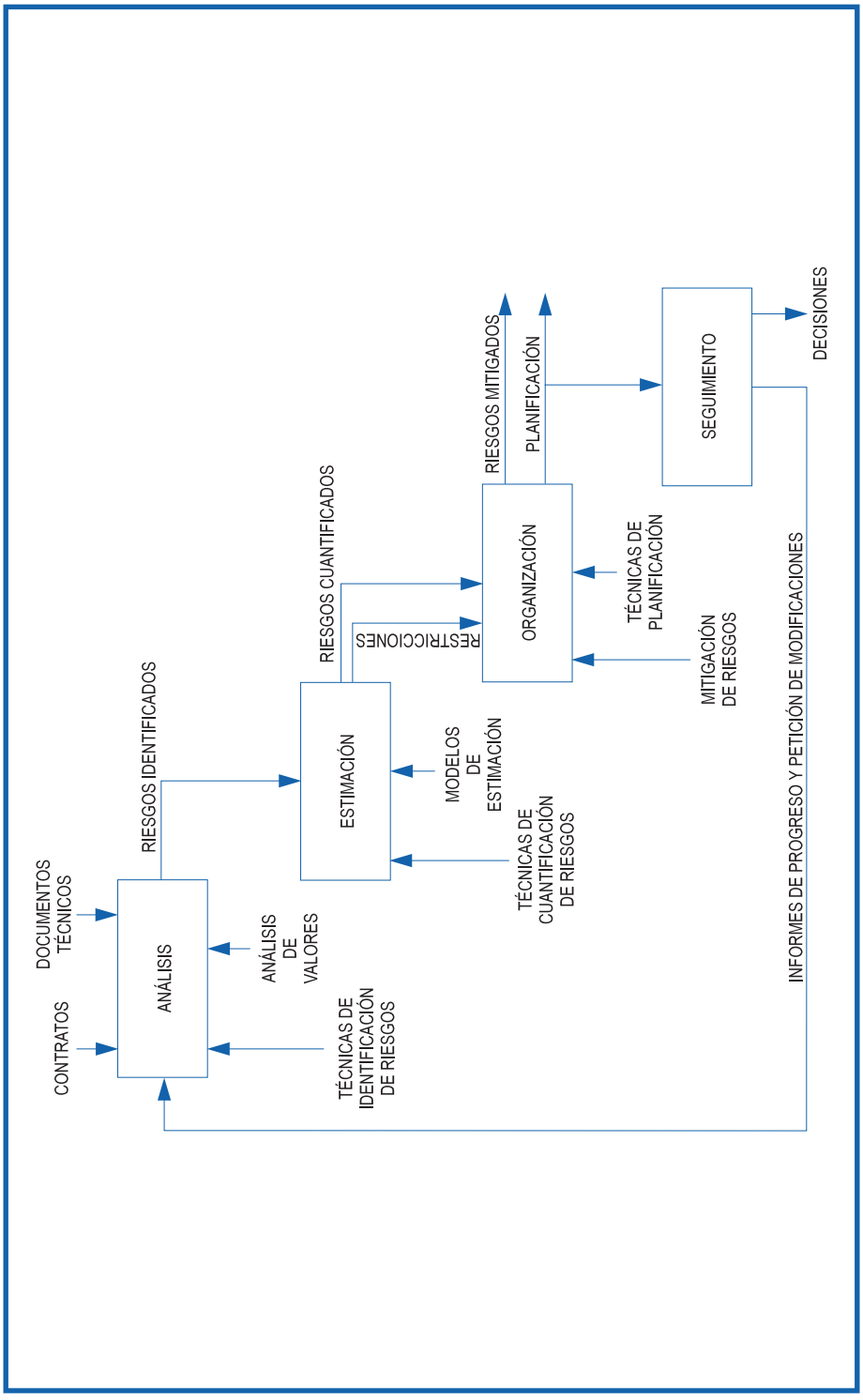


Figura 38 - PROCEDIMIENTOS DE GESTIÓN PARA CONTROL DE RIESGOS -

a los perfiles técnicos adecuados a las necesidades del proyecto. Si bien las estimaciones de recursos necesarios nos pueden dar una indicación de cuantas personas son necesarias, no nos dice quienes ni cuando deben formar parte del proyecto.

Los perfiles y el número de personas necesarias dependerán del tamaño del proyecto (si el proyecto es muy grande deberemos incrementar los gestores), del tipo de modelo de ciclo de vida seleccionado y del tipo de sistema. Una vez establecidos los perfiles que nos parecen necesarios, lo que nos interesa analizar en este Capítulo es la forma en la que estos recursos se gestionan en el desarrollo de un producto concreto.

La Figura 39 representa esquemáticamente la estructura de un grupo de trabajo y la asignación de personas con perfiles técnicos concretos. En la figura se ha representado un equipo humano reducido para el caso de un proyecto mediano. En ella podemos ver cómo una

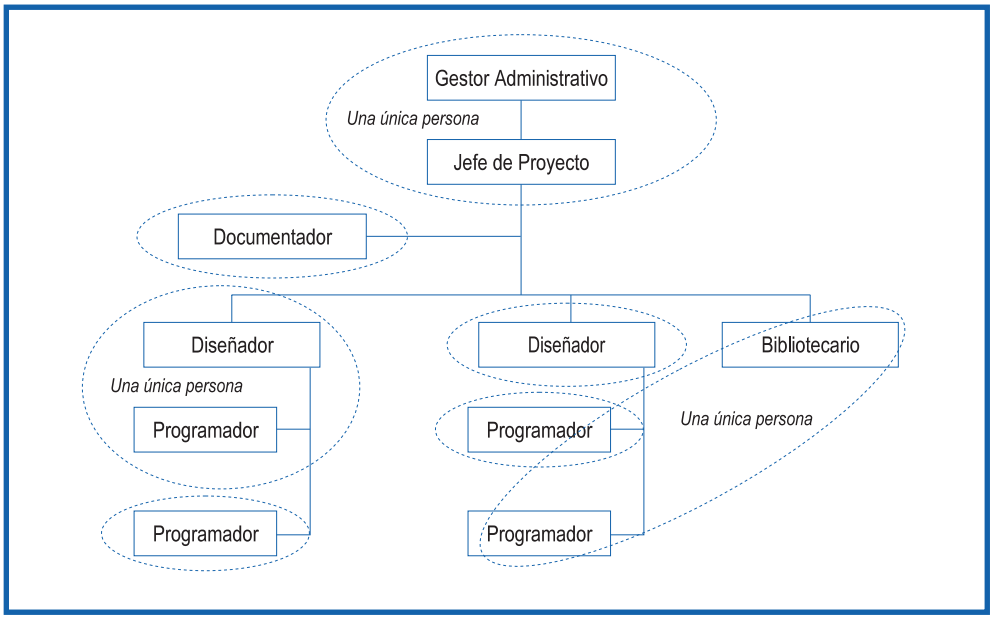


Figura 39 - ESTRUCTURA DE UN PROYECTO MEDIANO -

misma persona puede tener dos perfiles técnicos simultáneamente, o como de un perfil técnico se requieren varias personas.

La gestión de recursos humanos se basa en las siguientes premisas:

a) Selección del equipo humano en función no sólo de los conocimientos técnicos requeridos sino de su capacidad para formar un equipo conjuntado durante el desarrollo del proyecto.

b) Adquisición por parte del equipo humano de los conocimientos necesarios para llevar a cabo las funciones requeridas. Generalmente, este proceso implica el entrenamiento específico sobre las técnicas, métodos o herramientas que hayan sido seleccionadas para su uso en el proyecto.

c) Asignación de los recursos humanos a las diferentes tareas en función de las estimaciones de recursos necesarios procurando mantener la curva de esfuerzo más homogénea posible (picos de esfuerzo muy puntuales son difíciles de gestionar). Estas asignaciones se suelen realizar en unidades de personas-mes o personas-año lo que constituye la base para la estimación del presupuesto del proyecto.

d) Análisis, y mitigación en su caso, de los riesgos derivados de los componentes del equipo de trabajo a lo largo del tiempo. Como ejemplo de estos riesgos podemos mencionar el abandono del proyecto por alguna persona clave y la previsión de sustitución de la misma por otra persona.

e) Reasignaciones dinámicas de actividades a personas en función de la evolución del proyecto.

Estas tareas conllevan por parte del gestor una formación no exclusivamente técnica que permita asignar las actividades a aquellas

personas que estén más capacitadas para ello. Reconocer la labor de liderazgo es fundamental en el jefe del proyecto pero menos importante en el caso del programador; en este último es necesario, además de los conocimientos técnicos requeridos, poseer capacidad de trabajo en equipo y cuidado en los detalles. Casi todas las empresas de desarrollo de software poseen un equipo de psicólogos para facilitar la constitución de los equipos de trabajo.

Debemos también reconocer que, en muchos proyectos en los que participan varias entidades existen otras restricciones. El director técnico del proyecto o el responsable del mismo no tienen capacidad para determinar los componentes del equipo de trabajo. Suelen existir repartos de las actividades por otras muchas razones y su tarea es fundamentalmente de coordinación.

5.6. Gestión de la evolución del producto

Si recordamos los modelos de ciclo de vida presentados en el Capítulo 2, vemos que la entrega del producto al usuario no implica olvidarse de él. Muy al contrario, todo producto software pasa por procesos de cambio o evolución con el fin de adaptarlo a un entorno cambiante.

Con el fin de entender cuales son las actividades que se realizan en la fase de mantenimiento supongamos que existe una petición explícita de cambio sobre el producto (ya sea de un usuario externo o internamente desde los diseñadores). A partir de ella, se realizan una serie de actividades de mantenimiento que han sido esquemáticamente representadas en la Figura 40.

A) Fase de identificación del problema.

Esta fase comienza cuando el usuario, al detectar un problema con el sistema de software que está utilizando, genera

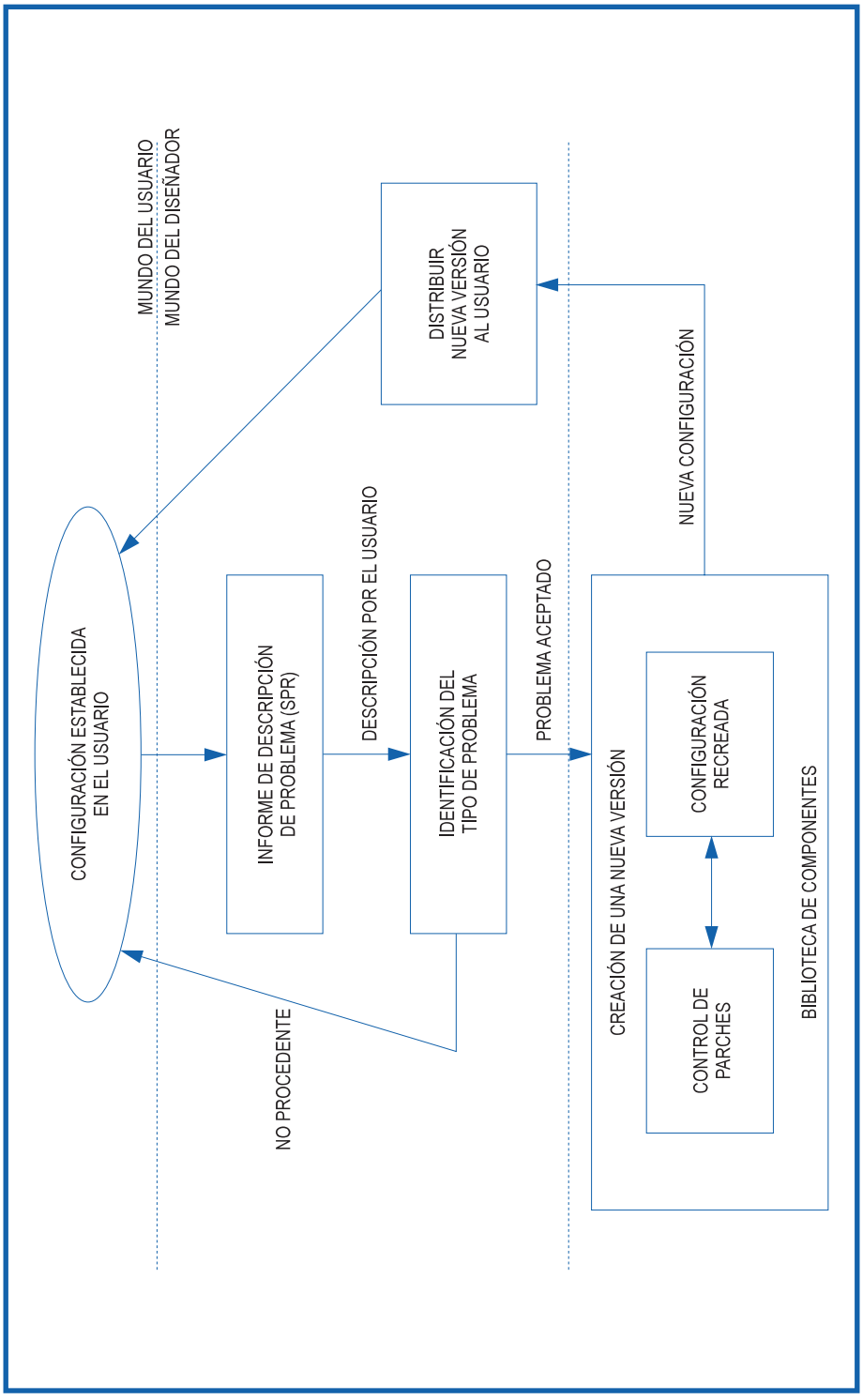


Figura 40 - FASES DE MANTENIMIENTO -

un informe del problema observado o de la modificación requerida que se entrega a los proveedores del producto. Este informe (generalmente empleando un formato normalizado) incluye información sobre la configuración software empleada por el usuario para que el proveedor pueda reproducirla en su instalación, una descripción en términos que el usuario desee y la documentación adicional que considere conveniente.

Seguidamente, se identifica el tipo de problema o modificación requerida (determinando si es procedente su resolución) y se bifurca hacia los responsables del mismo con una estimación de los recursos necesarios.

Como parte de este proceso se deberá determinar si el problema debe ser considerado desde el punto de vista legal (por ejemplo, si el problema está contemplado dentro del contrato de mantenimiento caso de que éste exista).

B) Fase de resolución del problema.

Una vez que el problema llega a los responsables, éstos tienen que recrear la versión del sistema de software y la infraestructura de ejecución que posee el usuario y realizar las correcciones adecuadas.

En muchos casos, la solución pasa por la utilización de documentación de diseño que debe extraerse de la base de datos del proyecto (recuperando la versión adecuada). En los casos en los que la documentación de diseño no existe (sólo está documentada la implementación), típico en sistemas desarrollados hace mucho tiempo y aún en vigor, puede ser necesario emplear técnicas de ingeniería inversa de software para obtener los diagramas de diseño a partir del código.

C) Fase de generación de una nueva versión.

Tras efectuar las acciones necesarias en función del tipo de modificación requerida, es necesario crear una nueva versión, modificar todos los elementos de la configuración que sea necesario y entregarla de nuevo al usuario. Para ello se emplean las herramientas de control de versiones y configuraciones mencionadas anteriormente.

5.7. Normativa en la ingeniería de sistemas de software

Todas las técnicas y actividades presentadas en este Capítulo pueden estar contenidas en normas aprobadas oficialmente por la organización que desarrolla el sistema de software. Adquieren, por tanto, un aspecto oficial en esa organización de la que se deriva la necesidad de su cumplimiento.

Esta situación conlleva, sin embargo, dos problemas: la necesidad de que la organización dedique esfuerzo a la creación y mantenimiento de sus normas (difícil en una pequeña empresa y, en todo caso, largo y costoso) y, más importante, dificulta la cooperación entre diferentes organizaciones al existir normas propietarias no compatibles entre sí en una época en el que el desarrollo de sistemas de software en entornos multiproveedor es cada vez más común.

Con el fin de evitar este doble problema y contribuir también a mejorar los procesos de desarrollo, diversas organizaciones internacionales han propuesto normas (estándares) de ingeniería de sistemas de software (incluyendo glosarios con la definición de los términos más importantes).

Las normas de ANSI/IEEE son generales y sirven de marco de referencia para la elaboración de normas concretas propias. Los

estándares de gestión de la ESA son más concretos (aunque derivados de los del IEEE) y pretenden servir para el proceso de control y evaluación de los trabajos encargados en el programa de desarrollo de la Agencia Espacial Europea.

En otro sentido, existe una tendencia hacia la certificación de la calidad del proceso de desarrollo seguido por una organización mediante normas descritas por organismos como ISO (serie de normas ISO 9001) o el Instituto de Ingeniería Software en EEUU (SEI) con el modelo de madurez (CMM).

El nivel de uso en la práctica de estas normas internacionales no es muy alto. Casi todas las organizaciones han tomado un modelo genérico y, sobre él, han realizado las adaptaciones necesarias para que sea factible su uso en la organización de que se trate.

5.8. Resumen

Este Capítulo concentra los aspectos de gestión del desarrollo de software de forma complementaria a los aspectos tecnológicos descritos en los Capítulos anteriores (3 y 4).

Los aspectos de gestión están fundamentalmente ligados a los procesos necesarios para el desarrollo de cualquier sistema de software dado un modelo de ciclo de vida en una organización determinada. Lo que con ello se busca, por tanto, es disponer de unos procedimientos asentados en la experiencia y adaptados al tipo de organización, tamaño del proyecto y tipo de sistema que permitan controlar el desarrollo por un grupo de trabajo.

A lo largo de este Capítulo se han presentado diversas técnicas empleadas para la gestión del desarrollo cuya importancia debe evaluarse de forma global. Individualmente, afectan a aspectos muy concretos, pero es su estrecha relación la que permite controlar el proceso de desarrollo.

Como ejemplo, la planificación del desarrollo tiene en cuenta los recursos humanos necesarios; pero éstos son obtenidos a partir de estimaciones en las que se emplean métricas del producto. Durante la prueba del sistema se asegura la calidad requerida del producto empleando métricas sobre aspectos concretos del producto y se actualiza la planificación del producto. Finalmente, todos los componentes del producto son controlados mediante los procedimientos de control de versiones y configuraciones.

La gestión de recursos humanos está en la base de todo lo anterior. El desarrollo de sistemas de software es una actividad intensiva en recursos humanos; de poco vale disponer de técnicas sofisticadas si el proyecto no aprovecha los recursos humanos disponibles. Por ello, la gestión de un proyecto de desarrollo de software es, ante todo, una gestión de los recursos humanos implicados; el resto de las técnicas está subordinado a ello. Los gestores también requieren una formación especializada en técnicas concretas y, sobre todo, una especial formación para la conducción de grupos humanos.

6

La mejora del proceso y la adopción de nuevas tecnologías de software

6.1. Introducción

La mayor parte de los sistemas de software no cubren las expectativas planteadas, no satisfacen totalmente los requisitos de los usuarios, necesitan mucho más tiempo para finalizar su desarrollo que lo planeado inicialmente y su coste es claramente superior.

Esta situación se deriva de dos grupos fundamentales de problemas:

1) Una deficiente gestión del proceso de desarrollo.

2) Una tecnología de software con importantes limitaciones.

Con respecto al primero de los problemas, en el Capítulo 5 hemos pasado revista a las técnicas de gestión de proyectos de software y analizado la importancia que tienen en el proceso de desarrollo de sistemas complejos. Esta importancia queda reflejada en que gran parte de los problemas de gestión del desarrollo se concentran en una insuficiente formalización de los procesos requeridos. No es extraño por ello que exista un gran interés en mejorar las prácticas de gestión dentro de la denominada **mejora del proceso**. Generalmente, este proceso de mejora es incremental y paulatino exigiendo una actitud definida de observación, análisis del problema e implantación de nuevos procesos.

El segundo elemento que debemos tener en cuenta es la tecnología de software utilizada en el desarrollo. De la multitud de tecnologías de software disponibles actualmente que han sido desarrolladas en los últimos años, sólo una pequeña fracción ha llegado a utilizarse ampliamente en la industria. La mayor parte de ellas no logra sobrepasar la fase de creación

y uso en un entorno restringido a pesar de sus potenciales ventajas; dicho de otra forma, no maduran completamente. En otros casos, si bien la tecnología de software está disponible, es la situación particular en una organización concreta la que ralentiza o dificulta su proceso de adopción.

Las dificultades mencionadas se están convirtiendo en un cuello de botella para la innovación tecnológica de las empresas de desarrollo de software. Incluso en aquellos casos en los que, finalmente, penetra en el sector productivo, lo hace a un ritmo muy lento condicionando la competitividad de la industria de sistemas de software.

La transferencia de tecnología desde los proveedores hasta los receptores de la misma y la adopción de ésta por parte de los receptores (dos caras de una misma moneda) ha sido objeto de un gran interés en los últimos años [23]. La aparición de técnicas específicas y, sobre todo, de una mejor comprensión del proceso de desarrollo de software puede ayudar a acelerar el proceso de innovación y a permitir una fuerte mejora de la productividad y calidad en el desarrollo de software. El cambio de tecnología puede ser rápido y con un impacto apreciable en la organización receptora.

En este Capítulo, revisaremos, en primer lugar, el contexto de innovación del proceso de desarrollo de software en la industria para, posteriormente, describir algunos modelos de adopción de nuevas tecnologías y las acciones encaminadas a una mejora de los procesos empleados. Finalmente, presentaremos algunas tendencias en el desarrollo de la ingeniería de sistemas de software para los próximos años tanto en tecnologías como en procesos.

6.2. La mejora del proceso de desarrollo del software

Cualquier proceso de innovación implica la modificación (en algunos casos de forma traumática) de los procesos de desarrollo empleados en la industria así como de la mentalidad de las personas que intervienen

en ese proceso. No es extraño, por tanto, que existan resistencias al cambio que explican la inercia en la utilización de nuevas técnicas.

La forma en la que una empresa asume el proceso de innovación software está ligada a la cultura empresarial existente en la misma y el estado en el que se encuentra la tecnología que se adopta. Podemos hablar de precursores, adaptadores tempranos, mayoría temprana, mayoría tardía y rezagados en función del momento en el que estas organizaciones incorporan una nueva tecnología. La Figura 41 describe esta situación en la que la mayor parte de las empresas pueden considerarse pertenecientes a los grupos de mayoría temprana o tardía. Generalmente, las empresas innovadoras (precursoras o adaptadoras tempranas) aceptan un riesgo mayor al utilizar tecnologías menos consolidadas.

Vamos a analizar la situación de la innovación en una empresa concreta desde dos perspectivas complementarias:

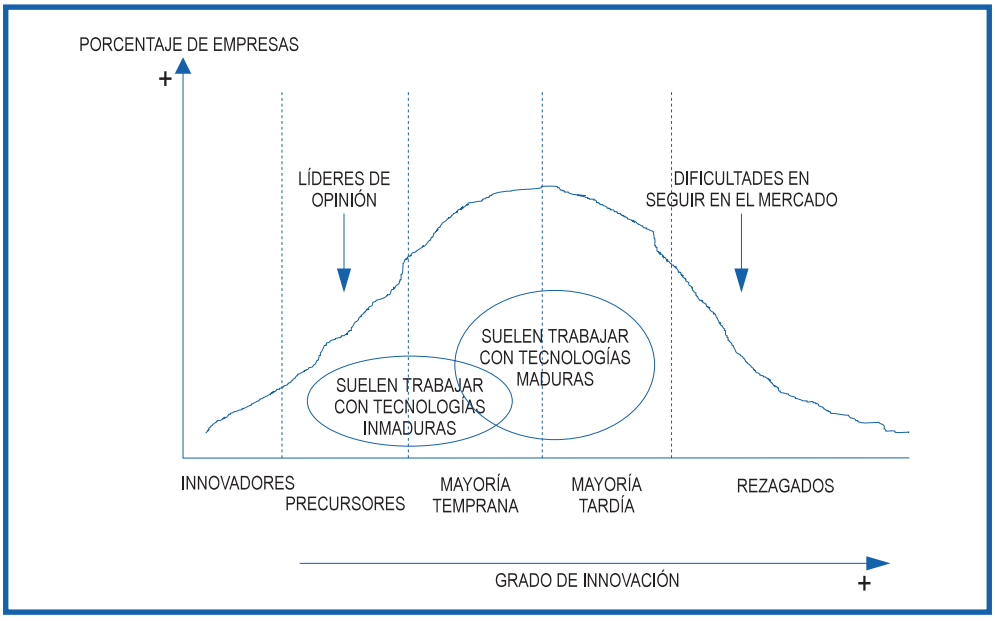


Figura 41 - LAS EMPRESAS FRENTE A LA INNOVACIÓN -

1) el nivel de madurez de su desarrollo de software y

2) el perfil de innovación de la empresa.

Ambas nos permiten obtener una visión global de la organización y definir el proceso de cambio más conveniente.

Como es bien conocido en la bibliografía de la innovación tecnológica [22], el énfasis en la mejora vía la optimización de los procesos, surge cuando la tecnología es estable. Es difícil para una organización asumir procedimientos de mejora de los procesos cuando la tecnología es inestable.

En los últimos años ésta ha sido la situación en la ingeniería de sistemas de software. Suponiendo estable un modelo de ciclo de vida en cascada, lenguajes de programación estructurados y un método de desarrollo concreto (por ejemplo, estructurado), el elemento que puede mejorar la calidad del producto es disponer de procesos de desarrollo bien definidos.

Si atendemos al tipo de innovación ligada a la ingeniería de sistemas de software, ésta siempre responde a la idea de sustitución de una tecnología preexistente. Globalmente considerada, no tenemos un nuevo producto, ni un nuevo mercado (eso dependerá de la penetración de los productos que se hagan con la tecnología), el objetivo es mejorar o facilitar la forma en la que se desarrolla un producto.

Hace unos años (1991) el SEI (Instituto de Ingeniería Software en EE.UU.) desarrolló un modelo denominado CMM, Modelo de Madurez («Capability Maturity Model») para conocer el grado en el que las organizaciones que desarrollaban sistemas de software se encontraban. Este modelo surgió a partir del trabajo que desde 1986 el SEI y MITRE (una empresa en EEUU) estaban realizando en la elaboración de un marco de mejora del proceso de desarrollo de software.

El modelo pretende establecer el grado de madurez que la organización posee en función de los procedimientos de gestión del desarrollo del software utilizados, y, derivar en función de ello un incremento de la confianza de sus clientes (inicialmente el Departamento de Defensa) en que el desarrollo de los productos encargados llegaría a buen puerto.

Con ello, se pretendía establecer objetivamente el grado de madurez de una organización en el desarrollo de software. Las organizaciones **inmaduras** son aquellas en las que los procesos software son generalmente improvisados por desarrolladores y gestores durante el proyecto. Las organizaciones **maduras** son aquellas en las que existe una capacidad al nivel global de la organización para gestionar el desarrollo y conseguir que las actividades se realicen de acuerdo al proceso planificado (di lo que vas a hacer y haz lo que dices que vas a hacer). En ellas existen mecanismos de evaluación y propuestas de mejora de forma continua.

El modelo posee cinco niveles que seguidamente presentamos. Es importante destacar, no obstante, que la mayor parte de las organizaciones (alrededor del 90%) se encuentran en los dos primeros niveles indicando con ello la escasa solidez del proceso de desarrollo en la industria de desarrollo de software actual. Estos niveles son:

1) **Inicial**. En este nivel el proceso de desarrollo de software es «reinventado» cada vez que se inicia un nuevo proyecto. Si alguna vez se cumplen los plazos estimados se debe más al esfuerzo y buen-hacer de diseñadores concretos que a un control del proceso. El éxito no es repetible.

2) **Repetible**. Existen políticas preestablecidas para gestionar un proyecto de software y procedimientos para implementarlo basado en la experiencia de proyectos similares. Implica un control de versiones y configuraciones. El proceso seguido es repetible.

3) **Definido**. Existe una definición documentada del modelo de ciclo de vida que permite su reutilización en sucesivos proyectos. Todos y cada uno de los procesos de desarrollo están definidos en unas guías de uso que los proyectos deben seguir. Los gestores, además, pueden evaluar su seguimiento.

4) **Gestionado**. Existen procedimientos de gestión y control de riesgos en base a la experiencia de proyectos anteriores.

5) **Optimizado**. Es en este nivel en el que se piensa en el cambio de tecnología. Los datos analizados permiten mejorar los procesos y modificarlos según objetivos predefinidos. Las mejoras en la tecnología y el proceso se gestionan de forma rutinaria.

El paso desde uno de los niveles al siguiente se consigue mediante la incorporación de nuevos procesos (o mejora de los preexistentes) ligados a algunas áreas consideradas clave. La Figura 42 indica las áreas clave que deben ser alcanzadas en cada uno de los niveles para poder pasar al siguiente. Obviamente, cada nivel implica la aceptación de los procesos empleados en los niveles anteriores.

Si fijamos la atención en el nivel 2, vemos que los aspectos que aparecen han sido mencionados en el Capítulo 5 como parte de la gestión del proyecto. La existencia de niveles superiores indica, sin embargo, que con ello no es suficiente.

Estos movimientos son lentos. De algunas experiencias publicadas podemos decir que pasar del nivel 1 al 3 puede requerir entre tres y cuatro años. Lo importante, detrás del establecimiento de un programa de mejora del desarrollo de software no es conocer el nivel de la organización de forma aislada, o alcanzar uno predeterminado como un objetivo en sí mismo, sino la **cultura de mejora**

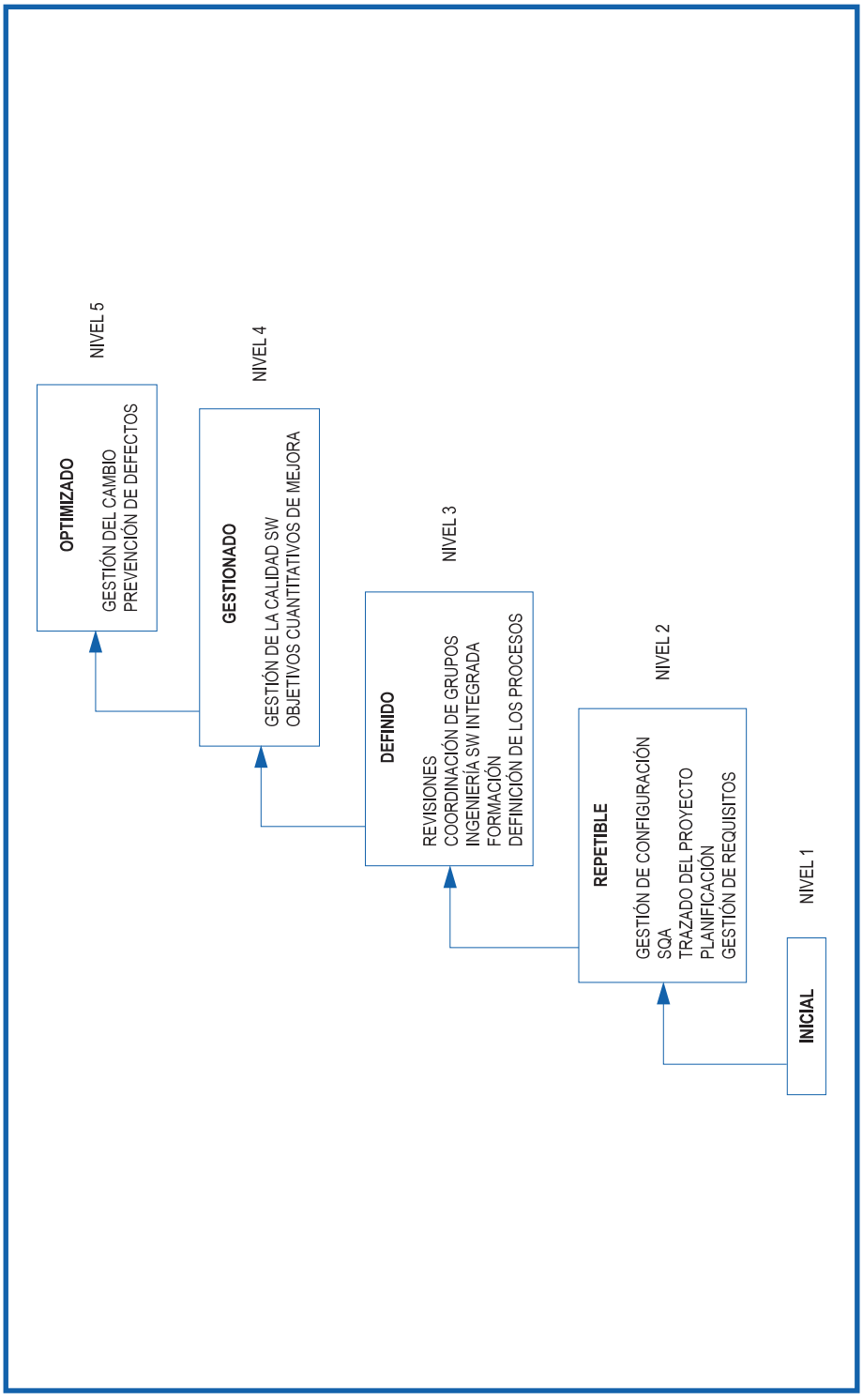


Figura 42 - ÁREAS BÁSICAS DE MEJORA DEL PROCESO SOFTWARE -

continua que impone en la organización. Ese cambio de mentalidad es, asimismo, un capital de la organización que irá influyendo en todas las unidades e individuos de la misma manera que lo hizo el control de calidad en la industria de fabricación hace años.

No es el CMM el único de los modelos de mejora del proceso de desarrollo de software existentes. De hecho, existen otros modelos, o combinación de los mismos, en esfuerzos en la misma línea apoyadas por diversas organizaciones.

En una línea ligeramente distinta se encuentran las normas de calidad ISO 9000. Concretamente, ISO 9001 (y los documentos asociados) están siendo empleados por muchas organizaciones como marco de referencia para indicar su madurez frente a potenciales clientes o para poder licitar en contratos públicos. Algunos informes indican que la certificación ISO sitúa a la organización que lo posea entre los niveles 2 y 3 del CMM.

Es interesante destacar desde el punto de vista de ingeniería de sistemas que el proceso de mejora del desarrollo de software afecta a la organización en su conjunto. El concepto de grado de madurez de los procesos de desarrollo, si bien ha surgido en el ámbito del desarrollo de software, es aplicable al desarrollo de cualquier tipo de sistemas. Todo sistema es desarrollado por una organización empleando una serie de procesos; su conocimiento preciso, su análisis bajo una óptica de mejora, y la puesta en marcha de modificaciones a los mismos es la base de un mejor desarrollo de sus productos.

6.3. Adopción de una tecnología de software

Si los procesos de mejora del proceso o las pequeñas optimizaciones que sobre la tecnología de software se pueden hacer como consecuencia de ello no son suficientes para mantener la

competitividad de la organización o el nivel de calidad de los productos software requeridos, debemos pensar seriamente en la necesidad de cambiar nuestra tecnología de software.

Adoptar una nueva tecnología implica que existe y que es adecuada para nuestros propósitos; aspectos que reflejan un proceso de selección y evaluación. Implícitamente también, estamos suponiendo que la tecnología es suficientemente madura para poder adoptarse por una organización comercial con riesgos controlados. Con ello queremos indicar que sus componentes (recuérdese el Capítulo 3) están suficientemente desarrolladas como para permitir su uso en productos críticos de la compañía.

La adopción de una tecnología también implica riesgos. Esto riesgos dependen de la tecnología, de la organización receptora y de la actitud de las personas involucradas.

Si atendemos al estado de la tecnología a adoptar, los modelos empleados para la adopción de tecnologías maduras e inmaduras son diferentes. Seguidamente, comentaremos brevemente cada uno de ellos.

6.3.1. Modelos para tecnologías maduras

Estos modelos suponen que los gestores de la organización, ya sea por presiones internas o externas, han decidido acometer un proceso de transferencia de una nueva tecnología en una unidad organizativa dada.

El modelo pasa por establecer un patrocinio claro desde la dirección de la organización que soporta todo el período de transición y la creación de un **grupo de transición** [22]. Este grupo es el encargado de poner en marcha las medidas adecuadas para planificar el proceso, evaluar la tecnología mediante el desarrollo de casos piloto

y, finalmente, en caso de que sea válida, tomar las medidas adecuadas para su difusión en la organización. La Figura 43 sugiere también que ese proceso puede implicar varios ciclos. En cada uno de los ciclos se ajusta la estructura organizativa de la empresa para optimizar el uso de la nueva tecnología.

Hemos representado dos elementos que interactúan en las actividades representadas: el grupo de transición empleando diversas técnicas y la existencia de un conjunto de herramientas de soporte en forma de entorno de ayuda a la innovación.

Un modelo como el descrito tiene el problema básico de que no permite una interacción entre los proveedores de la tecnología y los receptores de la misma con objeto de complementar ésta. Supone que la tecnología es suficientemente madura como para que la evaluación no se centre en ella sino en la adecuación de la misma a un entorno organizativo determinado. Para ello, los ciclos de adopción deben permitir una mejor realimentación entre proveedores y receptores.

En todo caso, sí implica cambios no despreciables en el entorno organizativo. Este es un punto de contacto con la reingeniería de procesos que será objeto de profunda atención en los próximos años.

6.3.2. Modelos para tecnologías inmaduras

Las tecnologías procedentes de los centros de investigación no pueden madurar en los mismos. Algunos de los componentes como los métodos para desarrollar grandes sistemas o las directrices industriales en un dominio de aplicación concreto, sólo pueden aparecer o madurar durante el proceso de transferencia controlada a un entorno industrial.

El modelo representado en la Figura 44 tiene en cuenta este hecho ofreciendo un marco en el que el grupo de transición (formado por proveedores y receptores) trabaja en varios ciclos transfiriendo

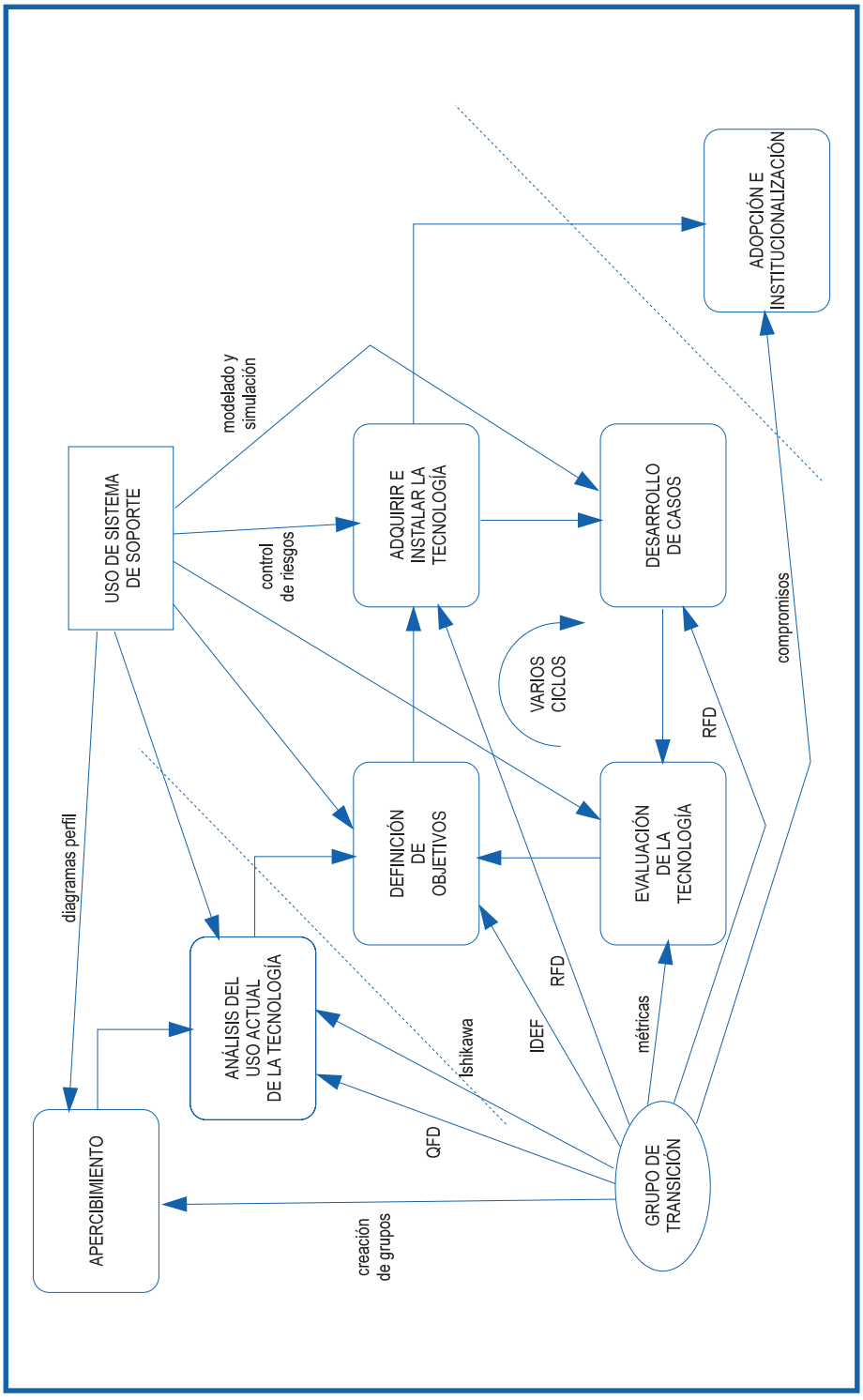


Figura 43 - MODELO PARA TECNOLOGÍAS MADURAS -

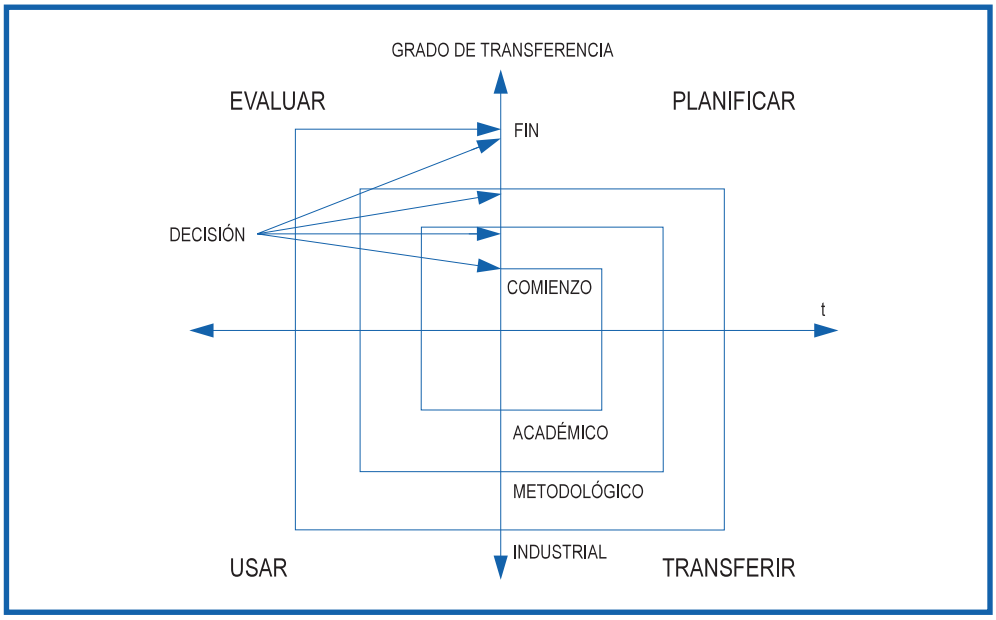


Figura 44 - MODELO PARA LA ADOPCIÓN DE TECNOLOGÍAS INMADURAS -

paulatinamente la tecnología. Inicialmente, componentes notacionales, algunas herramientas y formas de analizar el sistema en desarrollo para, finalmente, apoyados en casos de estudio, transferir el resto de los componentes.

En cada uno de los ciclos (en la figura se han representado tres) existen cuatro actividades básicas: **planificar** la introducción en ese ciclo de aquellos elementos necesarios, la **transferencia** de los mismos al entorno del receptor, su **uso** mediante el desarrollo de casos piloto controlados y finalmente la **evaluación** y la consiguiente toma de decisión.

Hemos considerado un primer ciclo dedicado a la adopción de las bases de la tecnología (académico), un segundo tipo orientado a los aspectos metodológicos de desarrollo de un sistema de software grande (metodológico) y un tercero para la institucionalización de la tecnología (industrial).

Este modelo también permite establecer puntos de compromiso al final de cada ciclo para que los gestores determinen si conviene continuar el proceso de adopción/transferencia, hibernar la decisión hasta que determinados componentes maduren o, sencillamente, pararlo. Con los ciclos está implícita una dimensión de coste de la misma forma en la que el meta-modelo en espiral propone para el desarrollo de sistemas de software (ver Capítulo 2).

6.3.3. Gestión de riesgos en la adopción de nuevas tecnologías

En el Capítulo 4 analizamos la gestión de riesgos en el desarrollo de un proyecto de desarrollo de software. Esta problemática también aparece en los períodos de transición desde una tecnología a otra.

Un proceso de transferencia de tecnología no siempre termina con éxito (entendiendo como tal la adopción completa de la tecnología seleccionada). Existen muchos factores de riesgo a lo largo del proceso que es necesario gestionar mediante métodos y técnicas concretas.

Por **riesgo** en transferencia tecnológica se entiende cualquier acontecimiento o decisión tomada que reduzca la probabilidad de éxito en la adopción. Existen muchos factores de riesgo que clasificamos en tres grupos:

1) Riesgos derivados de la tecnología. Generalmente derivados de la inmadurez de la tecnología al no adaptarse al tipo de sistemas de software a realizar, de las dificultades de su empleo o de la escalabilidad de la misma para soportar el tamaño de los sistemas.

2) Riesgos derivados de la estructura organizativa. En este caso, los riesgos están asociados a la implantación de los cambios organizativos necesarios para aprovechar la nueva tecnología.

3) Riesgos derivados del individuo y su interacción con el proceso de transición en forma de resistencia al cambio (falta de motivación o recompensa por el esfuerzo añadido de cambiar de modos de trabajo).

Actualmente, se están comenzando a adaptar las técnicas de gestión de riesgos empleadas en la gestión de proyectos al caso de transferencia de tecnología con el apoyo de herramientas adecuadas.

6.3.4. La formación requerida

El freno más importante y más difícil de superar en los procesos de innovación estriba en la capacidad de los equipos de desarrollo en «entender» los nuevos procesos que se quieren adoptar. Por entender quiero referirme a hacerlos suyos, a comprender por qué deben ser adoptados y no simplemente al conocimiento técnico para su uso. Para ello, deben sentirse parte del proceso de innovación y no únicamente los destinatarios pasivos del mismo.

Generalmente, la introducción de una nueva tecnología se acompaña de cursos de entrenamiento que atienden, fundamentalmente, a los aspectos técnicos de uso. Rara vez se ofrece algo más. Desgraciadamente, no es suficiente. La aparición de perfiles técnicos más horizontales como se ha expuesto en el Capítulo 2 requiere una formación integrada entre aspectos técnicos y de gestión con una visión global del proceso de desarrollo que favorezca la aceptación global del proceso de innovación.

Si las dificultades existentes han sido analizadas en países y organizaciones tan distintas, debe existir algún problema formativo de índole general. En este sentido, la formación del ingeniero de sistemas de software se produce puntualmente en el tiempo (durante unos años universitarios) con un enfoque fundamentalmente individualista seguido en el mejor de los casos de algunos cursos de formación continua y,

más comúnmente, de cursos de entrenamiento sobre las técnicas requeridas en el ejercicio profesional.

No obstante lo anterior, ya hemos visto a lo largo de esta monografía como la actividad de desarrollo es, ante todo, una actividad en equipo y, progresivamente más horizontal y menos parcelada en visiones muy limitadas sobre algunas actividades del desarrollo. Ello requiere una formación complementaria a la convencional que no está disponible actualmente y que no debe estar limitada en tiempo y espacio.

Generalmente, las ideas de evolución, de cambio continuo sobre los sistemas y tecnologías no son realmente entendidas. Debemos hacer un esfuerzo a que los procesos de cambio, su control y aprendizaje subsiguiente forman parte inherente de la actividad profesional. La formación del ingeniero de sistemas de software debe asentarse en ello y su aceptación, nos llevará a una consolidación de la ingeniería de sistemas de software como una disciplina real de ingeniería.

6.4. Resumen

En este Capítulo hemos ofrecido un marco de mejora de la ingeniería de sistemas de software desde una doble perspectiva: la mejora de los procesos y la mejora de la tecnología. Ambas perspectivas no son independientes; históricamente, han estado imbricados con mayor énfasis en unas etapas u otras a favor de cada uno de ellos.

La ingeniería de sistemas de software puede considerarse embebida en un avance simultáneo e interdependiente de la tecnología que permiten desarrollar los productos y los procesos que lo facilitan en una organización determinada.

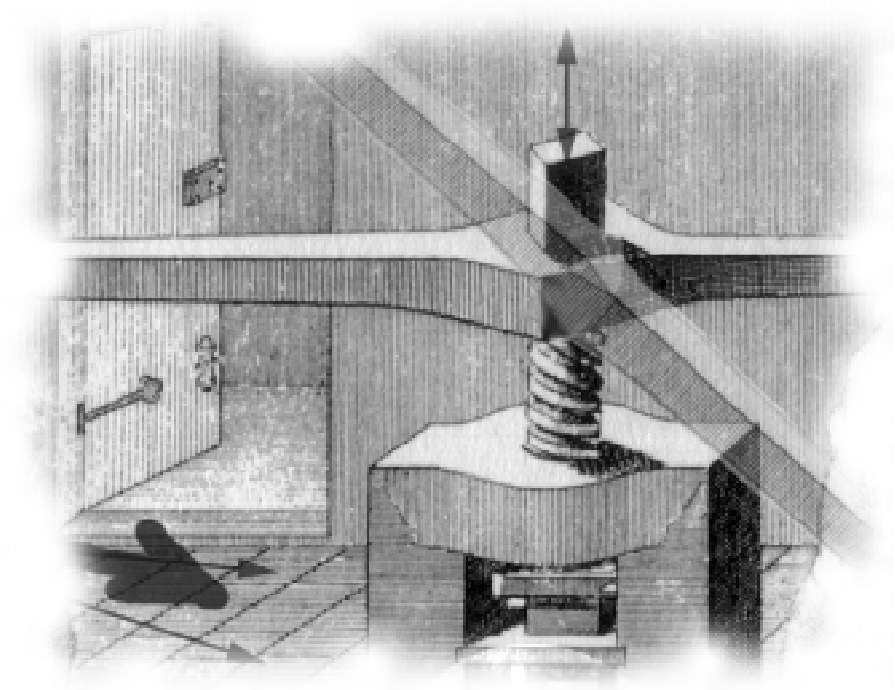
No existe, sin embargo, una acumulación de tecnologías inservibles. Muy al contrario, existen problemas técnicos no resueltos

que van a exigir mejores tecnologías de software y continuo ajuste y redefinición de los procesos implicados [24].

Además, la mejora del proceso de desarrollo de software no puede entenderse al margen de la mejora en el resto de las actividades de la organización. En muchos casos, los productos a desarrollar no son únicamente piezas de software; la calidad de los mismos está ligada a la calidad de los procesos y tecnología empleados para desarrollar cada uno de sus componentes y aquellos ligados a la integración del sistema final.

Si esto es así, lo es porque toda la organización en su conjunto debe asumir el reto de la calidad total. No se puede conseguir un nivel de calidad elevado en el desarrollo del software al margen del resto de los procesos empleados en la compañía. Desde esta perspectiva, la ingeniería de sistemas y todas las derivadas de ella como la de software tienen un horizonte común.

Referencias



[1] Blanchard, B.S., Ingeniería de Sistemas, Serie de monografías de ingeniería de sistemas, Isdefe, Madrid, 1995.

[2] Pressman, R.S., Ingeniería del Software: un enfoque práctico, 3ª edición, Mc Graw-Hill, 1994.

[3] Humphrey, W. S., A Discipline for Software Engineering, SEI Series in Software Engineering, Addison Wesley, 1995.

[4] Conger, S. The new Software Engineering,The Wadsworth Series in Management Information Systems, 1994.

[5] Guezzi, C., M. Jazayeri & D. Mandrioli, Fundamentals of Software Engineering, Prentice-Hall, 1991.

[6] Budde, R., K. Kautz, K. Kuhlenkamp, & H. Zullighoven, Prototyping, An Approach to Evolutionary System Development, Springer Verlag, 1992.

[7] Mazza, C., J. Farclough, B. Melton, D. de Pablo, A. Sheffer & R. Stevens, Software Engineering Standards, Prentice-Hall, 1995.

[8] Boehm, B.W., A Spiral Model of Software Development and Enhancement, IEEE Computer, págs. 61-72, mayo 1988.

[9] Boehm, B.W., Software Risk Management: Principles and Practices, IEEE Software, págs. 32-41, enero 1991.

[10] Burns, A. & Wellings, A., Real-Time Systems and their Programming Languages, Int. Computer Science Series, Addison Wesley, 1990 (2ª edición pendiente de publicación en 1996).

[11] Turner, K. (Ed.), Using Formal Description Techniques. An Introduction to Estelle, LOTOS and SDL, Wiley & Sons, 1993.

[12] De Marco, T., Structured Analysis, Yourdon Press, Nueva York (EE.UU.), 1979.

[13] Ward, P.T., The Transformation Schema: an Extensión of the Data Flow Diagram to Represent Control and Timing, IEEE Transactions on Software Engineering, Vol. 12, Nº 2, febrero 1986, págs. 189-210.

[14] Harel, D., Bitting the Silver Bullet, IEEE Software, 1993.

[15] i-Logix, The Statemate Workshop. Tutorial, i-Logix, 1995.

[16] Booch, G., Object Oriented Design with Applications, Redwood City, CA., Benjamming-Cummings, 1991.

[17] Halang, W.A., & A.D. Stoyenko, (Eds.), Real Time Computing, Springer-Verlag, 1994.

[18] Bologna, S. (Ed.), Incremental Prototyping Technology for Embedded Real-Time Systems, Special Issue of Real-Time Systems, Volume 5, Nº 2-3, Kluwer Academic Publishers, mayo 1993.

[19] Laplante, P.A., Real-Time Systems Design and Analysis. An Engineer's Handbook, IEEE Press, 1993.

[20] Fuggetta, A., A Classification of CASE Technology, IEEE Computer, diciembre 1993, págs. 25-38.

[21] Rendon, A., J.C. Dueñas, M. Miguel, Y. Leskela, J.A. Puente, G. León & A. Alonso, Animation of Heteregenous Prototypes of Real Time Systems, Conference of Engineering of Computer Complex Systems, Florida (EE.UU.), noviembre 1995.

[22] Fowler, P., & L. Levine, A Conceptual Framework for Software Technology Transition, Technical Report, CMU/SEI-93-TR-31, diciembre 1993.

[23] Kautz, K., J. Pries-Heje, T. Larsen, & P. Sorgaard, (Eds.), Diffusion and Adoption of Information Technology, First IFIP 8.6 Working Conference, Oslo (Noruega), octubre 1995.

[24] Brooks, F.P., No Silver Bullet, Essence and Accidents of Software Engineering, IEEE Computer, págs. 10-19, abril 1987.

Bibliografía

Bauer, F.L.: *A Trend for the Next 10 Years of Software Engineering,* Software Engineering, Ed. H. Freeman, P.M. Lewis, Academic Press, 1979.

Boehm, B. W.: *Software Engineering Economics,* Prentice-Hall, 1981.

Booch, G.: *Object Oriented Design with Applications,* Redwood City, CA. Benjamming-Cummings, 1991.

Brooks, F.: *The Mythical Man-Month,* Addison Wesley, 1975 (2ª edición en 1995).

Budde, R., K. Kautz, K. Kuhlenkamp & H. Zullighoven: *Prototyping: An approach to Evolutionary System Development,* Springer Verlag, 1992.

Hall, A.: *Seven Myths of Formal Methods,* IEEE Software, págs. 11-19, septiembre 1990.

Hoare, C.A.R.: *Communicating Sequential Processes,* Communications of the ACM, Vol. 21, Nº 8, agosto 1978.

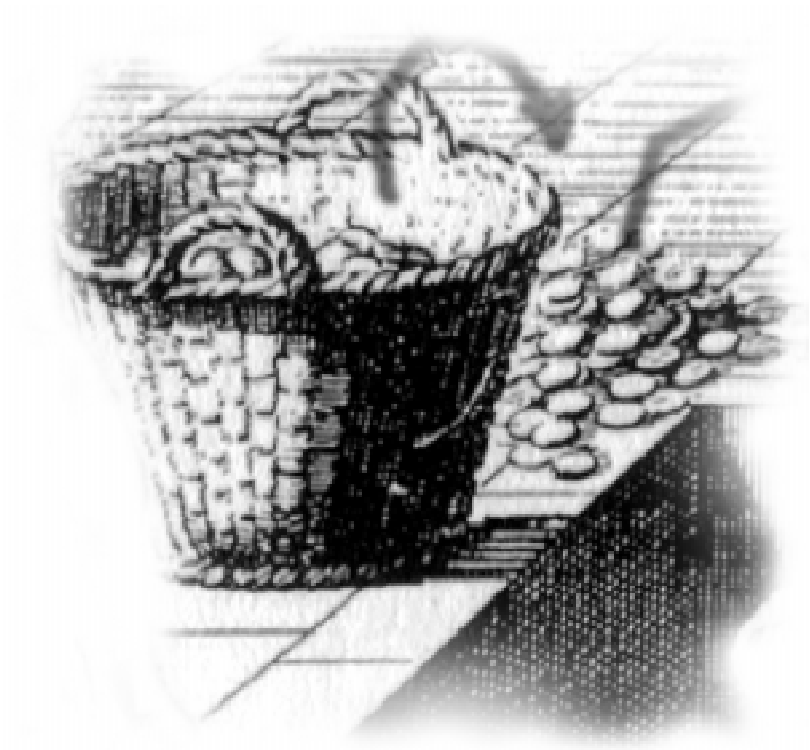
Parnas, D.L.: *On the Criteria to be used in Decomposing a System into Modules,* Communications of the ACM, Vol. 15 Nº 12, págs. 1053-1058, 1972.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy & W. Lorensen: *Object-oriented Modeling and Design,* Prentice-Hall, Int. 1991.

Sang, H.: *Advances in Real-Time Systems,* Prentice- Hall, 1995.

Thayer, R. H. & A. D. McGettrick (Eds.): *Software Engineering: A European Perspective,* IEEE Computer Society Press, 1993.

Glosario



1. **ACOPLAMIENTO.** Medida del intercambio de información entre módulos de un sistema de software durante la fase de diseño.

2. **ADT («Abstract Data Type»).** Tipo abstracto de datos. Concepto empleado en programación y base teórica de los métodos de desarrollo orientados a objetos.

3. **ANIMACIÓN.** Técnica de validación de un sistema de software por el que se visualiza la evolución dinámica del sistema mediante la ejecución de un modelo del mismo.

4. **ARQUITECTURA SOFTWARE.** Descripción de los módulos de un sistema de software y su relación.

5. **CAIE («Computer Aided Innovation Engineering»).** Conjunto de herramientas de software para el soporte del proceso de innovación.

6. **CASE («Computer Aided Software Engineering»).** Conjunto de herramientas de software integradas para apoyar el desarrollo de un sistema de software.

7. **CALIDAD DE UN PROCESO SOFTWARE.** Grado en el que el proceso realiza la función para la que se ha definido.

8. **CALIDAD DE UN PRODUCTO SOFTWARE.** Grado en el que satisface las expectativas planteadas por los usuarios.

9. **CASO DE PRUEBA.** Definición de una prueba concreta que debe superar un sistema de software.

10. COHESIÓN. Medida de la relación existente entre las funciones que se incluyen dentro de un módulo.

11. CMM («Capability Maturity Model»). Modelo de madurez de procesos de desarrollo de software propuesto en EEUU por el Instituto de Ingeniería de Software.

12. ENTORNO DE PROGRAMACIÓN. Sistema CASE que soporta la fase de implementación; por extensión, cualquier sistema CASE.

13. ESA («European Space Agency»). Agencia Espacial Europea.

14. IEEE («Institute of Electrical and Electronics Engineering»). Instituto de Ingenieros Eléctricos y Electrónicos. Organización profesional de EEUU.

15. INGENIERÍA DE SISTEMAS DE SOFTWARE. Aplicación de la ingeniería de sistemas al desarrollo de un sistema de software. Conjunto de técnicas de desarrollo y procedimientos de gestión necesarios para el desarrollo y mantenimiento de un sistema de software para obtener un sistema de calidad optimizando los recursos disponibles.

16. INTEGRACIÓN. Relación existente entre las herramientas de un sistema CASE. Se definen cuatro niveles de integración: visual, de datos, de control y de proceso.

17. ISO («International Standard Organization»). Organización Internacional de Normas.

18. LOTOS («Language for Temporal Ordering of Specifications»). Norma internacional propuesta por la ISO para la especificación formal de sistemas de comunicaciones.

19. MÉTRICAS DE DESARROLLO SOFTWARE. Factores susceptibles de ser medidos cuantitativamente en un sistema de software con el fin de evaluar algunos parámetros de calidad del mismo.

20. MODELO DE CICLO DE VIDA. Secuencia de fases ordenadas en el tiempo y las relaciones entre ellas que sirven de marco de referencia para el desarrollo de un sistema de software.

21. MODELO DE SÍNTESIS AUTOMÁTICA. Modelo de ciclo de vida en el que mediante el empleo de métodos formales es posible generar automáticamente el sistema de software a partir de la especificación del mismo.

22. MODELO EN CASCADA. Modelo de ciclo de vida convencional en el que el desarrollo se realiza en una serie de fases en las que, en cada una de ellas, se parte de los resultados alcanzados en la anterior.

23. MODELO EN ESPIRAL. Modelo de ciclo de vida orientado al control de riesgos propuesto por Boehm en el que el desarrollo se realiza en varios ciclos. Considerado como meta-modelo al permitir el uso de cualquiera de los modelos de ciclo de vida.

24. PROCESO. Conjunto de actividades orientadas a un fin concreto dentro del desarrollo o gestión de un sistema de software.

25. PROTOTIPADO. Técnica de desarrollo de software en el que se genera un sistema incompleto con el fin de ayudar a completar la especificación de requisitos o la arquitectura del sistema.

26. PROTOTIPADO INCREMENTAL. Técnica de desarrollo en la que se realizan diversos prototipos que van acercándose a la funcionalidad del sistema final.

27. PROTOTIPO. Sistema de software construido de forma rápida cuya misión es ayudar a fijar la funcionalidad deseada del sistema final. Utilizado como base para las técnicas de prototipado.

28. PROTOTIPO HETEROGÉNEO. Prototipo constituido por componentes a diferentes niveles de abstracción que cooperan para ofrecer la funcionalidad deseada por el usuario.

29. PUNTOS DE FUNCIÓN. Métrica empleada para conocer la complejidad de un sistema basada en el número de unidades funcionales que posee un módulo concreto de cinco tipos predefinidos.

30. REPOSITORIO. Componente de un sistema CASE en el que se controla el almacenamiento y acceso de la información relativa a un sistema de software en desarrollo.

31. REQUISITO FUNCIONAL. Relación precisa entre entradas y salidas que debe satisfacer un sistema de software.

32. REQUISITO NO FUNCIONAL. Atributo de calidad que debe satisfacer un sistema de software.

33. REQUISITOS. Funciones o limitaciones que debe satisfacer un sistema de software. Se denominan requisitos de usuario cuando son definidos por éstos y de sistemas cuando surgen de la funcionalidad que debe tener un sistema para cumplir con los requisitos de usuario.

34. SA/RT («Structured Analysis for Real Time»). Análisis Estructurado para Tiempo Real. Extensión de las técnicas de desarrollo estructurado.

35. SDL («Specification and Design Language»). Norma internacional promovida por el UIT-T (Unión Internacional de Telecomunicaciones) para la descripción de sistemas de comunicación.

36. SISTEMA DE SOFTWARE. Sistema en el que la funcionalidad ofrecida al usuario se consigue mediante el desarrollo de uno o varios programas ejecutables.

37. SISTEMA DE TIEMPO REAL (STR). Es aquél sistema de software que debe completar sus actividades en plazos de tiempo predeterminados. Como consecuencia, su ejecución debe satisfacer restricciones temporales cuyo incumplimiento supone el funcionamiento incorrecto del sistema.

38. TECNOLOGÍA DE SOFTWARE. Conjunto de elementos que puede utilizar un desarrollador de un sistema de software durante las diferentes fases del modelo de ciclo de vida elegido. Está constituida por un conjunto de notaciones, formas de razonar, herramientas, método de desarrollo y directrices de aplicación industrial.

39. TRANSFERENCIA DE TECNOLOGÍA. Conjunto de procedimientos necesarios para que una organización adopte una tecnología desde un proveedor de la misma.

40. VALIDACIÓN. Procedimientos de gestión requeridos para asegurar que un sistema de software satisface los requisitos de calidad impuestos.

41. VERIFICACIÓN. Procedimiento matemático para asegurar la corrección de un algoritmo

Esta primera edición de
INGENIERÍA DE SISTEMAS DE SOFTWARE
de la serie de
Monografías de Ingeniería de Sistemas
se terminó de imprimir el día
3 de mayo de 1996.