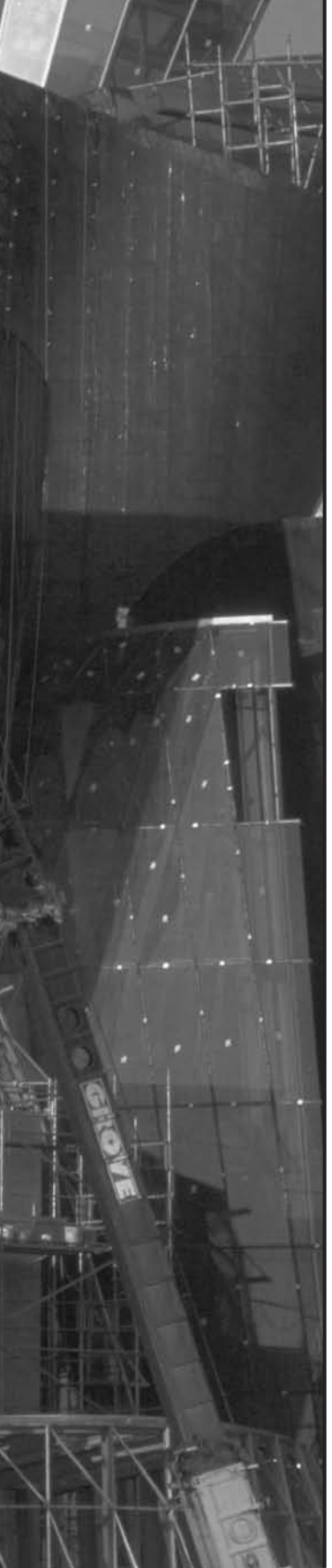


2

Procesos de software

Objetivos

El objetivo de este capítulo es introducirlo hacia la idea de un proceso de software: un conjunto coherente de actividades para la producción de software. Al estudiar este capítulo:

- comprenderá los conceptos y modelos sobre procesos de software;
- se introducirá en los tres modelos de proceso de software genérico y sabrá cuándo usarlos;
- entenderá las principales actividades del proceso de ingeniería de requerimientos de software, así como del desarrollo, las pruebas y la evolución del software;
- comprenderá por qué deben organizarse los procesos para enfrentar los cambios en los requerimientos y el diseño de software;
- entenderá cómo el Proceso Unificado Racional (Rational Unified Process, RUP) integra buenas prácticas de ingeniería de software para crear procesos de software adaptables.

Contenido

Un proceso de software es una serie de actividades relacionadas que conduce a la elaboración de un producto de software. Estas actividades pueden incluir el desarrollo de software desde cero en un lenguaje de programación estándar como Java o C. Sin embargo, las aplicaciones de negocios no se desarrollan precisamente de esta forma. El nuevo software empresarial con frecuencia ahora se desarrolla extendiendo y modificando los sistemas existentes, o configurando e integrando el software comercial o componentes del sistema.

Existen muchos diferentes procesos de software, pero todos deben incluir cuatro actividades que son fundamentales para la ingeniería de software:

1. *Especificación del software* Tienen que definirse tanto la funcionalidad del software como las restricciones de su operación.
2. *Diseño e implementación del software* Debe desarrollarse el software para cumplir con las especificaciones.
3. *Validación del software* Hay que validar el software para asegurarse de que cumple lo que el cliente quiere.
4. *Evolución del software* El software tiene que evolucionar para satisfacer las necesidades cambiantes del cliente.

En cierta forma, tales actividades forman parte de todos los procesos de software. Por supuesto, en la práctica éstas son actividades complejas en sí mismas e incluyen subactividades tales como la validación de requerimientos, el diseño arquitectónico, la prueba de unidad, etcétera. También existen actividades de soporte al proceso, como la documentación y el manejo de la configuración del software.

Cuando los procesos se discuten y describen, por lo general se habla de actividades como especificar un modelo de datos, diseñar una interfaz de usuario, etcétera, así como del orden de dichas actividades. Sin embargo, al igual que las actividades, también las descripciones de los procesos deben incluir:

1. Productos, que son los resultados de una actividad del proceso. Por ejemplo, el resultado de la actividad del diseño arquitectónico es un modelo de la arquitectura de software.
2. Roles, que reflejan las responsabilidades de la gente que interviene en el proceso. Ejemplos de roles: gerente de proyecto, gerente de configuración, programador, etcétera.
3. Precondiciones y postcondiciones, que son declaraciones válidas antes y después de que se realice una actividad del proceso o se cree un producto. Por ejemplo, antes de comenzar el diseño arquitectónico, una precondición es que el cliente haya aprobado todos los requerimientos; después de terminar esta actividad, una postcondición podría ser que se revisen aquellos modelos UML que describen la arquitectura.

Los procesos de software son complejos y, como todos los procesos intelectuales y creativos, se apoyan en personas con capacidad de juzgar y tomar decisiones. No hay un proceso ideal; además, la mayoría de las organizaciones han diseñado sus propios procesos de desarrollo de software. Los procesos han evolucionado para beneficiarse de las capacidades de la gente en una organización y de las características específicas de los

sistemas que se están desarrollando. Para algunos sistemas, como los sistemas críticos, se requiere de un proceso de desarrollo muy estructurado. Para los sistemas empresariales, con requerimientos rápidamente cambiantes, es probable que sea más efectivo un proceso menos formal y flexible.

En ocasiones, los procesos de software se clasifican como dirigidos por un plan (*plan-driven*) o como procesos ágiles. Los procesos dirigidos por un plan son aquellos donde todas las actividades del proceso se planean por anticipado y el avance se mide contra dicho plan. En los procesos ágiles, que se estudiarán en el capítulo 3, la planeación es incremental y es más fácil modificar el proceso para reflejar los requerimientos cambiantes del cliente. Como plantean Boehm y Turner (2003), cada enfoque es adecuado para diferentes tipos de software. Por lo general, uno necesita encontrar un equilibrio entre procesos dirigidos por un plan y procesos ágiles.

Aunque no hay un proceso de software "ideal", en muchas organizaciones sí existe un ámbito para mejorar el proceso de software. Los procesos quizás incluyan técnicas obsoletas o tal vez no aprovechen las mejores prácticas en la industria de la ingeniería de software. En efecto, muchas organizaciones aún no sacan ventaja de los métodos de la ingeniería de software en su desarrollo de software.

Los procesos de software pueden mejorarse con la estandarización de los procesos, donde se reduce la diversidad en los procesos de software en una organización. Esto conduce a mejorar la comunicación, a reducir el tiempo de capacitación, y a que el soporte de los procesos automatizados sea más económico. La estandarización también representa un primer paso importante tanto en la introducción de nuevos métodos y técnicas de ingeniería de software, como en sus buenas prácticas. En el capítulo 26 se analiza con más detalle la mejora en el proceso de software.

2.1 Modelos de proceso de software

Como se explicó en el capítulo 1, un modelo de proceso de software es una representación simplificada de este proceso. Cada modelo del proceso representa a otro desde una particular perspectiva y, por lo tanto, ofrece sólo información parcial acerca de dicho proceso. Por ejemplo, un modelo de actividad del proceso muestra las actividades y su secuencia, pero quizá sin presentar los roles de las personas que intervienen en esas actividades. En esta sección se introducen algunos modelos de proceso muy generales (en ocasiones llamados "paradigmas de proceso") y se muestran desde una perspectiva arquitectónica. En otras palabras, se ve el marco (framework) del proceso, pero no los detalles de las actividades específicas.

Tales modelos genéricos no son descripciones definitivas de los procesos de software. Más bien, son abstracciones del proceso que se utilizan para explicar los diferentes enfoques del desarrollo de software. Se pueden considerar marcos del proceso que se extienden y se adaptan para crear procesos más específicos de ingeniería de software.

Los modelos del proceso que se examinan aquí son:

1. *El modelo en cascada* (*waterfall*) Éste toma las actividades fundamentales del proceso de especificación, desarrollo, validación y evolución y, luego, los representa como fases separadas del proceso, tal como especificación de requerimientos, diseño de software, implementación, pruebas, etcétera.

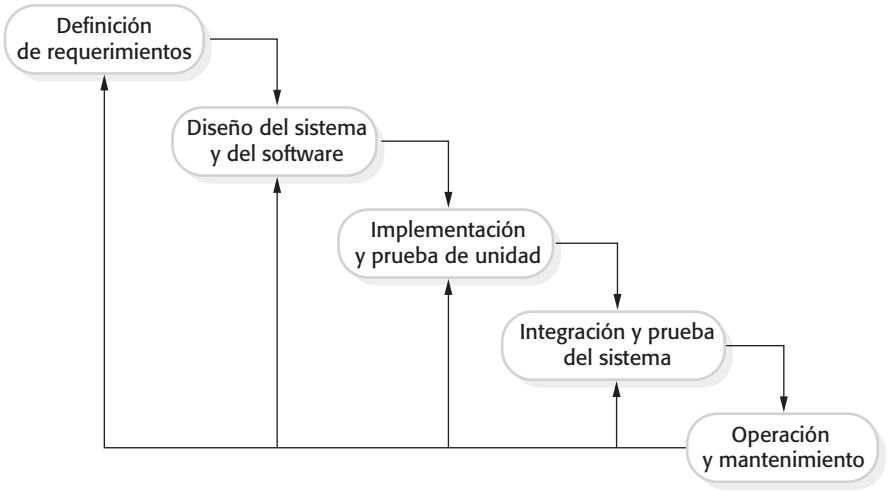


Figura 2.1 El modelo en cascada

2. *Desarrollo incremental* Este enfoque vincula las actividades de especificación, desarrollo y validación. El sistema se desarrolla como una serie de versiones (incrementos), y cada versión añade funcionalidad a la versión anterior.

3. *Ingeniería de software orientada a la reutilización* Este enfoque se basa en la existencia de un número significativo de componentes reutilizables. El proceso de desarrollo del sistema se enfoca en la integración de estos componentes en un sistema, en vez de desarrollarlo desde cero.

Dichos modelos no son mutuamente excluyentes y con frecuencia se usan en conjunto, sobre todo para el desarrollo de grandes sistemas. Para este tipo de sistemas, tiene sentido combinar algunas de las mejores características de los modelos de desarrollo en cascada e incremental. Se necesita contar con información sobre los requerimientos esenciales del sistema para diseñar la arquitectura de software que apoye dichos requerimientos. No puede desarrollarse de manera incremental. Los subsistemas dentro de un sistema más grande se desarrollan usando diferentes enfoques. Partes del sistema que son bien comprendidas pueden especificarse y desarrollarse al utilizar un proceso basado en cascada. Partes del sistema que por adelantado son difíciles de especificar, como la interfaz de usuario, siempre deben desarrollarse con un enfoque incremental.

2.1.1 El modelo en cascada

El primer modelo publicado sobre el proceso de desarrollo de software se derivó a partir de procesos más generales de ingeniería de sistemas (Royce, 1970). Este modelo se ilustra en la figura 2.1. Debido al paso de una fase en cascada a otra, este modelo se conoce como "modelo en cascada" o ciclo de vida del software. El modelo en cascada es un ejemplo de un proceso dirigido por un plan; en principio, usted debe planear y programar todas las actividades del proceso, antes de comenzar a trabajar con ellas.

Las principales etapas del modelo en cascada reflejan directamente las actividades fundamentales del desarrollo:

1. *Análisis y definición de requerimientos* Los servicios, las restricciones y las metas del sistema se establecen mediante consulta a los usuarios del sistema. Luego, se definen con detalle y sirven como una especificación del sistema.
2. *Diseño del sistema y del software* El proceso de diseño de sistemas asigna los requerimientos, para sistemas de hardware o de software, al establecer una arquitectura de sistema global. El diseño del software implica identificar y describir las abstracciones fundamentales del sistema de software y sus relaciones.
3. *Implementación y prueba de unidad* Durante esta etapa, el diseño de software se realiza como un conjunto de programas o unidades del programa. La prueba de unidad consiste en verificar que cada unidad cumpla con su especificación.
4. *Integración y prueba de sistema* Las unidades del programa o los programas individuales se integran y prueban como un sistema completo para asegurarse de que se cumplan los requerimientos de software. Después de probarlo, se libera el sistema de software al cliente.
5. *Operación y mantenimiento* Por lo general (aunque no necesariamente), ésta es la fase más larga del ciclo de vida, donde el sistema se instala y se pone en práctica. El mantenimiento incluye corregir los errores que no se detectaron en etapas anteriores del ciclo de vida, mejorar la implementación de las unidades del sistema e incrementar los servicios del sistema conforme se descubren nuevos requerimientos.

En principio, el resultado de cada fase consiste en uno o más documentos que se autorizaron (“firmaron”). La siguiente fase no debe comenzar sino hasta que termine la fase previa. En la práctica, dichas etapas se traslapan y se nutren mutuamente de información. Durante el diseño se identifican los problemas con los requerimientos. En la codificación se descubren problemas de diseño, y así sucesivamente. El proceso de software no es un simple modelo lineal, sino que implica retroalimentación de una fase a otra. Entonces, es posible que los documentos generados en cada fase deban modificarse para reflejar los cambios que se realizan.

Debido a los costos de producción y aprobación de documentos, las iteraciones suelen ser onerosas e implicar un rediseño significativo. Por lo tanto, después de un pequeño número de iteraciones, es normal detener partes del desarrollo, como la especificación, y continuar con etapas de desarrollo posteriores. Los problemas se dejan para una resolución posterior, se ignoran o se programan. Este freno prematuro de los requerimientos quizá signifique que el sistema no hará lo que el usuario desea. También podría conducir a sistemas mal estructurados conforme los problemas de diseño se evadan con la implementación de trucos.

Durante la fase final del ciclo de vida (operación y mantenimiento), el software se pone en servicio. Se descubren los errores y las omisiones en los requerimientos originales del software. Surgen los errores de programa y diseño, y se detecta la necesidad de nueva funcionalidad. Por lo tanto, el sistema debe evolucionar para mantenerse útil. Hacer tales cambios (mantenimiento de software) puede implicar la repetición de etapas anteriores del proceso.

Ingeniería de software de cuarto limpio

Un ejemplo del proceso de desarrollo formal, diseñado originalmente por IBM, es el proceso de cuarto limpio (*cleanroom*). En el proceso de cuarto limpio, cada incremento de software se especifica formalmente y tal especificación se transforma en una implementación. La exactitud del software se demuestra mediante un enfoque formal. No hay prueba de unidad para defectos en el proceso y la prueba del sistema se enfoca en la valoración de la fiabilidad del sistema.

El objetivo del proceso de cuarto limpio es obtener un software con cero defectos, de modo que los sistemas que se entreguen cuenten con un alto nivel de fiabilidad.

http://www.SoftwareEngineering-9.com/Web/Cleanroom/

El modelo en cascada es consecuente con otros modelos del proceso de ingeniería y en cada fase se produce documentación. Esto hace que el proceso sea visible, de modo que los administradores monitoricen el progreso contra el plan de desarrollo. Su principal problema es la partición inflexible del proyecto en distintas etapas. Tienen que establecerse compromisos en una etapa temprana del proceso, lo que dificulta responder a los requerimientos cambiantes del cliente.

En principio, el modelo en cascada sólo debe usarse cuando los requerimientos se entiendan bien y sea improbable el cambio radical durante el desarrollo del sistema. Sin embargo, el modelo en cascada refleja el tipo de proceso utilizado en otros proyectos de ingeniería. Como es más sencillo emplear un modelo de gestión común durante todo el proyecto, aún son de uso común los procesos de software basados en el modelo en cascada.

Una variación importante del modelo en cascada es el desarrollo de sistemas formales, donde se crea un modelo matemático para una especificación del sistema. Después se corrige este modelo, mediante transformaciones matemáticas que preservan su consistencia en un código ejecutable. Con base en la suposición de que son correctas sus transformaciones matemáticas, se puede aseverar, por lo tanto, que un programa generado de esta forma es consecuente con su especificación.

Los procesos formales de desarrollo, como el que se basa en el método B (Schneider, 2001; Wordsworth, 1996) son muy adecuados para el desarrollo de sistemas que cuenten con rigurosos requerimientos de seguridad, fiabilidad o protección. El enfoque formal simplifica la producción de un caso de protección o seguridad. Esto demuestra a los clientes o reguladores que el sistema en realidad cumple sus requerimientos de protección o seguridad.

Los procesos basados en transformaciones formales se usan por lo general sólo en el desarrollo de sistemas críticos para protección o seguridad. Requieren experiencia especializada. Para la mayoría de los sistemas, este proceso no ofrece costo/beneficio significativos sobre otros enfoques en el desarrollo de sistemas.

2.1.2 Desarrollo incremental

El desarrollo incremental se basa en la idea de diseñar una implementación inicial, exponer ésta al comentario del usuario, y luego desarrollarla en sus diversas versiones hasta producir un sistema adecuado (figura 2.2). Las actividades de especificación, desarrollo

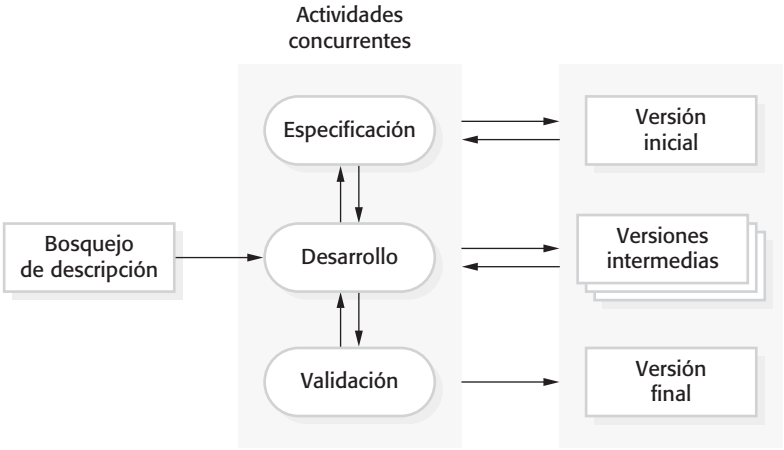


Figura 2.2 Desarrollo incremental

y validación están entrelazadas en vez de separadas, con rápida retroalimentación a través de las actividades.

El desarrollo de software incremental, que es una parte fundamental de los enfoques ágiles, es mejor que un enfoque en cascada para la mayoría de los sistemas empresariales, de comercio electrónico y personales. El desarrollo incremental refleja la forma en que se resuelven problemas. Rara vez se trabaja por adelantado una solución completa del problema, más bien se avanza en una serie de pasos hacia una solución y se retrocede cuando se detecta que se cometieron errores. Al desarrollar el software de manera incremental, resulta más barato y fácil realizar cambios en el software conforme éste se diseña.

Cada incremento o versión del sistema incorpora algunas de las funciones que necesita el cliente. Por lo general, los primeros incrementos del sistema incluyen la función más importante o la más urgente. Esto significa que el cliente puede evaluar el desarrollo del sistema en una etapa relativamente temprana, para constatar si se entrega lo que se requiere. En caso contrario, sólo el incremento actual debe cambiarse y, posiblemente, definir una nueva función para incrementos posteriores.

Comparado con el modelo en cascada, el desarrollo incremental tiene tres beneficios importantes:

1. Se reduce el costo de adaptar los requerimientos cambiantes del cliente. La cantidad de análisis y la documentación que tiene que reelaborarse son mucho menores de lo requerido con el modelo en cascada.
2. Es más sencillo obtener retroalimentación del cliente sobre el trabajo de desarrollo que se realizó. Los clientes pueden comentar las demostraciones del software y darse cuenta de cuánto se ha implementado. Los clientes encuentran difícil juzgar el avance a partir de documentos de diseño de software.
3. Es posible que sea más rápida la entrega e implementación de software útil al cliente, aun si no se ha incluido toda la funcionalidad. Los clientes tienen posibilidad de usar y ganar valor del software más temprano de lo que sería posible con un proceso en cascada.

Problemas con el desarrollo incremental

Aunque el desarrollo incremental tiene muchas ventajas, no está exento de problemas. La principal causa de la dificultad es el hecho de que las grandes organizaciones tienen procedimientos burocráticos que han evolucionado con el tiempo y pueden suscitar falta de coordinación entre dichos procedimientos y un proceso iterativo o ágil más informal.

En ocasiones, tales procedimientos se hallan ahí por buenas razones: por ejemplo, pueden existir procedimientos para garantizar que el software implementa de manera adecuada regulaciones externas (en Estados Unidos, por ejemplo, las regulaciones de contabilidad Sarbanes-Oxley). El cambio de tales procedimientos podría resultar imposible, de manera que los conflictos son inevitables.

http://www.SoftwareEngineering-9.com/Web/IncrementalDev/

El desarrollo incremental ahora es en cierta forma el enfoque más común para el desarrollo de sistemas de aplicación. Este enfoque puede estar basado en un plan, ser ágil o, más usualmente, una mezcla de dichos enfoques. En un enfoque basado en un plan se identifican por adelantado los incrementos del sistema; si se adopta un enfoque ágil, se detectan los primeros incrementos, aunque el desarrollo de incrementos posteriores depende del avance y las prioridades del cliente.

Desde una perspectiva administrativa, el enfoque incremental tiene dos problemas:

1. El proceso no es visible. Los administradores necesitan entregas regulares para medir el avance. Si los sistemas se desarrollan rápidamente, resulta poco efectivo en términos de costos producir documentos que reflejen cada versión del sistema.
2. La estructura del sistema tiende a degradarse conforme se tienen nuevos incrementos. A menos que se gaste tiempo y dinero en la refactorización para mejorar el software, el cambio regular tiende a corromper su estructura. La incorporación de más cambios de software se vuelve cada vez más difícil y costosa.

Los problemas del desarrollo incremental se tornan particularmente agudos para sistemas grandes, complejos y de larga duración, donde diversos equipos desarrollan diferentes partes del sistema. Los grandes sistemas necesitan de un marco o una arquitectura estable y es necesario definir con claridad, respecto a dicha arquitectura, las responsabilidades de los distintos equipos que trabajan en partes del sistema. Esto debe planearse por adelantado en vez de desarrollarse de manera incremental.

Se puede desarrollar un sistema incremental y exponerlo a los clientes para su comentario, sin realmente entregarlo e implementarlo en el entorno del cliente. La entrega y la implementación incrementales significan que el software se usa en procesos operacionales reales. Esto no siempre es posible, ya que la experimentación con un nuevo software llega a alterar los procesos empresariales normales. En la sección 2.3.2 se estudian las ventajas y desventajas de la entrega incremental.

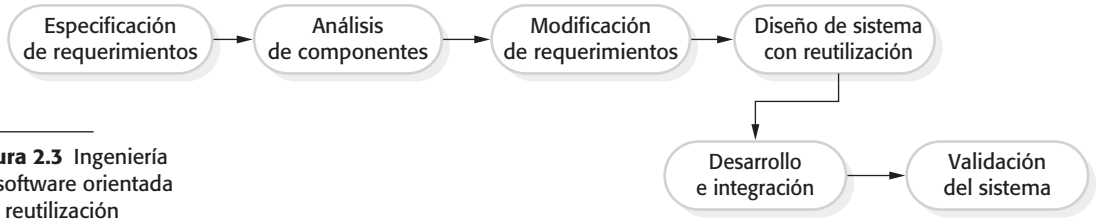


Figura 2.3 Ingeniería de software orientada a la reutilización

2.1.3 Ingeniería de software orientada a la reutilización

En la mayoría de los proyectos de software hay cierta reutilización de software. Sucede con frecuencia de manera informal, cuando las personas que trabajan en el proyecto conocen diseños o códigos que son similares a lo que se requiere. Los buscan, los modifican según se necesite y los incorporan en sus sistemas.

Esta reutilización informal ocurre independientemente del proceso de desarrollo que se emplee. Sin embargo, en el siglo XXI, los procesos de desarrollo de software que se enfocaban en la reutilización de software existente se utilizan ampliamente. Los enfoques orientados a la reutilización se apoyan en una gran base de componentes de software reutilizable y en la integración de marcos para la composición de dichos componentes. En ocasiones, tales componentes son sistemas por derecho propio (sistemas comerciales, off-the-shelf o COTS) que pueden mejorar la funcionalidad específica, como el procesador de textos o la hoja de cálculo.

En la figura 2.3 se muestra un modelo del proceso general para desarrollo basado en reutilización. Aunque la etapa inicial de especificación de requerimientos y la etapa de validación se comparan con otros procesos de software en un proceso orientado a la reutilización, las etapas intermedias son diferentes. Dichas etapas son:

1. *Análisis de componentes* Dada la especificación de requerimientos, se realiza una búsqueda de componentes para implementar dicha especificación. Por lo general, no hay coincidencia exacta y los componentes que se usan proporcionan sólo parte de la funcionalidad requerida.

2. *Modificación de requerimientos* Durante esta etapa se analizan los requerimientos usando información de los componentes descubiertos. Luego se modifican para reflejar los componentes disponibles. Donde las modificaciones son imposibles, puede regresarse a la actividad de análisis de componentes para buscar soluciones alternativas.

3. *Diseño de sistema con reutilización* Durante esta fase se diseña el marco conceptual del sistema o se reutiliza un marco conceptual existente. Los creadores toman en cuenta los componentes que se reutilizan y organizan el marco de referencia para atenderlo. Es posible que deba diseñarse algo de software nuevo, si no están disponibles los componentes reutilizables.

4. *Desarrollo e integración* Se diseña el software que no puede procurarse de manera externa, y se integran los componentes y los sistemas COTS para crear el nuevo sistema. La integración del sistema, en este modelo, puede ser parte del proceso de desarrollo, en vez de una actividad independiente.

Existen tres tipos de componentes de software que pueden usarse en un proceso orientado a la reutilización:

1. Servicios Web que se desarrollan en concordancia para atender servicios estándares y que están disponibles para la invocación remota.
2. Colecciones de objetos que se desarrollan como un paquete para su integración con un marco de componentes como .NET o J2EE.
3. Sistemas de software independientes que se configuran para usar en un entorno particular.

La ingeniería de software orientada a la reutilización tiene la clara ventaja de reducir la cantidad de software a desarrollar y, por lo tanto, la de disminuir costos y riesgos; por lo general, también conduce a entregas más rápidas del software. Sin embargo, son inevitables los compromisos de requerimientos y esto conduciría hacia un sistema que no cubra las necesidades reales de los usuarios. Más aún, se pierde algo de control sobre la evolución del sistema, conforme las nuevas versiones de los componentes reutilizables no estén bajo el control de la organización que los usa.

La reutilización de software es muy importante y en la tercera parte del libro se dedican varios capítulos a este tema. En el capítulo 16 se tratan los conflictos generales de la reutilización de software y la reutilización de COTS, en los capítulos 17 y 18 se estudia la ingeniería de software basada en componentes, y en el capítulo 19 se explican los sistemas orientados al servicio.

2.2 Actividades del proceso

Los procesos de software real son secuencias entrelazadas de actividades técnicas, colaborativas y administrativas con la meta general de especificar, diseñar, implementar y probar un sistema de software. Los desarrolladores de software usan en su trabajo diferentes herramientas de software. Las herramientas son útiles particularmente para dar apoyo a la edición de distintos tipos de documento y para manejar el inmenso volumen de información detallada que se reproduce en un gran proyecto de software.

Las cuatro actividades básicas de proceso de especificación, desarrollo, validación y evolución se organizan de diversa manera en diferentes procesos de desarrollo. En el modelo en cascada se organizan en secuencia, mientras que se entrelazan en el desarrollo incremental. La forma en que se llevan a cabo estas actividades depende del tipo de software, del personal y de la inclusión de estructuras organizativas. En la programación extrema, por ejemplo, las especificaciones se escriben en tarjetas. Las pruebas son ejecutables y se desarrollan antes del programa en sí. La evolución incluye la reestructuración o refactorización sustancial del sistema.

2.2.1 Especificación del software

La especificación del software o la ingeniería de requerimientos consisten en el proceso de comprender y definir qué servicios se requieren del sistema, así como la identificación de las restricciones sobre la operación y el desarrollo del sistema. La ingeniería de requerimientos es una etapa particularmente crítica del proceso de software, ya que los

Herramientas de desarrollo de software

Las herramientas de desarrollo del software (llamadas en ocasiones herramientas de Ingeniería de Software Asistido por Computadora o CASE, por las siglas de *Computer-Aided Software Engineering*) son programas usados para apoyar las actividades del proceso de la ingeniería de software. En consecuencia, estas herramientas incluyen editores de diseño, diccionarios de datos, compiladores, depuradores (*debuggers*), herramientas de construcción de sistema, etcétera.

Las herramientas de software ofrecen apoyo de proceso al automatizar algunas actividades del proceso y brindar información sobre el software que se desarrolla. Los ejemplos de actividades susceptibles de automatizarse son:

- Desarrollo de modelos de sistemas gráficos, como parte de la especificación de requerimientos o del diseño del software.
- Generación de código a partir de dichos modelos de sistemas gráficos.
- Producción de interfaces de usuario a partir de una descripción de interfaz gráfica, creada por el usuario de manera interactiva.
- Depuración del programa mediante el suministro de información sobre un programa que se ejecuta.
- Traducción automatizada de programas escritos, usando una versión anterior de un lenguaje de programación para tener una versión más reciente.

Las herramientas pueden combinarse en un marco llamado ambiente de desarrollo interactivo o IDE (por las siglas de *Interactive Development Environment*). Esto ofrece un conjunto común de facilidades, que usan las herramientas para comunicarse y operar con mayor destreza en una forma integrada. El ECLIPSE IDE se usa ampliamente y se diseñó para incorporar muchos tipos diferentes de herramientas de software.

http://www.SoftwareEngineering-9.com/Web/CASE/

errores en esta etapa conducen de manera inevitable a problemas posteriores tanto en el diseño como en la implementación del sistema.

El proceso de ingeniería de requerimientos (figura 2.4) se enfoca en producir un documento de requerimientos convenido que especifique los requerimientos de los interesados que cumplirá el sistema. Por lo general, los requerimientos se presentan en dos niveles de detalle. Los usuarios finales y clientes necesitan un informe de requerimientos de alto nivel; los desarrolladores de sistemas precisan una descripción más detallada del sistema.

Existen cuatro actividades principales en el proceso de ingeniería de requerimientos:

1. *Estudio de factibilidad* Se realiza una estimación sobre si las necesidades identificadas del usuario se cubren con las actuales tecnologías de software y hardware. El estudio considera si el sistema propuesto tendrá un costo-beneficio desde un punto de vista empresarial, y si éste puede desarrollarse dentro de las restricciones presupuestales existentes. Un estudio de factibilidad debe ser rápido y relativamente barato. El resultado debe informar la decisión respecto a si se continúa o no continúa con un análisis más detallado.
2. *Obtención y análisis de requerimientos* Éste es el proceso de derivar los requerimientos del sistema mediante observación de los sistemas existentes, discusiones con los usuarios y proveedores potenciales, análisis de tareas, etcétera. Esto puede incluir el desarrollo de uno o más modelos de sistemas y prototipos, lo que ayuda a entender el sistema que se va a especificar.

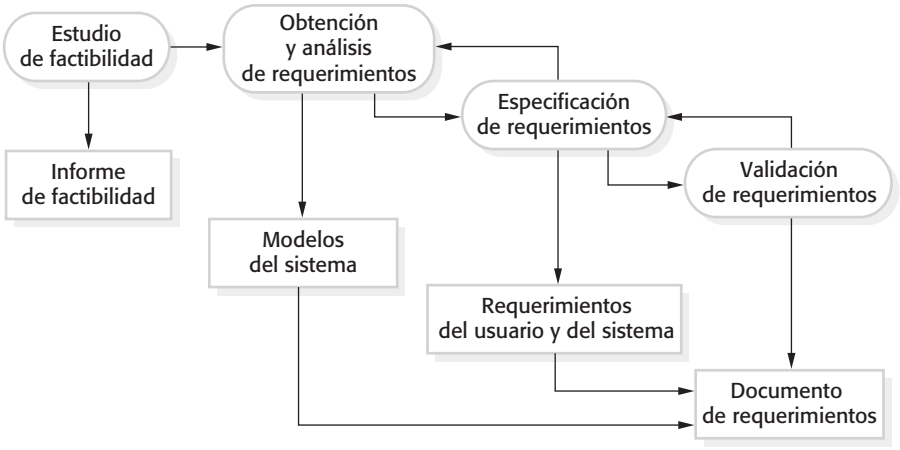


Figura 2.4 Proceso de ingeniería de requerimientos

3. *Especificación de requerimientos* Consiste en la actividad de transcribir la información recopilada durante la actividad de análisis, en un documento que define un conjunto de requerimientos. En este documento se incluyen dos clases de requerimientos. Los requerimientos del usuario son informes abstractos de requerimientos del sistema para el cliente y el usuario final del sistema; y los requerimientos de sistema son una descripción detallada de la funcionalidad a ofrecer.

4. *Validación de requerimientos* Esta actividad verifica que los requerimientos sean realistas, coherentes y completos. Durante este proceso es inevitable descubrir errores en el documento de requerimientos. En consecuencia, deberían modificarse con la finalidad de corregir dichos problemas.

Desde luego, las actividades en el proceso de requerimientos no se realizan simplemente en una secuencia estricta. El análisis de requerimientos continúa durante la definición y especificación, y a lo largo del proceso salen a la luz nuevos requerimientos; por lo tanto, las actividades de análisis, definición y especificación están vinculadas. En los métodos ágiles, como programación extrema, los requerimientos se desarrollan de manera incremental según las prioridades del usuario, en tanto que la obtención de requerimientos proviene de los usuarios que son parte del equipo de desarrollo.

2.2.2 Diseño e implementación del software

La etapa de implementación de desarrollo del software corresponde al proceso de convertir una especificación del sistema en un sistema ejecutable. Siempre incluye procesos de diseño y programación de software, aunque también puede involucrar la corrección en la especificación del software, si se utiliza un enfoque incremental de desarrollo.

Un diseño de software se entiende como una descripción de la estructura del software que se va a implementar, los modelos y las estructuras de datos utilizados por el sistema, las interfaces entre componentes del sistema y, en ocasiones, los algoritmos usados. Los diseñadores no llegan inmediatamente a una creación terminada, sino que desarrollan el diseño de manera iterativa. Agregan formalidad y detalle conforme realizan su diseño con *backtracking* (vuelta atrás) constante para corregir diseños anteriores.

La figura 2.5 es un modelo abstracto de este proceso, que ilustra las entradas al proceso de diseño, las actividades del proceso y los documentos generados como salidas de este proceso.

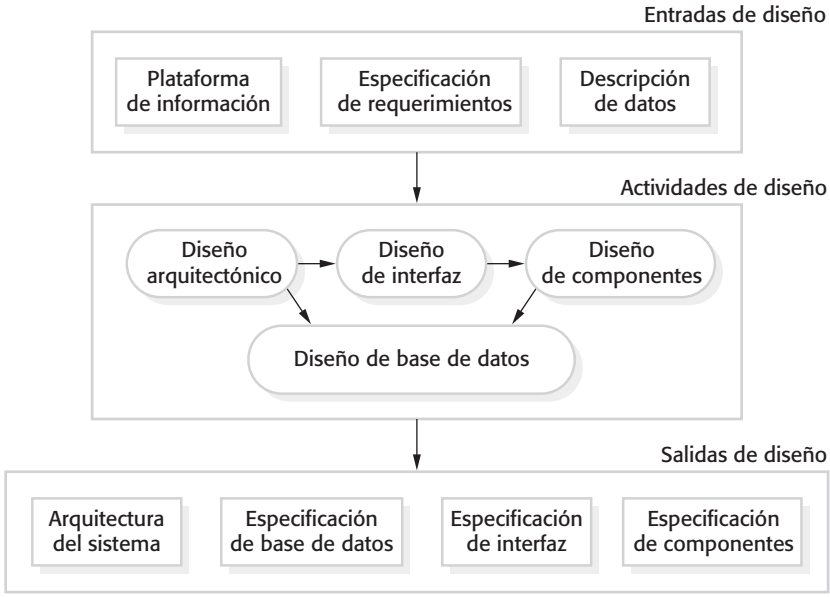


Figura 2.5 Modelo general del proceso de diseño

El diagrama sugiere que las etapas del proceso de diseño son secuenciales. De hecho, las actividades de proceso de diseño están vinculadas. En todos los procesos de diseño es inevitable la retroalimentación de una etapa a otra y la consecuente reelaboración del diseño.

La mayoría del software tiene interfaz junto con otros sistemas de software. En ellos se incluyen sistema operativo, base de datos, *middleware* y otros sistemas de aplicación. Éstos constituyen la "plataforma de software", es decir, el entorno donde se ejecutará el software. La información sobre esta plataforma es una entrada esencial al proceso de diseño, así que los diseñadores tienen que decidir la mejor forma de integrarla con el entorno de software. La especificación de requerimientos es una descripción de la funcionalidad que debe brindar el software, en conjunción con sus requerimientos de rendimiento y confiabilidad. Si el sistema debe procesar datos existentes, entonces en la especificación de la plataforma se incluirá la descripción de tales datos; de otro modo, la descripción de los datos será una entrada al proceso de diseño, de manera que se defina la organización del sistema de datos.

Las actividades en el proceso de diseño varían dependiendo del tipo de sistema a desarrollar. Por ejemplo, los sistemas de tiempo real precisan del diseño de temporización, pero sin incluir una base de datos, por lo que no hay que integrar un diseño de base de datos. La figura 2.5 muestra cuatro actividades que podrían formar parte del proceso de diseño para sistemas de información:

1. *Diseño arquitectónico*, aquí se identifica la estructura global del sistema, los principales componentes (llamados en ocasiones subsistemas o módulos), sus relaciones y cómo se distribuyen.
2. *Diseño de interfaz*, en éste se definen las interfaces entre los componentes de sistemas. Esta especificación de interfaz no tiene que presentar ambigüedades. Con una interfaz precisa, es factible usar un componente sin que otros tengan que saber cómo se implementó. Una vez que se acuerdan las especificaciones de interfaz, los componentes se diseñan y se desarrollan de manera concurrente.

Métodos estructurados

Los métodos estructurados son un enfoque al diseño de software donde se definen los modelos gráficos que hay que desarrollar, como parte del proceso de diseño. El método también define un proceso para diseñar los modelos y las reglas que se aplican a cada tipo de modelo. Los métodos estructurados conducen a documentación estandarizada para un sistema y son muy útiles al ofrecer un marco de desarrollo para los creadores de software con menor experiencia.

http://www.SoftwareEngineering-9.com/Web/Structured-methods/

3. *Diseño de componentes*, en él se toma cada componente del sistema y se diseña cómo funcionará. Esto puede ser un simple dato de la funcionalidad que se espera implementar, y al programador se le deja el diseño específico. Como alternativa, habría una lista de cambios a realizar sobre un componente que se reutiliza o sobre un modelo de diseño detallado. El modelo de diseño sirve para generar en automático una implementación.
4. *Diseño de base de datos*, donde se diseñan las estructuras del sistema de datos y cómo se representarán en una base de datos. De nuevo, el trabajo aquí depende de si una base de datos se reutilizará o se creará una nueva.

Tales actividades conducen a un conjunto de salidas de diseño, que también se muestran en la figura 2.5. El detalle y la representación de las mismas varían considerablemente. Para sistemas críticos, deben producirse documentos de diseño detallados que establezcan descripciones exactas del sistema. Si se usa un enfoque dirigido por un modelo, dichas salidas serían sobre todo diagramas. Donde se usen métodos ágiles de desarrollo, las salidas del proceso de diseño no podrían ser documentos de especificación separados, sino que tendrían que representarse en el código del programa.

Los métodos estructurados para el diseño se desarrollaron en las décadas de 1970 y 1980, y fueron precursores del UML y del diseño orientado a objetos (Budgen, 2003). Se apoyan en la producción de modelos gráficos del sistema y, en muchos casos, generan instantáneamente un código a partir de dichos modelos. El desarrollo dirigido por modelo (MDD) o la ingeniería dirigida por modelo (Schmidt, 2006), donde se crean modelos de software a diferentes niveles de abstracción, es una evolución de los métodos estructurados. En el MDD hay mayor énfasis en los modelos arquitectónicos con una separación entre modelos abstractos independientes de implementación y modelos específicos de implementación. Los modelos se desarrollan con detalle suficiente, de manera que el sistema ejecutable puede generarse a partir de ellos. En el capítulo 5 se estudia este enfoque de desarrollo.

El diseño de un programa para implementar el sistema se sigue naturalmente de los procesos de elaboración del sistema. Aunque algunas clases de programa, como los sistemas críticos para la seguridad, por lo general se diseñan con detalle antes de comenzar cualquier implementación, es más común que se entrelacen en etapas posteriores del diseño y el desarrollo del programa. Las herramientas de desarrollo de software se usan para generar un programa de "esqueleto" a partir de un diseño. Esto incluye un código para definir e implementar interfaces y, en muchos casos, el desarrollador sólo necesita agregar detalles de la operación de cada componente del programa.

La programación es una actividad personal y no hay un proceso que se siga de manera general. Algunos programadores comienzan con componentes que entienden, los desarrollan y, luego, cambian hacia componentes que entienden menos. Otros toman el enfoque opuesto,

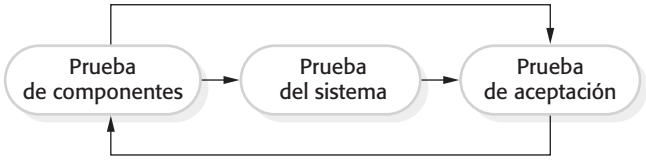


Figura 2.6 Etapas de pruebas

y dejan hasta el último los componentes familiares, porque saben cómo diseñarlos. A algunos desarrolladores les agrada definir con anticipación datos en el proceso, que luego usan para impulsar el desarrollo del programa; otros dejan datos sin especificar tanto como sea posible.

Por lo general, los programadores realizan algunas pruebas del código que desarrollaron. Esto revela con frecuencia defectos del programa que deben eliminarse del programa. A esta actividad se le llama depuración (*debugging*). La prueba de defectos y la depuración son procesos diferentes. La primera establece la existencia de defectos, en tanto que la segunda se dedica a localizar y corregir dichos defectos.

Cuando se depura, uno debe elaborar una hipótesis sobre el comportamiento observable del programa y, luego, poner a prueba dichas hipótesis con la esperanza de encontrar la falla que causó la salida anómala. Poner a prueba las hipótesis quizá requiera rastrear manualmente el código del programa; o bien, tal vez se necesiten nuevos casos de prueba para localizar el problema. Con la finalidad de apoyar el proceso de depuración, se deben utilizar herramientas interactivas que muestren valores intermedios de las variables del programa, así como el rastro de las instrucciones ejecutadas.

2.2.3 Validación de software

La validación de software o, más generalmente, su verificación y validación (V&V), se crea para mostrar que un sistema cumple tanto con sus especificaciones como con las expectativas del cliente. Las pruebas del programa, donde el sistema se ejecuta a través de datos de prueba simulados, son la principal técnica de validación. Esta última también puede incluir procesos de comprobación, como inspecciones y revisiones en cada etapa del proceso de software, desde la definición de requerimientos del usuario hasta el desarrollo del programa. Dada la predominancia de las pruebas, se incurre en la mayoría de los costos de validación durante la implementación y después de ésta.

Con excepción de los programas pequeños, los sistemas no deben ponerse a prueba como una unidad monolítica. La figura 2.6 muestra un proceso de prueba en tres etapas, donde los componentes del sistema se ponen a prueba; luego, se hace lo mismo con el sistema integrado y, finalmente, el sistema se pone a prueba con los datos del cliente. De manera ideal, los defectos de los componentes se detectan oportunamente en el proceso, en tanto que los problemas de interfaz se localizan cuando el sistema se integra. Sin embargo, conforme se descubran los defectos, el programa deberá depurarse y esto quizá requiera la repetición de otras etapas en el proceso de pruebas. Los errores en los componentes del programa pueden salir a la luz durante las pruebas del sistema. En consecuencia, el proceso es iterativo, con información retroalimentada desde etapas posteriores hasta las partes iniciales del proceso.

Las etapas en el proceso de pruebas son:

1. *Prueba de desarrollo* Las personas que desarrollan el sistema ponen a prueba los componentes que constituyen el sistema. Cada componente se prueba de manera independiente, es decir, sin otros componentes del sistema. Éstos pueden ser simples

entidades, como funciones o clases de objeto, o agrupamientos coherentes de dichas entidades. Por lo general, se usan herramientas de automatización de pruebas, como JUnit (Massol y Husted, 2003), que pueden volver a correr pruebas de componentes cuando se crean nuevas versiones del componente.

2. *Pruebas del sistema* Los componentes del sistema se integran para crear un sistema completo. Este proceso tiene la finalidad de descubrir errores que resulten de interacciones no anticipadas entre componentes y problemas de interfaz de componente, así como de mostrar que el sistema cubre sus requerimientos funcionales y no funcionales, y poner a prueba las propiedades emergentes del sistema. Para sistemas grandes, esto puede ser un proceso de múltiples etapas, donde los componentes se conjuntan para formar subsistemas que se ponen a prueba de manera individual, antes de que dichos subsistemas se integren para establecer el sistema final.

3. *Pruebas de aceptación* Ésta es la etapa final en el proceso de pruebas, antes de que el sistema se acepte para uso operacional. El sistema se pone a prueba con datos suministrados por el cliente del sistema, en vez de datos de prueba simulados. Las pruebas de aceptación revelan los errores y las omisiones en la definición de requerimientos del sistema, ya que los datos reales ejercitan el sistema en diferentes formas a partir de los datos de prueba. Asimismo, las pruebas de aceptación revelan problemas de requerimientos, donde las instalaciones del sistema en realidad no cumplan las necesidades del usuario o cuando sea inaceptable el rendimiento del sistema.

Por lo general, los procesos de desarrollo y de pruebas de componentes están entrelazados. Los programadores construyen sus propios datos de prueba y experimentan el código de manera incremental conforme lo desarrollan. Éste es un enfoque económicamente sensible, ya que el programador conoce el componente y, por lo tanto, es el más indicado para generar casos de prueba.

Si se usa un enfoque incremental para el desarrollo, cada incremento debe ponerse a prueba conforme se diseña, y tales pruebas se basan en los requerimientos para dicho incremento. En programación extrema, las pruebas se desarrollan junto con los requerimientos antes de comenzar el desarrollo. Esto ayuda a los examinadores y desarrolladores a comprender los requerimientos, y garantiza que no haya demoras conforme se creen casos de prueba.

Cuando se usa un proceso de software dirigido por un plan (como en el desarrollo de sistemas críticos), las pruebas se realizan mediante un conjunto de planes de prueba. Un equipo independiente de examinadores trabaja con base en dichos planes de prueba preformulados, que se desarrollaron a partir de la especificación y el diseño del sistema. La figura 2.7 ilustra cómo se vinculan los planes de prueba entre las actividades de pruebas y desarrollo. A esto se le conoce en ocasiones como modelo V de desarrollo (colóquelo de lado para distinguir la V).

En ocasiones, a las pruebas de aceptación se les identifica como “pruebas alfa”. Los sistemas a la medida se desarrollan sólo para un cliente. El proceso de prueba alfa continúa hasta que el desarrollador del sistema y el cliente estén de acuerdo en que el sistema entregado es una implementación aceptable de los requerimientos.

Cuando un sistema se marca como producto de software, se utiliza con frecuencia un proceso de prueba llamado “prueba beta”. Ésta incluye entregar un sistema a algunos clientes potenciales que están de acuerdo con usar ese sistema. Ellos reportan los problemas a los desarrolladores del sistema. Dicho informe expone el producto a uso real y detecta errores que no anticiparon los constructores del sistema. Después de esta retroalimentación, el sistema se modifica y libera, ya sea para más pruebas beta o para su venta general.

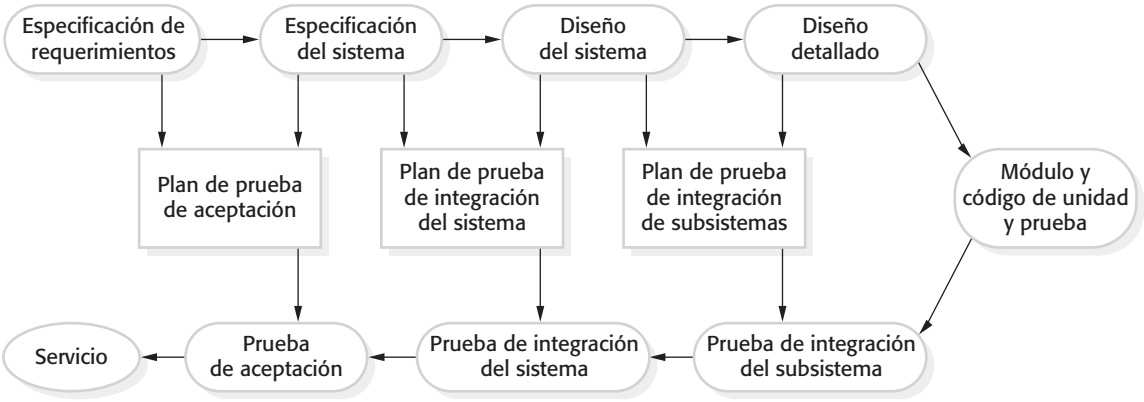


Figura 2.7 Probando fases en un proceso de software dirigido por un plan

2.2.4 Evolución del software

La flexibilidad de los sistemas de software es una de las razones principales por las que cada vez más software se incorpora en los sistemas grandes y complejos. Una vez tomada la decisión de fabricar hardware, resulta muy costoso hacer cambios a su diseño. Sin embargo, en cualquier momento durante o después del desarrollo del sistema, pueden hacerse cambios al software. Incluso los cambios mayores son todavía más baratos que los correspondientes cambios al hardware del sistema.

En la historia, siempre ha habido división entre el proceso de desarrollo del software y el proceso de evolución del software (mantenimiento de software). Las personas consideran el desarrollo de software como una actividad creativa, en la cual se diseña un sistema de software desde un concepto inicial y a través de un sistema de trabajo. No obstante, consideran en ocasiones el mantenimiento del software como insulso y poco interesante. Aunque en la mayoría de los casos los costos del mantenimiento son varias veces los costos iniciales de desarrollo, los procesos de mantenimiento se consideran en ocasiones como menos desafiantes que el desarrollo de software original.

Esta distinción entre desarrollo y mantenimiento es cada vez más irrelevante. Es muy difícil que cualquier sistema de software sea un sistema completamente nuevo, y tiene mucho más sentido ver el desarrollo y el mantenimiento como un continuo. En lugar de dos procesos separados, es más realista pensar en la ingeniería de software como un proceso evolutivo (figura 2.8), donde el software cambia continuamente a lo largo de su vida, en función de los requerimientos y las necesidades cambiantes del cliente.

2.3 Cómo enfrentar el cambio

El cambio es inevitable en todos los grandes proyectos de software. Los requerimientos del sistema varían conforme la empresa procura que el sistema responda a presiones externas y se modifican las prioridades administrativas. A medida que se ponen a disposición nuevas tecnologías, surgen nuevas posibilidades de diseño e implementación. Por ende, cualquiera que sea el modelo del proceso de software utilizado, es esencial que ajuste los cambios al software a desarrollar.

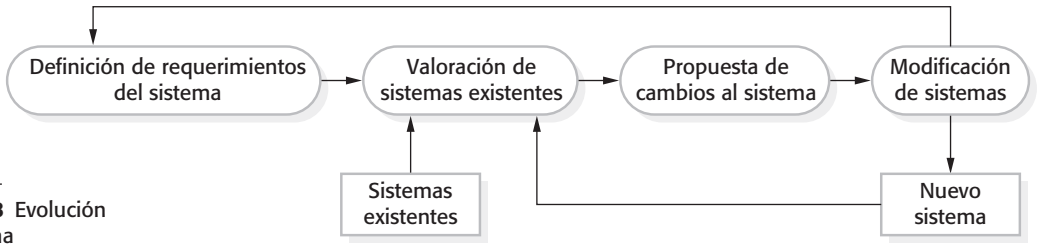


Figura 2.8 Evolución del sistema

El cambio se agrega a los costos del desarrollo de software debido a que, por lo general, significa que el trabajo ya terminado debe volver a realizarse. A esto se le llama rehacer. Por ejemplo, si se analizaron las relaciones entre los requerimientos en un sistema y se identifican nuevos requerimientos, parte o todo el análisis de requerimientos tiene que repetirse. Entonces, es necesario rediseñar el sistema para entregar los nuevos requerimientos, cambiar cualquier programa que se haya desarrollado y volver a probar el sistema.

Existen dos enfoques relacionados que se usan para reducir los costos del rehacer:

1. Evitar el cambio, donde el proceso de software incluye actividades que anticipan cambios posibles antes de requerirse la labor significativa de rehacer. Por ejemplo, puede desarrollarse un sistema prototipo para demostrar a los clientes algunas características clave del sistema. Ellos podrán experimentar con el prototipo y refinar sus requerimientos, antes de comprometerse con mayores costos de producción de software.
2. Tolerancia al cambio, donde el proceso se diseña de modo que los cambios se ajusten con un costo relativamente bajo. Por lo general, esto comprende algunas formas de desarrollo incremental. Los cambios propuestos pueden implementarse en incrementos que aún no se desarrollan. Si no es posible, entonces tal vez sólo un incremento (una pequeña parte del sistema) tendría que alterarse para incorporar el cambio.

En esta sección se estudian dos formas de enfrentar el cambio y los requerimientos cambiantes del sistema. Se trata de lo siguiente:

1. Prototipo de sistema, donde rápidamente se desarrolla una versión del sistema o una parte del mismo, para comprobar los requerimientos del cliente y la factibilidad de algunas decisiones de diseño. Esto apoya el hecho de evitar el cambio, al permitir que los usuarios experimenten con el sistema antes de entregarlo y así refinar sus requerimientos. Como resultado, es probable que se reduzca el número de propuestas de cambio de requerimientos posterior a la entrega.
2. Entrega incremental, donde los incrementos del sistema se entregan al cliente para su comentario y experimentación. Esto apoya tanto al hecho de evitar el cambio como a tolerar el cambio. Por un lado, evita el compromiso prematuro con los requerimientos para todo el sistema y, por otro, permite la incorporación de cambios en incrementos mayores a costos relativamente bajos.

La noción de refactorización, esto es, el mejoramiento de la estructura y organización de un programa, es también un mecanismo importante que apoya la tolerancia al cambio. Este tema se explica en el capítulo 3, que se ocupa de los métodos ágiles.

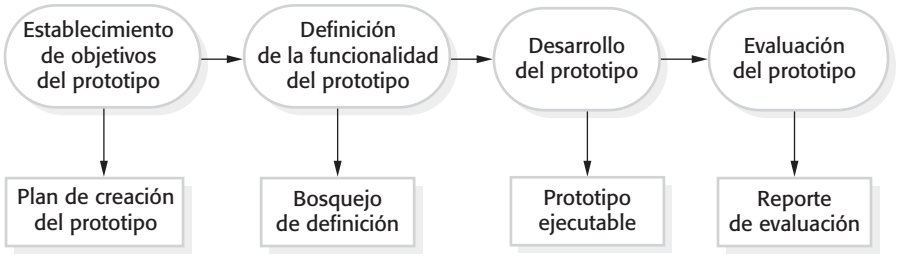


Figura 2.9 Proceso de desarrollo del prototipo

2.3.1 Creación del prototipo

Un prototipo es una versión inicial de un sistema de software que se usa para demostrar conceptos, tratar opciones de diseño y encontrar más sobre el problema y sus posibles soluciones. El rápido desarrollo iterativo del prototipo es esencial, de modo que se controlen los costos, y los interesados en el sistema experimenten por anticipado con el prototipo durante el proceso de software.

Un prototipo de software se usa en un proceso de desarrollo de software para contribuir a anticipar los cambios que se requieran:

1. En el proceso de ingeniería de requerimientos, un prototipo ayuda con la selección y validación de requerimientos del sistema.
2. En el proceso de diseño de sistemas, un prototipo sirve para buscar soluciones específicas de software y apoyar el diseño de interfaces del usuario.

Los prototipos del sistema permiten a los usuarios ver qué tan bien el sistema apoya su trabajo. Pueden obtener nuevas ideas para requerimientos y descubrir áreas de fortalezas y debilidades en el software. Entonces, proponen nuevos requerimientos del sistema. Más aún, conforme se desarrolla el prototipo, quizá se revelen errores y omisiones en los requerimientos propuestos. Una función descrita en una especificación puede parecer útil y bien definida. Sin embargo, cuando dicha función se combina con otras operaciones, los usuarios descubren frecuentemente que su visión inicial era incorrecta o estaba incompleta. Entonces, se modifica la especificación del sistema con la finalidad de reflejar su nueva comprensión de los requerimientos.

Mientras se elabora el sistema para la realización de experimentos de diseño, un prototipo del mismo sirve para comprobar la factibilidad de un diseño propuesto. Por ejemplo, puede crearse un prototipo del diseño de una base de datos y ponerse a prueba, con el objetivo de comprobar que soporta de forma eficiente el acceso de datos para las consultas más comunes del usuario. Asimismo, la creación de prototipos es una parte esencial del proceso de diseño de interfaz del usuario. Debido a la dinámica natural de las interfaces de usuario, las descripciones textuales y los diagramas no son suficientemente buenos para expresar los requerimientos de la interfaz del usuario. Por lo tanto, la creación rápida de prototipos con la participación del usuario final es la única forma sensible para desarrollar interfaces de usuario gráficas para sistemas de software.

En la figura 2.9 se muestra un modelo del proceso para desarrollo de prototipos. Los objetivos de la creación de prototipos deben ser más explícitos desde el inicio del proceso. Esto tendría la finalidad de desarrollar un sistema para un prototipo de la interfaz

del usuario, y diseñar un sistema que valide los requerimientos funcionales del sistema o desarrolle un sistema que demuestre a los administradores la factibilidad de la aplicación. El mismo prototipo no puede cumplir con todos los objetivos, ya que si éstos quedan sin especificar, los administradores o usuarios finales quizá malinterpreten la función del prototipo. En consecuencia, es posible que no obtengan los beneficios esperados del desarrollo del prototipo.

La siguiente etapa del proceso consiste en decidir qué poner y, algo quizá más importante, qué dejar fuera del sistema de prototipo. Para reducir los costos de creación de prototipos y acelerar las fechas de entrega, es posible dejar cierta funcionalidad fuera del prototipo y, también, decidir hacer más flexible los requerimientos no funcionales, como el tiempo de respuesta y la utilización de memoria. El manejo y la gestión de errores pueden ignorarse, a menos que el objetivo del prototipo sea establecer una interfaz de usuario. Además, es posible reducir los estándares de fiabilidad y calidad del programa.

La etapa final del proceso es la evaluación del prototipo. Hay que tomar provisiones durante esta etapa para la capacitación del usuario y usar los objetivos del prototipo para derivar un plan de evaluación. Los usuarios requieren tiempo para sentirse cómodos con un nuevo sistema e integrarse a un patrón normal de uso. Una vez que utilizan el sistema de manera normal, descubren errores y omisiones en los requerimientos.

Un problema general con la creación de prototipos es que quizás el prototipo no se utilice necesariamente en la misma forma que el sistema final. El revisor del prototipo tal vez no sea un usuario típico del sistema. También, podría resultar insuficiente el tiempo de capacitación durante la evaluación del prototipo. Si el prototipo es lento, los evaluadores podrían ajustar su forma de trabajar y evitar aquellas características del sistema con tiempos de respuesta lentos. Cuando se da una mejor respuesta en el sistema final, se puede usar de forma diferente.

En ocasiones, los desarrolladores están presionados por los administradores para entregar prototipos desechables, sobre todo cuando existen demoras en la entrega de la versión final del software. Sin embargo, por lo general esto no es aconsejable:

1. Puede ser imposible corregir el prototipo para cubrir requerimientos no funcionales, como los requerimientos de rendimiento, seguridad, robustez y fiabilidad, ignorados durante el desarrollo del prototipo.
2. El cambio rápido durante el desarrollo significa claramente que el prototipo no está documentado. La única especificación de diseño es el código del prototipo. Esto no es muy bueno para el mantenimiento a largo plazo.
3. Probablemente los cambios realizados durante el desarrollo de prototipos degradarán la estructura del sistema, y este último será difícil y costoso de mantener.
4. Por lo general, durante el desarrollo de prototipos se hacen más flexibles los estándares de calidad de la organización.

Los prototipos no tienen que ser ejecutables para ser útiles. Los modelos en papel de la interfaz de usuario del sistema (Rettig, 1994) pueden ser efectivos para ayudar a los usuarios a refinar un diseño de interfaz y trabajar a través de escenarios de uso. Su desarrollo es muy económico y suelen construirse en pocos días. Una extensión de esta técnica es un prototipo de Mago de Oz, donde sólo se desarrolle la interfaz del usuario. Los usuarios interactúan con esta interfaz, pero sus solicitudes pasan a una persona que los interpreta y les devuelve la respuesta adecuada.

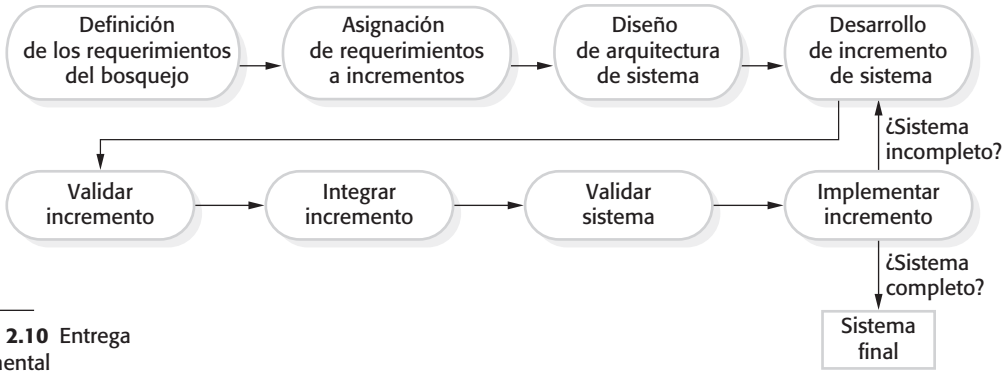


Figura 2.10 Entrega incremental

2.3.2 Entrega incremental

La entrega incremental (figura 2.10) es un enfoque al desarrollo de software donde algunos de los incrementos diseñados se entregan al cliente y se implementan para usarse en un entorno operacional. En un proceso de entrega incremental, los clientes identifican, en un bosquejo, los servicios que proporciona el sistema. Identifican cuáles servicios son más importantes y cuáles son menos significativos para ellos. Entonces, se define un número de incrementos de entrega, y cada incremento proporciona un subconjunto de la funcionalidad del sistema. La asignación de servicios por incrementos depende de la prioridad del servicio, donde los servicios de más alta prioridad se implementan y entregan primero.

Una vez identificados los incrementos del sistema, se definen con detalle los requerimientos de los servicios que se van a entregar en el primer incremento, y se desarrolla ese incremento. Durante el desarrollo, puede haber un mayor análisis de requerimientos para incrementos posteriores, aun cuando se rechacen cambios de requerimientos para el incremento actual.

Una vez completado y entregado el incremento, los clientes lo ponen en servicio. Esto significa que toman la entrega anticipada de la funcionalidad parcial del sistema. Pueden experimentar con el sistema que les ayuda a clarificar sus requerimientos, para posteriores incrementos del sistema. A medida que se completan nuevos incrementos, se integran con los incrementos existentes, de modo que con cada incremento entregado mejore la funcionalidad del sistema.

La entrega incremental tiene algunas ventajas:

1. Los clientes pueden usar los primeros incrementos como prototipos y adquirir experiencia que informe sobre sus requerimientos, para posteriores incrementos del sistema. A diferencia de los prototipos, éstos son parte del sistema real, de manera que no hay reaprendizaje cuando está disponible el sistema completo.
2. Los clientes deben esperar hasta la entrega completa del sistema, antes de ganar valor del mismo. El primer incremento cubre sus requerimientos más críticos, de modo que es posible usar inmediatamente el software.
3. El proceso mantiene los beneficios del desarrollo incremental en cuanto a que debe ser relativamente sencillo incorporar cambios al sistema.
4. Puesto que primero se entregan los servicios de mayor prioridad y luego se integran los incrementos, los servicios de sistema más importantes reciben mayores pruebas.

Esto significa que los clientes tienen menos probabilidad de encontrar fallas de software en las partes más significativas del sistema.

Sin embargo, existen problemas con la entrega incremental:

1. La mayoría de los sistemas requieren de una serie de recursos que se utilizan para diferentes partes del sistema. Dado que los requerimientos no están definidos con detalle sino hasta que se implementa un incremento, resulta difícil identificar recursos comunes que necesiten todos los incrementos.
2. Asimismo, el desarrollo iterativo resulta complicado cuando se diseña un sistema de reemplazo. Los usuarios requieren de toda la funcionalidad del sistema antiguo, ya que es común que no deseen experimentar con un nuevo sistema incompleto. Por lo tanto, es difícil conseguir retroalimentación útil del cliente.
3. La esencia de los procesos iterativos es que la especificación se desarrolla en conjunto con el software. Sin embargo, esto se puede contradecir con el modelo de adquisiciones de muchas organizaciones, donde la especificación completa del sistema es parte del contrato de desarrollo del sistema. En el enfoque incremental, no hay especificación completa del sistema, sino hasta que se define el incremento final. Esto requiere una nueva forma de contrato que los grandes clientes, como las agencias gubernamentales, encontrarían difícil de adoptar.

Existen algunos tipos de sistema donde el desarrollo incremental y la entrega no son el mejor enfoque. Hay sistemas muy grandes donde el desarrollo incluye equipos que trabajan en diferentes ubicaciones, algunos sistemas embebidos donde el software depende del desarrollo de hardware y algunos sistemas críticos donde todos los requerimientos tienen que analizarse para comprobar las interacciones que comprometan la seguridad o protección del sistema.

Estos sistemas, desde luego, enfrentan los mismos problemas de incertidumbre y requerimientos cambiantes. En consecuencia, para solucionar tales problemas y obtener algunos de los beneficios del desarrollo incremental, se utiliza un proceso donde un prototipo del sistema se elabore iterativamente y se utilice como plataforma, para experimentar con los requerimientos y el diseño del sistema. Con la experiencia obtenida del prototipo, pueden concertarse los requerimientos definitivos.

2.3.3 Modelo en espiral de Boehm

Boehm (1988) propuso un marco del proceso de software dirigido por el riesgo (el modelo en espiral), que se muestra en la figura 2.11. Aquí, el proceso de software se representa como una espiral, y no como una secuencia de actividades con cierto retroceso de una actividad a otra. Cada ciclo en la espiral representa una fase del proceso de software. Por ende, el ciclo más interno puede relacionarse con la factibilidad del sistema, el siguiente ciclo con la definición de requerimientos, el ciclo que sigue con el diseño del sistema, etcétera. El modelo en espiral combina el evitar el cambio con la tolerancia al cambio. Lo anterior supone que los cambios son resultado de riesgos del proyecto e incluye actividades de gestión de riesgos explícitas para reducir tales riesgos.

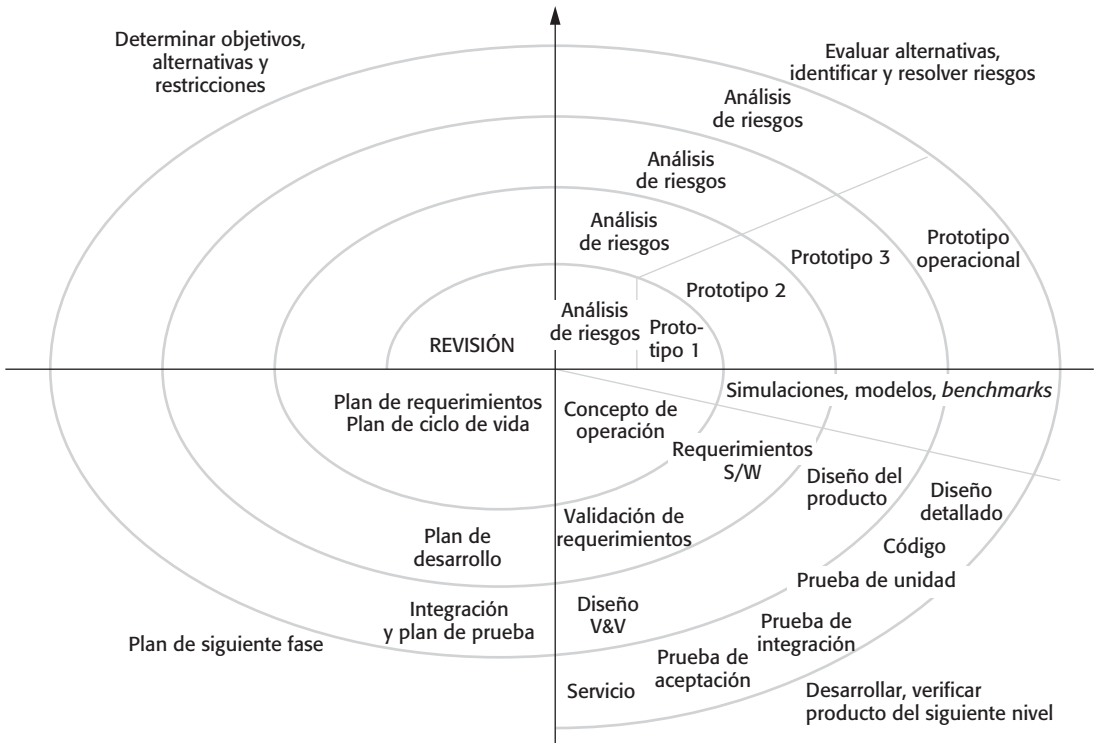


Figura 2.11 Modelo en espiral de Boehm del proceso de software (© IEEE, 1988)

Cada ciclo en la espiral se divide en cuatro sectores:

1. *Establecimiento de objetivos* Se definen objetivos específicos para dicha fase del proyecto. Se identifican restricciones en el proceso y el producto, y se traza un plan de gestión detallado. Se identifican los riesgos del proyecto. Pueden planearse estrategias alternativas, según sean los riesgos.

2. *Valoración y reducción del riesgo* En cada uno de los riesgos identificados del proyecto, se realiza un análisis minucioso. Se dan acciones para reducir el riesgo. Por ejemplo, si existe un riesgo de que los requerimientos sean inadecuados, puede desarrollarse un sistema prototipo.

3. *Desarrollo y validación* Después de una evaluación del riesgo, se elige un modelo de desarrollo para el sistema. Por ejemplo, la creación de prototipos desechables sería el mejor enfoque de desarrollo, si predominan los riesgos en la interfaz del usuario. Si la principal consideración son los riesgos de seguridad, el desarrollo con base en transformaciones formales sería el proceso más adecuado, entre otros. Si el principal riesgo identificado es la integración de subsistemas, el modelo en cascada sería el mejor modelo de desarrollo a utilizar.

4. *Planeación* El proyecto se revisa y se toma una decisión sobre si hay que continuar con otro ciclo de la espiral. Si se opta por continuar, se trazan los planes para la siguiente fase del proyecto.

La diferencia principal entre el modelo en espiral con otros modelos de proceso de software es su reconocimiento explícito del riesgo. Un ciclo de la espiral comienza por elaborar objetivos como rendimiento y funcionalidad. Luego, se numeran formas alternativas de alcanzar dichos objetivos y de lidiar con las restricciones en cada uno de ellos. Cada alternativa se valora contra cada objetivo y se identifican las fuentes de riesgo del proyecto. El siguiente paso es resolver dichos riesgos, mediante actividades de recopilación de información, como análisis más detallado, creación de prototipos y simulación.

Una vez valorados los riesgos se realiza cierto desarrollo, seguido por una actividad de planeación para la siguiente fase del proceso. De manera informal, el riesgo significa simplemente algo que podría salir mal. Por ejemplo, si la intención es usar un nuevo lenguaje de programación, un riesgo sería que los compiladores disponibles no sean confiables o no produzcan un código-objeto suficientemente eficaz. Los riesgos conducen a propuestas de cambios de software y a problemas de proyecto como exceso en las fechas y el costo, de manera que la minimización del riesgo es una actividad muy importante de administración del proyecto. En el capítulo 22 se tratará la gestión del riesgo, una parte esencial de la administración del proyecto.

2.4 El Proceso Unificado Racional

El Proceso Unificado Racional (RUP, por las siglas de *Rational Unified Process*) (Krutchen, 2003) es un ejemplo de un modelo de proceso moderno que se derivó del trabajo sobre el UML y el proceso asociado de desarrollo de software unificado (Rumbaugh *et al*., 1999; Arlow y Neustadt, 2005). Aquí se incluye una descripción, pues es un buen ejemplo de un modelo de proceso híbrido. Conjunta elementos de todos los modelos de proceso genéricos (sección 2.1), ilustra la buena práctica en especificación y diseño (sección 2.2), y apoya la creación de prototipos y entrega incremental (sección 2.3).

El RUP reconoce que los modelos de proceso convencionales presentan una sola visión del proceso. En contraste, el RUP por lo general se describe desde tres perspectivas:

1. Una perspectiva dinámica que muestra las fases del modelo a través del tiempo.
2. Una perspectiva estática que presenta las actividades del proceso que se establecen.
3. Una perspectiva práctica que sugiere buenas prácticas a usar durante el proceso.

La mayoría de las descripciones del RUP buscan combinar las perspectivas estática y dinámica en un solo diagrama (Krutchen, 2003). Esto hace que el proceso resulte más difícil de entender, por lo que en este texto se usan descripciones separadas de cada una de estas perspectivas.

El RUP es un modelo en fases que identifica cuatro fases discretas en el proceso de software. Sin embargo, a diferencia del modelo en cascada, donde las fases se igualan con actividades del proceso, las fases en el RUP están más estrechamente vinculadas con la empresa que con las preocupaciones técnicas. La figura 2.11 muestra las fases del RUP. Éstas son:

1. *Concepción* La meta de la fase de concepción es establecer un caso empresarial para el sistema. Deben identificarse todas las entidades externas (personas y sistemas)

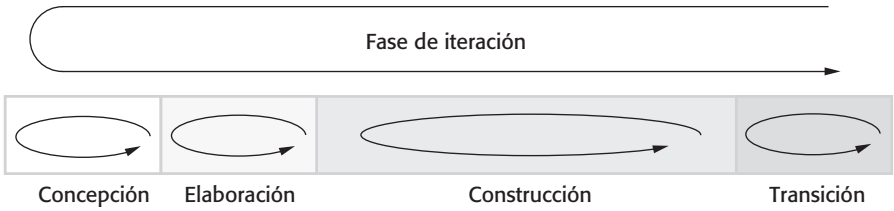


Figura 2.12 Fases en el Proceso Unificado Racional

que interactuarán con el sistema y definirán dichas interacciones. Luego se usa esta información para valorar la aportación del sistema hacia la empresa. Si esta aportación es menor, entonces el proyecto puede cancelarse después de esta fase.

2. *Elaboración* Las metas de la fase de elaboración consisten en desarrollar la comprensión del problema de dominio, establecer un marco conceptual arquitectónico para el sistema, diseñar el plan del proyecto e identificar los riesgos clave del proyecto. Al completar esta fase, debe tenerse un modelo de requerimientos para el sistema, que podría ser una serie de casos de uso del UML, una descripción arquitectónica y un plan de desarrollo para el software.

3. *Construcción* La fase de construcción incluye diseño, programación y pruebas del sistema. Partes del sistema se desarrollan en paralelo y se integran durante esta fase. Al completar ésta, debe tenerse un sistema de software funcionando y la documentación relacionada y lista para entregarse al usuario.

4. *Transición* La fase final del RUP se interesa por el cambio del sistema desde la comunidad de desarrollo hacia la comunidad de usuarios, y por ponerlo a funcionar en un ambiente real. Esto es algo ignorado en la mayoría de los modelos de proceso de software aunque, en efecto, es una actividad costosa y en ocasiones problemática. En el complemento de esta fase se debe tener un sistema de software documentado que funcione correctamente en su entorno operacional.

La iteración con el RUP se apoya en dos formas. Cada fase puede presentarse en una forma iterativa, con los resultados desarrollados incrementalmente. Además, todo el conjunto de fases puede expresarse de manera incremental, como se muestra en la flecha en curva desde *transición* hasta *concepción* en la figura 2.12.

La visión estática del RUP se enfoca en las actividades que tienen lugar durante el proceso de desarrollo. Se les llama flujos de trabajo en la descripción RUP. En el proceso se identifican seis flujos de trabajo de proceso centrales y tres flujos de trabajo de apoyo centrales. El RUP se diseñó en conjunto con el UML, de manera que la descripción del flujo de trabajo se orienta sobre modelos UML asociados, como modelos de secuencia, modelos de objeto, etcétera. En la figura 2.13 se describen la ingeniería central y los flujos de trabajo de apoyo.

La ventaja en la presentación de las visiones dinámica y estática radica en que las fases del proceso de desarrollo no están asociadas con flujos de trabajo específicos. En principio, al menos, todos los flujos de trabajo RUP pueden estar activos en la totalidad de las etapas del proceso. En las fases iniciales del proceso, es probable que se use mayor esfuerzo en los flujos de trabajo como modelado del negocio y requerimientos y, en fases posteriores, en las pruebas y el despliegue.

Flujo de trabajo	Descripción
Modelado del negocio	Se modelan los procesos de negocios utilizando casos de uso de la empresa.
Requerimientos	Se identifican los actores que interactúan con el sistema y se desarrollan casos de uso para modelar los requerimientos del sistema.
Análisis y diseño	Se crea y documenta un modelo de diseño utilizando modelos arquitectónicos, de componentes, de objetos y de secuencias.
Implementación	Se implementan y estructuran los componentes del sistema en subsistemas de implementación. La generación automática de código a partir de modelos de diseño ayuda a acelerar este proceso.
Pruebas	Las pruebas son un proceso iterativo que se realiza en conjunto con la implementación. Las pruebas del sistema siguen al completar la implementación.
Despliegue	Se crea la liberación de un producto, se distribuye a los usuarios y se instala en su lugar de trabajo.
Administración de la configuración y del cambio	Este flujo de trabajo de apoyo gestiona los cambios al sistema (véase el capítulo 25).
Administración del proyecto	Este flujo de trabajo de apoyo gestiona el desarrollo del sistema (véase los capítulos 22 y 23).
Entorno	Este flujo de trabajo pone a disposición del equipo de desarrollo de software, las herramientas adecuadas de software.

Figura 2.13 Flujos de trabajo estáticos en el Proceso Unificado Racional

El enfoque práctico del RUP describe las buenas prácticas de ingeniería de software que se recomiendan para su uso en el desarrollo de sistemas. Las seis mejores prácticas fundamentales que se recomiendan son:

1. *Desarrollo de software de manera iterativa* Incrementar el plan del sistema con base en las prioridades del cliente, y desarrollar oportunamente las características del sistema de mayor prioridad en el proceso de desarrollo.
2. *Gestión de requerimientos* Documentar de manera explícita los requerimientos del cliente y seguir la huella de los cambios a dichos requerimientos. Analizar el efecto de los cambios sobre el sistema antes de aceptarlos.
3. *Usar arquitecturas basadas en componentes* Estructurar la arquitectura del sistema en componentes, como se estudió anteriormente en este capítulo.
4. *Software modelado visualmente* Usar modelos UML gráficos para elaborar representaciones de software estáticas y dinámicas.
5. *Verificar la calidad del software* Garantizar que el software cumpla con los estándares de calidad de la organización.

6. *Controlar los cambios al software* Gestionar los cambios al software con un sistema de administración del cambio, así como con procedimientos y herramientas de administración de la configuración.

El RUP no es un proceso adecuado para todos los tipos de desarrollo, por ejemplo, para desarrollo de software embebido. Sin embargo, sí representa un enfoque que potencialmente combina los tres modelos de proceso genéricos que se estudiaron en la sección 2.1. Las innovaciones más importantes en el RUP son la separación de fases y flujos de trabajo, y el reconocimiento de que el despliegue del software en un entorno del usuario forma parte del proceso. Las fases son dinámicas y tienen metas. Los flujos de trabajo son estáticos y son actividades técnicas que no se asocian con una sola fase, sino que pueden usarse a lo largo del desarrollo para lograr las metas de cada fase.

PUNTOS CLAVE

- Los procesos de software son actividades implicadas en la producción de un sistema de software. Los modelos de proceso de software consisten en representaciones abstractas de dichos procesos.
- Los modelos de proceso general describen la organización de los procesos de software. Los ejemplos de estos modelos generales incluyen el modelo en cascada, el desarrollo incremental y el desarrollo orientado a la reutilización.
- La ingeniería de requerimientos es el proceso de desarrollo de una especificación de software. Las especificaciones tienen la intención de comunicar las necesidades de sistema del cliente a los desarrolladores del sistema.
- Los procesos de diseño e implementación tratan de transformar una especificación de requerimientos en un sistema de software ejecutable. Pueden usarse métodos de diseño sistemáticos como parte de esta transformación.
- La validación del software es el proceso de comprobar que el sistema se conforma a su especificación y que satisface las necesidades reales de los usuarios del sistema.
- La evolución del software tiene lugar cuando cambian los sistemas de software existentes para satisfacer nuevos requerimientos. Los cambios son continuos y el software debe evolucionar para seguir siendo útil.
- Los procesos deben incluir actividades para lidiar con el cambio. Esto puede implicar una fase de creación de prototipos que ayude a evitar malas decisiones sobre los requerimientos y el diseño. Los procesos pueden estructurarse para desarrollo y entrega iterativos, de forma que los cambios se realicen sin perturbar al sistema como un todo.
- El Proceso Unificado Racional es un modelo de proceso genérico moderno que está organizado en fases (concepción, elaboración, construcción y transición), pero separa las actividades (requerimientos, análisis y diseño, etcétera) de dichas fases.

LECTURAS SUGERIDAS

Managing Software Quality and Business Risk. Aun cuando éste es principalmente un libro sobre administración de software, incluye un excelente capítulo (capítulo 4) de modelos de proceso. (M. Ould, John Wiley and Sons Ltd, 1999.)

Process Models in Software Engineering. Ofrece una excelente visión de un amplio rango de modelos de proceso de ingeniería de software que se han propuesto. (W. Scacchi, *Encyclopaedia of Software Engineering,* ed. J.J. Marciniak, John Wiley and Sons, 2001.) http://www.ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SE-Encyc.pdf.

The Rational Unified Process—An Introduction (3rd edition). Éste es el libro más legible que hay disponible sobre RUP hasta ahora. Krutchen describe bien el proceso, pero sería más deseable ver las dificultades prácticas de usar el proceso. (P. Krutchen, Addison-Wesley, 2003.)

EJERCICIOS

2.1. Explicando las razones para su respuesta, y con base en el tipo de sistema a desarrollar, sugiera el modelo de proceso de software genérico más adecuado que se use como fundamento para administrar el desarrollo de los siguientes sistemas:

Un sistema para controlar el antibloqueo de frenos en un automóvil

Un sistema de realidad virtual para apoyar el mantenimiento de software

Un sistema de contabilidad universitario que sustituya a uno existente

Un sistema interactivo de programación de viajes que ayude a los usuarios a planear viajes con el menor impacto ambiental

2.2. Explique por qué el desarrollo incremental es el enfoque más efectivo para diseñar sistemas de software empresariales. ¿Por qué este modelo es menos adecuado para ingeniería de sistemas de tiempo real?

2.3. Considere el modelo de proceso basado en reutilización que se muestra en la figura 2.3. Explique por qué durante el proceso es esencial tener dos actividades separadas de ingeniería de requerimientos.

2.4. Sugiera por qué, en el proceso de ingeniería de requerimientos, es importante hacer una distinción entre desarrollar los requerimientos del usuario y desarrollar los requerimientos del sistema.

2.5. Describa las principales actividades en el proceso de diseño de software y las salidas de dichas actividades. Con un diagrama, muestre las posibles relaciones entre las salidas de dichas actividades.

2.6. Explique por qué el cambio es inevitable en los sistemas complejos, y mencione ejemplos (además de la creación de prototipos y la entrega incremental) de las actividades de proceso de software que ayudan a predecir los cambios y a lograr que el software por desarrollar sea más resistente al cambio.

2.7. Explique por qué los sistemas desarrollados como prototipos por lo general no deben usarse como sistemas de producción.

2.8. Exponga por qué el modelo en espiral de Boehm es un modelo adaptable que puede apoyar las actividades tanto de evitar el cambio como de tolerar el cambio. En la práctica, este modelo no se ha usado ampliamente. Sugiera por qué éste podría ser el caso.

2.9. ¿Cuáles son las ventajas de proporcionar visiones estática y dinámica del proceso de software como en el Proceso Unificado Racional?

2.10. Históricamente, la introducción de la tecnología ha causado profundos cambios en el mercado laboral y, al menos temporalmente, ha reemplazado a personas en los puestos de trabajo. Explique si es probable que la introducción de extensos procesos de automatización tenga las mismas consecuencias para los ingenieros de software. Si no cree que haya consecuencias, explique por qué. Si cree que reducirá las oportunidades laborales, ¿es ético que los ingenieros afectados resistan pasiva o activamente la introducción de esta tecnología?

REFERENCIAS

Arlow, J. y Neustadt, I. (2005). *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd Edition)*. Boston: Addison-Wesley.

Boehm, B. y Turner, R. (2003). *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston: Addison-Wesley.

Boehm, B. W. (1988). "A Spiral Model of Software Development and Enhancement". *IEEE Computer*, **21** (5), 61–72.

Budgen, D. (2003). *Software Design (2nd Edition)*. Harlow, UK.: Addison-Wesley.

Krutchen, P. (2003). *The Rational Unified Process—An Introduction (3rd Edition)*. Reading, MA: Addison-Wesley.

Massol, V. y Husted, T. (2003). *JUnit in Action*. Greenwich, Conn.: Manning Publications Co.

Rettig, M. (1994). "Practical Programmer: Prototyping for Tiny Fingers". *Comm. ACM*, **37** (4), 21–7.

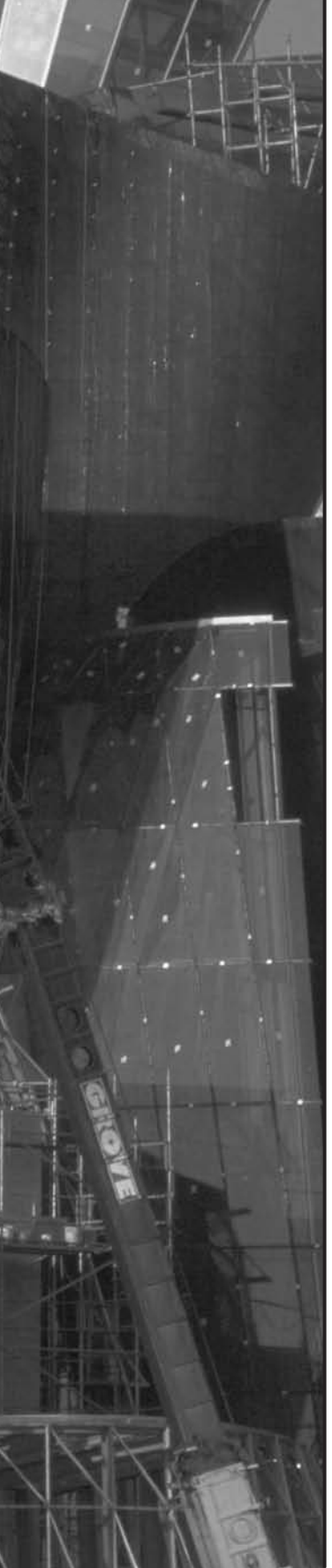
Royce, W. W. (1970). "Managing the Development of Large Software Systems: Concepts and Techniques". IEEE WESTCON, Los Angeles CA: 1–9.

Rumbaugh, J., Jacobson, I. y Booch, G. (1999). *The Unified Software Development Process*. Reading, Mass.: Addison-Wesley.

Schmidt, D. C. (2006). "Model-Driven Engineering". *IEEE Computer*, **39** (2), 25–31.

Schneider, S. (2001). *The B Method*. Houndmills, UK: Palgrave Macmillan.

Wordsworth, J. (1996). *Software Engineering with B*. Wokingham: Addison-Wesley.



4

Ingeniería de requerimientos

Objetivos

El objetivo de este capítulo es introducir los requerimientos de software y discutir los procesos que hay en el descubrimiento y la documentación de tales requerimientos. Al estudiar este capítulo:

- entenderá los conceptos de requerimientos del usuario y del sistema, así como por qué tales requerimientos se deben escribir en diferentes formas;
- comprenderá las diferencias entre requerimientos de software funcionales y no funcionales;
- reconocerá cómo se organizan los requerimientos dentro de un documento de requerimientos de software;
- conocerá las principales actividades de la ingeniería de requerimientos: adquisición, análisis y validación, así como las relaciones entre dichas actividades;
- analizará por qué es necesaria la administración de requerimientos y cómo ésta apoya otras actividades de la ingeniería de requerimientos.

Contenido

Los requerimientos para un sistema son descripciones de lo que el sistema debe hacer: el servicio que ofrece y las restricciones en su operación. Tales requerimientos reflejan las necesidades de los clientes por un sistema que atienda cierto propósito, como sería controlar un dispositivo, colocar un pedido o buscar información. Al proceso de descubrir, analizar, documentar y verificar estos servicios y restricciones se le llama ingeniería de requerimientos (IR).

El término "requerimiento" no se usa de manera continua en la industria del software. En algunos casos, un requerimiento es simplemente un enunciado abstracto de alto nivel en un servicio que debe proporcionar un sistema, o bien, una restricción sobre un sistema. En el otro extremo, consiste en una definición detallada y formal de una función del sistema. Davis (1993) explica por qué existen esas diferencias:

> *Si una compañía desea otorgar un contrato para un gran proyecto de desarrollo de software, tiene que definir sus necesidades de una forma suficientemente abstracta para que una solución no esté predefinida. Los requerimientos deben redactarse de tal forma que muchos proveedores liciten en pos del contrato, ofreciendo, tal vez, diferentes maneras de cubrir las necesidades de organización del cliente. Una vez otorgado el contrato, el proveedor tiene que escribir con más detalle una definición del sistema para el cliente, de modo que éste comprenda y valide lo que hará el software. Estos documentos suelen nombrarse documentos de requerimientos para el sistema.*

Algunos de los problemas que surgen durante el proceso de ingeniería de requerimientos son resultado del fracaso de hacer una separación clara entre esos diferentes niveles de descripción. En este texto se distinguen con el uso del término "requerimientos del usuario" para representar los requerimientos abstractos de alto nivel; y "requerimientos del sistema" para caracterizar la descripción detallada de lo que el sistema debe hacer. Los requerimientos del usuario y los requerimientos del sistema se definen del siguiente modo:

1. Los requerimientos del usuario son enunciados, en un lenguaje natural junto con diagramas, acerca de qué servicios esperan los usuarios del sistema, y de las restricciones con las cuales éste debe operar.
2. Los requerimientos del sistema son descripciones más detalladas de las funciones, los servicios y las restricciones operacionales del sistema de software. El documento de requerimientos del sistema (llamado en ocasiones especificación funcional) tiene que definir con exactitud lo que se implementará. Puede formar parte del contrato entre el comprador del sistema y los desarrolladores del software.

Los diferentes niveles de requerimientos son útiles debido a que informan sobre el sistema a distintos tipos de lector. La figura 4.1 ilustra la diferencia entre los requerimientos del usuario y del sistema. Este ejemplo de un sistema de administración de pacientes para apoyar la atención a la salud mental (MHC-PMS) muestra cómo los requerimientos del usuario se extienden hacia varios requerimientos del sistema. En la figura 4.1 se observa que el requerimiento del usuario es muy general. Los requerimientos del sistema ofrecen información más específica sobre los servicios y las funciones del sistema que se implementará.

Definición del requerimiento del usuario

1. El MHC-PMS elaborará mensualmente informes administrativos que revelen el costo de los medicamentos prescritos por cada clínica durante ese mes.

Especificación de los requerimientos del sistema

1.1 En el último día laboral de cada mes se redactará un resumen de los medicamentos prescritos, su costo y las clínicas que los prescriben.
1.2 El sistema elaborará automáticamente el informe que se imprimirá después de las 17:30 del último día laboral del mes.
1.3 Se realizará un reporte para cada clínica junto con los nombres de cada medicamento, el número de prescripciones, las dosis prescritas y el costo total de los medicamentos prescritos.
1.4 Si los medicamentos están disponibles en diferentes unidades de dosis (por ejemplo, 10 mg, 20 mg) se harán informes por separado para cada unidad de dosis.
1.5 El acceso a los informes de costos se restringirá a usuarios autorizados en la lista de control de acceso administrativo.

Figura 4.1 Requerimientos del usuario y requerimientos del sistema

Es necesario escribir los requerimientos con diferentes niveles de detalle, ya que varios lectores los usarán de distintas formas. La figura 4.2 muestra los posibles lectores de los requerimientos del usuario y los del sistema. De éstos, los primeros por lo general no están interesados en la manera en que se implementará el sistema, y quizá sean administradores a quienes no les atraigan las facilidades detalladas del sistema. Mientras que los segundos necesitan conocer con más precisión qué hará el sistema, ya que están preocupados sobre cómo apoyará los procesos de negocios o porque están inmersos en la implementación del sistema.

En este capítulo se presenta un panorama "tradicional" de los requerimientos, más que de los requerimientos en los procesos ágiles. Para la mayoría de los sistemas grandes, todavía se presenta una fase de ingeniería de requerimientos claramente identificable, antes de comenzar la implementación del sistema. El resultado es un documento de requerimientos que puede formar parte del contrato de desarrollo del sistema. Desde luego, por lo común hay cambios posteriores a los requerimientos, en tanto que los requerimientos del usuario podrían extenderse como requerimientos de sistema más detallados. Sin embargo, el enfoque ágil para alcanzar, al mismo tiempo, los requerimientos a medida que el sistema se desarrolla rara vez se utiliza en el diseño de sistemas grandes.

4.1 Requerimientos funcionales y no funcionales

A menudo, los requerimientos del sistema de software se clasifican como requerimientos funcionales o requerimientos no funcionales:

1. *Requerimientos funcionales* Son enunciados acerca de servicios que el sistema debe proveer, de cómo debería reaccionar el sistema a entradas particulares y de cómo

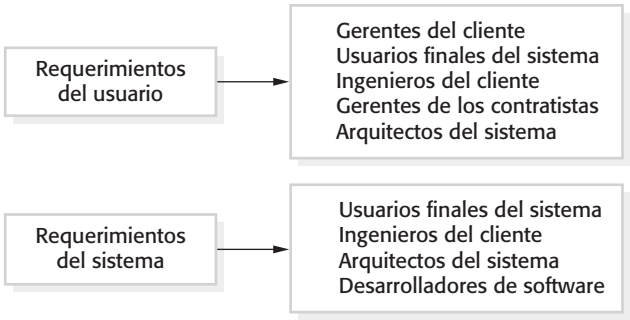


Figura 4.2 Lectores de diferentes tipos de especificación de requerimientos

debería comportarse el sistema en situaciones específicas. En algunos casos, los requerimientos funcionales también explican lo que no debe hacer el sistema.

2. *Requerimientos no funcionales* Son limitaciones sobre servicios o funciones que ofrece el sistema. Incluyen restricciones tanto de temporización y del proceso de desarrollo, como impuestas por los estándares. Los requerimientos no funcionales se suelen aplicar al sistema como un todo, más que a características o a servicios individuales del sistema.

En realidad, la distinción entre los diferentes tipos de requerimientos no es tan clara como sugieren estas definiciones sencillas. Un requerimiento de un usuario interesado por la seguridad, como el enunciado que limita el acceso a usuarios autorizados, parecería un requerimiento no funcional. Sin embargo, cuando se desarrolla con más detalle, este requerimiento puede generar otros requerimientos que son evidentemente funcionales, como la necesidad de incluir facilidades de autenticación en el sistema.

Esto muestra que los requerimientos no son independientes y que un requerimiento genera o restringe normalmente otros requerimientos. Por lo tanto, los requerimientos del sistema no sólo detallan los servicios o las características que se requieren del mismo, sino también especifican la funcionalidad necesaria para asegurar que estos servicios y características se entreguen de manera adecuada.

4.1.1 Requerimientos funcionales

Los requerimientos funcionales para un sistema refieren lo que el sistema debe hacer. Tales requerimientos dependen del tipo de software que se esté desarrollando, de los usuarios esperados del software y del enfoque general que adopta la organización cuando se escriben los requerimientos. Al expresarse como requerimientos del usuario, los requerimientos funcionales se describen por lo general de forma abstracta que entiendan los usuarios del sistema. Sin embargo, requerimientos funcionales más específicos del sistema detallan las funciones del sistema, sus entradas y salidas, sus excepciones, etcétera.

Los requerimientos funcionales del sistema varían desde requerimientos generales que cubren lo que tiene que hacer el sistema, hasta requerimientos muy específicos que reflejan maneras locales de trabajar o los sistemas existentes de una organización. Por ejemplo, veamos algunos casos de requerimientos funcionales para el sistema MHC-PMS, que

Requerimientos de dominio

Los requerimientos de dominio se derivan del dominio de aplicación del sistema, más que a partir de las necesidades específicas de los usuarios del sistema. Pueden ser requerimientos funcionales nuevos por derecho propio, restricciones a los requerimientos funcionales existentes o formas en que deben realizarse cálculos particulares.

El problema con los requerimientos de dominio es que los ingenieros de software no pueden entender las características del dominio en que opera el sistema. Por lo común, no pueden indicar si un requerimiento de dominio se perdió o entró en conflicto con otros requerimientos.

http://www.SoftwareEngineering-9.com/Web/Requirements/DomainReq.html

se usan para mantener información de pacientes que reciben tratamiento por problemas de salud mental:

1. Un usuario podrá buscar en todas las clínicas las listas de citas.
2. El sistema elaborará diariamente, para cada clínica, una lista de pacientes que se espera que asistan a cita ese día.
3. Cada miembro del personal que usa el sistema debe identificarse de manera individual con su número de ocho dígitos.

Estos requerimientos funcionales del usuario definen las actividades específicas que debe proporcionar el sistema. Se tomaron del documento de requerimientos del usuario y muestran que los requerimientos funcionales pueden escribirse con diferentes niveles de detalle (contraste los requerimientos 1 y 3).

La inexactitud en la especificación de requerimientos causa muchos problemas en la ingeniería de software. Es natural que un desarrollador de sistemas interprete un requerimiento ambiguo de forma que simplifique su implementación. Sin embargo, con frecuencia, esto no es lo que desea el cliente. Tienen que establecerse nuevos requerimientos y efectuar cambios al sistema. Desde luego, esto aplaza la entrega del sistema y aumenta los costos.

Es el caso del primer ejemplo de requerimiento para el MHC-PMS que establece que un usuario podrá buscar las listas de citas en todas las clínicas. El motivo para este requerimiento es que los pacientes con problemas de salud mental en ocasiones están confundidos. Quizá tengan una cita en una clínica y en realidad acudan a una diferente. De ahí que si tienen una cita, se registrará que asistieron, sin importar la clínica.

Los miembros del personal médico que especifican esto quizás esperen que "buscar" significa que, dado el nombre de un paciente, el sistema busca dicho nombre en las citas de todas las clínicas. Sin embargo, esto no es claro en el requerimiento. Los desarrolladores del sistema pueden interpretar el requerimiento de forma diferente e implementar una búsqueda, de tal modo que el usuario deba elegir una clínica y luego realizar la búsqueda. Evidentemente, esto implicará más entradas del usuario y tomará más tiempo.

En principio, la especificación de los requerimientos funcionales de un sistema debe ser completa y consistente. Totalidad significa que deben definirse todos los servicios requeridos por el usuario. Consistencia quiere decir que los requerimientos tienen que evitar definiciones contradictorias. En la práctica, para sistemas complejos grandes, es

casi imposible lograr la consistencia y la totalidad de los requerimientos. Una causa para ello es la facilidad con que se cometen errores y omisiones al escribir especificaciones para sistemas complejos. Otra es que hay muchos participantes en un sistema grande. Un participante es un individuo o una función que se ve afectado de alguna forma por el sistema. Los participantes tienen diferentes necesidades, pero con frecuencia son inconsistentes. Tales inconsistencias tal vez no sean evidentes cuando se especifican por primera vez los requerimientos, de modo que en la especificación se incluyen requerimientos inconsistentes. Los problemas suelen surgir sólo después de un análisis en profundidad o después de que se entregó el sistema al cliente.

4.1.2 Requerimientos no funcionales

Los requerimientos no funcionales, como indica su nombre, son requerimientos que no se relacionan directamente con los servicios específicos que el sistema entrega a sus usuarios. Pueden relacionarse con propiedades emergentes del sistema, como fiabilidad, tiempo de respuesta y uso de almacenamiento. De forma alternativa, pueden definir restricciones sobre la implementación del sistema, como las capacidades de los dispositivos I/O o las representaciones de datos usados en las interfaces con otros sistemas.

Los requerimientos no funcionales, como el rendimiento, la seguridad o la disponibilidad, especifican o restringen por lo general características del sistema como un todo. Los requerimientos no funcionales a menudo son más significativos que los requerimientos funcionales individuales. Es común que los usuarios del sistema encuentren formas para trabajar en torno a una función del sistema que realmente no cubre sus necesidades. No obstante, el fracaso para cubrir los requerimientos no funcionales haría que todo el sistema fuera inútil. Por ejemplo, si un sistema de aeronave no cubre sus requerimientos de fiabilidad, no será certificado para su operación como dispositivo seguro; si un sistema de control embebido fracasa para cubrir sus requerimientos de rendimiento, no operarán correctamente las funciones de control.

Aunque es posible identificar con regularidad cuáles componentes de sistema implementan requerimientos funcionales específicos (por ejemplo, hay componentes de formateo que implementan requerimientos de informe), por lo general es más difícil relacionar componentes con requerimientos no funcionales. La implementación de dichos requerimientos puede propagarse a lo largo del sistema. Para esto existen dos razones:

1. Los requerimientos no funcionales afectan más la arquitectura global de un sistema que los componentes individuales. Por ejemplo, para garantizar que se cumplan los requerimientos de rendimiento, quizá se deba organizar el sistema para minimizar las comunicaciones entre componentes.
2. Un requerimiento no funcional individual, como un requerimiento de seguridad, podría generar algunos requerimientos funcionales relacionados que definan nuevos servicios del sistema que se requieran. Además, también podría generar requerimientos que restrinjan los requerimientos ya existentes.

Los requerimientos no funcionales surgen a través de necesidades del usuario, debido a restricciones presupuestales, políticas de la organización, necesidad de interoperabilidad con otro software o sistemas de hardware, o factores externos como regulaciones de seguridad o legislación sobre privacidad. La figura 4.3 es una clasificación de requerimientos

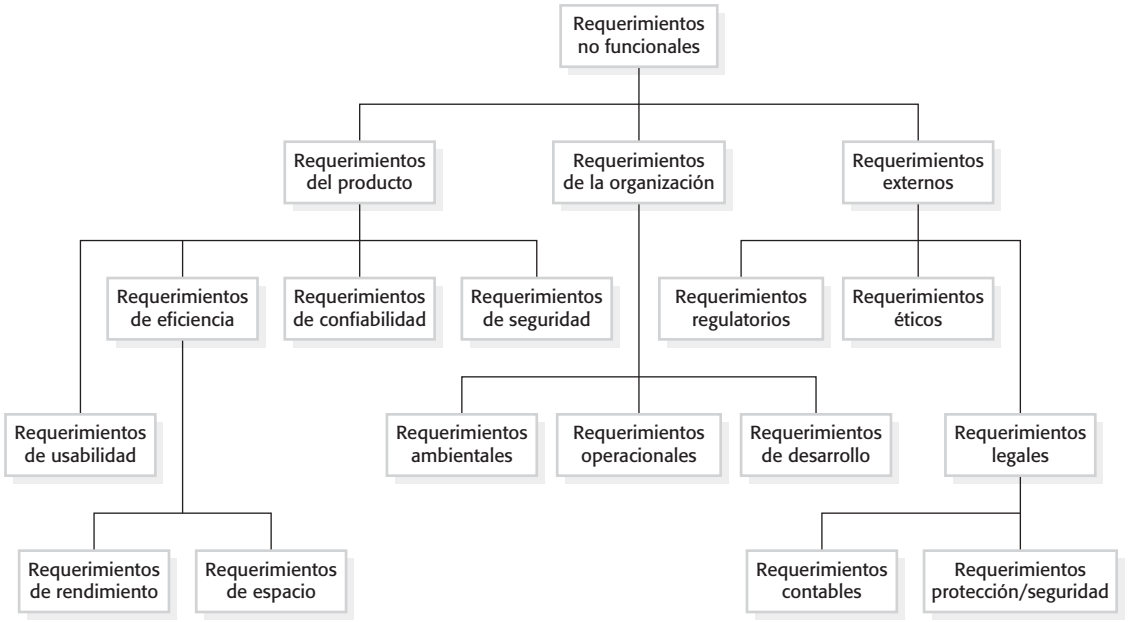


Figura 4.3 Tipos de requerimientos no funcionales

no funcionales. Observe a partir de este diagrama que los requerimientos no funcionales provienen de características requeridas del software (requerimientos del producto), la organización que desarrolla el software (requerimientos de la organización) o de fuentes externas:

1. *Requerimientos del producto* Estos requerimientos especifican o restringen el comportamiento del software. Los ejemplos incluyen requerimientos de rendimiento sobre qué tan rápido se debe ejecutar el sistema y cuánta memoria requiere, requerimientos de fiabilidad que establecen la tasa aceptable de fallas, requerimientos de seguridad y requerimientos de usabilidad.
2. *Requerimientos de la organización* Son requerimientos de sistemas amplios, derivados de políticas y procedimientos en la organización del cliente y del desarrollador. Los ejemplos incluyen requerimientos del proceso operacional que definen cómo se usará el sistema, requerimientos del proceso de desarrollo que especifican el lenguaje de programación, estándares del entorno o el proceso de desarrollo a utilizar, y requerimientos ambientales que definen el entorno de operación del sistema.
3. *Requerimientos externos* Este término cubre todos los requerimientos derivados de factores externos al sistema y su proceso de desarrollo. En ellos se incluyen requerimientos regulatorios que establecen lo que debe hacer el sistema para ser aprobado en su uso por un regulador, como sería un banco central; requerimientos legislativos que tienen que seguirse para garantizar que el sistema opere conforme a la ley, y requerimientos éticos que garanticen que el sistema será aceptable para sus usuarios y el público en general.

La figura 4.4 muestra ejemplos de requerimientos del producto, de la organización y requerimientos externos tomados del MHC-PMS, cuyos requerimientos de usuario se

REQUERIMIENTO DEL PRODUCTO
El MHC-PMS estará disponible en todas las clínicas durante las horas de trabajo normales (lunes a viernes, de 8:30 a 17:30). En cualquier día, los tiempos muertos dentro de las horas laborales normales no rebasarán los cinco segundos.

REQUERIMIENTOS DE LA ORGANIZACIÓN
Los usuarios del sistema MHC-PMS se acreditarán a sí mismos con el uso de la tarjeta de identidad de la autoridad sanitaria.

REQUERIMIENTOS EXTERNOS
Como establece la HStan-03-2006-priv, el sistema implementará provisiones para la privacidad del paciente.

Figura 4.4 Ejemplos de requerimientos no funcionales en el MHC-PMS

introdujeron en la sección 4.1.1. El requerimiento del producto es un requerimiento de disponibilidad que define cuándo estará disponible el sistema y el tiempo muerto permitido cada día. No dice algo sobre la funcionalidad del MHC-PMS e identifica con claridad una restricción que deben considerar los diseñadores del sistema.

El requerimiento de la organización especifica cómo se autentican los usuarios en el sistema. La autoridad sanitaria que opera el sistema se mueve hacia un procedimiento de autenticación estándar para cualquier software donde, en vez de que los usuarios tengan un nombre de conexión (login), pasan su tarjeta de identidad por un lector para identificarse a sí mismos. El requerimiento externo se deriva de la necesidad de que el sistema esté conforme con la legislación de privacidad. Evidentemente, la privacidad es un asunto muy importante en los sistemas de atención a la salud, y el requerimiento especifica que el sistema debe desarrollarse conforme a un estándar de privacidad nacional.

Un problema común con requerimientos no funcionales es que los usuarios o clientes con frecuencia proponen estos requerimientos como metas generales, como facilidad de uso, capacidad de que el sistema se recupere de fallas, o rapidez de respuesta al usuario. Las metas establecen buenas intenciones; no obstante, ocasionan problemas a los desarrolladores del sistema, pues dejan espacio para la interpretación y la disputa posterior una vez que se entregue el sistema. Por ejemplo, la siguiente meta del sistema es típica de cómo un administrador expresa los requerimientos de usabilidad:

> *Para el personal médico debe ser fácil usar el sistema, y este último debe organizarse de tal forma que minimice los errores del usuario.*

Lo anterior se escribió para mostrar cómo podría expresarse la meta como un requerimiento no funcional "comprobable". Aun cuando es imposible comprobar de manera objetiva la meta del sistema, en la siguiente descripción se puede incluir, al menos, la instrumentación de software para contar los errores cometidos por los usuarios cuando prueban el sistema.

> *Después de cuatro horas de capacitación, el personal médico usará todas las funciones del sistema. Después de esta capacitación, los usuarios experimentados no deberán superar el promedio de dos errores cometidos por hora de uso del sistema.*

Siempre que sea posible, se deberán escribir de manera cuantitativa los requerimientos no funcionales, de manera que puedan ponerse objetivamente a prueba. La figura 4.5 muestra las métricas que se utilizan para especificar propiedades no funcionales del sistema.

Propiedad	Medida
Rapidez	Transacciones/segundo procesadas Tiempo de respuesta usuario/evento Tiempo de regeneración de pantalla
Tamaño	Mbytes Número de chips ROM
Facilidad de uso	Tiempo de capacitación Número de cuadros de ayuda
Fiabilidad	Tiempo medio para falla Probabilidad de indisponibilidad Tasa de ocurrencia de falla Disponibilidad
Robustez	Tiempo de reinicio después de falla Porcentaje de eventos que causan falla Probabilidad de corrupción de datos en falla
Portabilidad	Porcentaje de enunciados dependientes de objetivo Número de sistemas objetivo

Figura 4.5 Métricas para especificar requerimientos no funcionales

Usted puede medir dichas características cuando el sistema se pone a prueba para comprobar si éste cumple o no cumple con sus requerimientos no funcionales.

En la práctica, los usuarios de un sistema suelen encontrar difícil traducir sus metas en requerimientos mensurables. Para algunas metas, como la mantenibilidad, no hay métricas para usarse. En otros casos, incluso cuando sea posible la especificación cuantitativa, los clientes no logran relacionar sus necesidades con dichas especificaciones. No comprenden qué significa algún número que define la fiabilidad requerida (por así decirlo), en términos de su experiencia cotidiana con los sistemas de cómputo. Más aún, el costo por verificar objetivamente los requerimientos no funcionales mensurables suele ser muy elevado, y los clientes que pagan por el sistema quizá piensen que dichos costos no están justificados.

Los requerimientos no funcionales entran a menudo en conflicto e interactúan con otros requerimientos funcionales o no funcionales. Por ejemplo, el requerimiento de autenticación en la figura 4.4 requiere, indiscutiblemente, la instalación de un lector de tarjetas en cada computadora unida al sistema. Sin embargo, podría haber otro requerimiento que solicite acceso móvil al sistema desde las computadoras portátiles de médicos o enfermeras. Por lo general, las computadoras portátiles no están equipadas con lectores de tarjeta, de modo que, ante tales circunstancias, probablemente deba permitirse algún método de autenticación alternativo.

En la práctica, en el documento de requerimientos, resulta difícil separar los requerimientos funcionales de los no funcionales. Si los requerimientos no funcionales se expresan por separado de los requerimientos funcionales, las relaciones entre ambos serían difíciles de entender. No obstante, se deben destacar de manera explícita los requerimientos que están claramente relacionados con las propiedades emergentes del sistema, como el rendimiento o la fiabilidad. Esto se logra al ponerlos en una sección separada del documento de requerimientos o al distinguirlos, en alguna forma, de otros requerimientos del sistema.

Estándares del documento de requerimientos

Algunas organizaciones grandes, como el Departamento de Defensa estadounidense y el Institute of Electrical and Electronic Engineers (IEEE), definieron estándares para los documentos de requerimientos. Comúnmente son muy genéricos, pero útiles como base para desarrollar estándares organizativos más detallados. El IEEE es uno de los proveedores de estándares mejor conocidos y desarrolló un estándar para la estructura de documentos de requerimientos. Este estándar es más adecuado para sistemas como comando militar y sistemas de control que tienen un largo tiempo de vida y, por lo general, los diseña un grupo de organizaciones.

http://www.SoftwareEngineering-9.com/Web/Requirements/IEEE-standard.html

Los requerimientos no funcionales, como los requerimientos de fiabilidad, protección y confidencialidad, son en particular importantes para los sistemas fundamentales. En el capítulo 12 se incluyen estos requerimientos, donde se describen técnicas específicas para definir requerimientos de confiabilidad y seguridad.

4.2 El documento de requerimientos de software

El documento de requerimientos de software (llamado algunas veces especificación de requerimientos de software o SRS) es un comunicado oficial de lo que deben implementar los desarrolladores del sistema. Incluye tanto los requerimientos del usuario para un sistema, como una especificación detallada de los requerimientos del sistema. En ocasiones, los requerimientos del usuario y del sistema se integran en una sola descripción. En otros casos, los requerimientos del usuario se definen en una introducción a la especificación de requerimientos del sistema. Si hay un gran número de requerimientos, los requerimientos del sistema detallados podrían presentarse en un documento aparte.

Son esenciales los documentos de requerimientos cuando un contratista externo diseña el sistema de software. Sin embargo, los métodos de desarrollo ágiles argumentan que los requerimientos cambian tan rápidamente que un documento de requerimientos se vuelve obsoleto tan pronto como se escribe, así que el esfuerzo se desperdicia en gran medida. En lugar de un documento formal, los enfoques como la programación extrema (Beck, 1999) recopilan de manera incremental requerimientos del usuario y los escriben en tarjetas como historias de usuario. De esa manera, el usuario da prioridad a los requerimientos para su implementación en el siguiente incremento del sistema.

Este enfoque es adecuado para sistemas empresariales donde los requerimientos son inestables. Sin embargo, aún resulta útil escribir un breve documento de apoyo que defina los requerimientos de la empresa y los requerimientos de confiabilidad para el sistema; es fácil olvidar los requerimientos que se aplican al sistema como un todo, cuando uno se enfoca en los requerimientos funcionales para la siguiente liberación del sistema.

El documento de requerimientos tiene un conjunto variado de usuarios, desde el administrador ejecutivo de la organización que paga por el sistema, hasta los ingenieros responsables del desarrollo del software. La figura 4.6, tomada del libro del autor con Gerald Kotonya sobre ingeniería de requerimientos (Kotonya y Sommerville, 1998), muestra a los posibles usuarios del documento y cómo ellos lo utilizan.

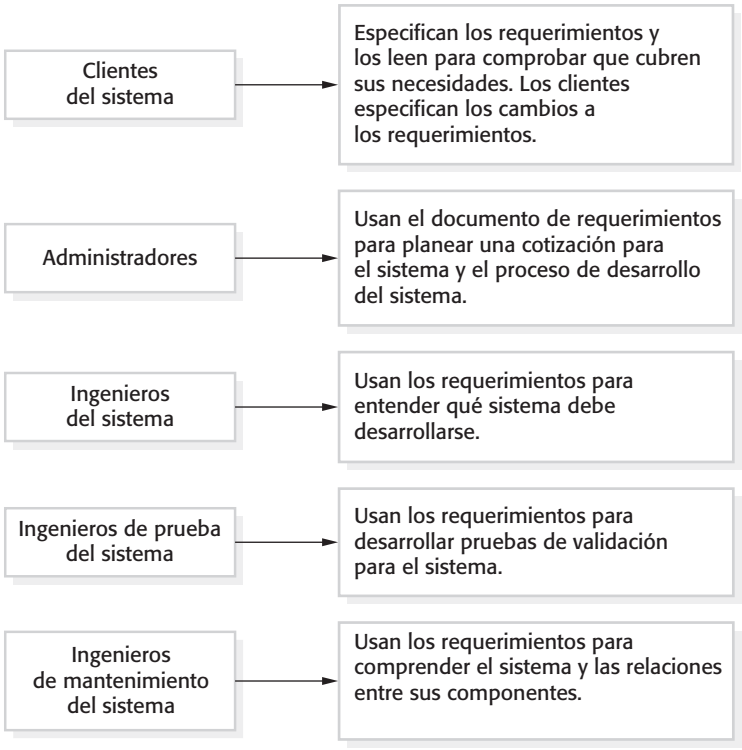


Figura 4.6 Usuarios de un documento de requerimientos

La diversidad de posibles usuarios significa que el documento de requerimientos debe ser un compromiso entre la comunicación de los requerimientos a los clientes, la definición de los requerimientos con detalle preciso para desarrolladores y examinadores, y la inclusión de información sobre la posible evolución del sistema. La información de cambios anticipados ayuda tanto a los diseñadores del sistema a evitar decisiones de diseño restrictivas, como a los ingenieros de mantenimiento del sistema que deben adaptar el sistema a los nuevos requerimientos.

El nivel de detalle que se incluya en un documento de requerimientos depende del tipo de sistema a diseñar y el proceso de desarrollo utilizado. Los sistemas críticos necesitan tener requerimientos detallados porque la seguridad y la protección también deben analizarse de forma pormenorizada. Cuando el sistema lo desarrolla una compañía independiente (por ejemplo, mediante la subcontratación), deben detallarse y precisarse las especificaciones del sistema. Si se utiliza un proceso de desarrollo iterativo interno, entonces el documento de requerimientos suele ser mucho menos detallado y cualquier ambigüedad puede resolverse durante el desarrollo del sistema.

La figura 4.7 indica una posible organización para un documento de requerimientos basada en un estándar del IEEE para documentos de requerimientos (IEEE, 1998). Este estándar es genérico y se adapta a usos específicos. En este caso, el estándar se extendió para incluir información de la evolución prevista del sistema. Esta información ayuda a los encargados del sistema y permite a los diseñadores incluir soporte para características futuras del sistema.

Naturalmente, la información que se incluya en un documento de requerimientos depende del tipo de software que se va a desarrollar y del enfoque para el desarrollo que se use. Si se adopta un enfoque evolutivo para un producto de software (por ejemplo), el

Capítulo	Descripción
Prefacio	Debe definir el número esperado de lectores del documento, así como describir su historia de versiones, incluidas las causas para la creación de una nueva versión y un resumen de los cambios realizados en cada versión.
Introducción	Describe la necesidad para el sistema. Debe detallar brevemente las funciones del sistema y explicar cómo funcionará con otros sistemas. También tiene que indicar cómo se ajusta el sistema en los objetivos empresariales o estratégicos globales de la organización que comisiona el software.
Glosario	Define los términos técnicos usados en el documento. No debe hacer conjeturas sobre la experiencia o la habilidad del lector.
Definición de requerimientos del usuario	Aquí se representan los servicios que ofrecen al usuario. También, en esta sección se describen los requerimientos no funcionales del sistema. Esta descripción puede usar lenguaje natural, diagramas u otras observaciones que sean comprensibles para los clientes. Deben especificarse los estándares de producto y proceso que tienen que seguirse.
Arquitectura del sistema	Este capítulo presenta un panorama de alto nivel de la arquitectura anticipada del sistema, que muestra la distribución de funciones a través de los módulos del sistema. Hay que destacar los componentes arquitectónicos que sean de reutilización.
Especificación de requerimientos del sistema	Debe representar los requerimientos funcionales y no funcionales con más detalle. Si es preciso, también pueden detallarse más los requerimientos no funcionales. Pueden definirse las interfaces a otros sistemas.
Modelos del sistema	Pueden incluir modelos gráficos del sistema que muestren las relaciones entre componentes del sistema, el sistema y su entorno. Ejemplos de posibles modelos son los modelos de objeto, modelos de flujo de datos o modelos de datos semánticos.
Evolución del sistema	Describe los supuestos fundamentales sobre los que se basa el sistema, y cualquier cambio anticipado debido a evolución de hardware, cambio en las necesidades del usuario, etc. Esta sección es útil para los diseñadores del sistema, pues los ayuda a evitar decisiones de diseño que restringirían probablemente futuros cambios al sistema.
Apéndices	Brindan información específica y detallada que se relaciona con la aplicación a desarrollar; por ejemplo, descripciones de hardware y bases de datos. Los requerimientos de hardware definen las configuraciones, mínima y óptima, del sistema. Los requerimientos de base de datos delimitan la organización lógica de los datos usados por el sistema y las relaciones entre datos.
Índice	Pueden incluirse en el documento varios índices. Así como un índice alfabético normal, uno de diagramas, un índice de funciones, etcétera.

Figura 4.7 Estructura de un documento de requerimientos

documento de requerimientos dejará fuera muchos de los capítulos detallados que se sugirieron anteriormente. El enfoque estará en especificar los requerimientos del usuario y los requerimientos no funcionales de alto nivel del sistema. En este caso, diseñadores y programadores usan su criterio para decidir cómo cubrir los requerimientos establecidos del usuario para el sistema.

Sin embargo, cuando el software sea parte de un proyecto de sistema grande que incluya la interacción de sistemas de hardware y software, será necesario por lo general

Problemas con el uso de lenguaje natural para la especificación de requerimientos

La flexibilidad del lenguaje natural, que es tan útil para la especificación, causa problemas frecuentemente. Hay espacio para escribir requerimientos poco claros, y los lectores (los diseñadores) pueden malinterpretar los requerimientos porque tienen un antecedente diferente al del usuario. Es fácil mezclar muchos requerimientos en una sola oración y quizá sea difícil estructurar los requerimientos en lenguaje natural.

http://www.SoftwareEngineering-9.com/Web/Requirements/NL-problems.html

definir los requerimientos a un nivel detallado. Esto significa que es probable que los documentos de requerimientos sean muy largos y deban incluir la mayoría, si no es que todos, los capítulos que se muestran en la figura 4.7. Para documentos extensos, es muy importante incluir una tabla de contenido global y un índice del documento, de manera que los lectores encuentren con facilidad la información que necesitan.

4.3 Especificación de requerimientos

La especificación de requerimientos es el proceso de escribir, en un documento de requerimientos, los requerimientos del usuario y del sistema. De manera ideal, los requerimientos del usuario y del sistema deben ser claros, sin ambigüedades, fáciles de entender, completos y consistentes. Esto en la práctica es difícil de lograr, pues los participantes interpretan los requerimientos de formas diferentes y con frecuencia en los requerimientos hay conflictos e inconsistencias inherentes.

Los requerimientos del usuario para un sistema deben describir los requerimientos funcionales y no funcionales, de forma que sean comprensibles para los usuarios del sistema que no cuentan con un conocimiento técnico detallado. De manera ideal, deberían especificar sólo el comportamiento externo del sistema. El documento de requerimientos no debe incluir detalles de la arquitectura o el diseño del sistema. En consecuencia, si usted escribe los requerimientos del usuario, no tiene que usar jerga de software, anotaciones estructuradas o formales. Debe escribir los requerimientos del usuario en lenguaje natural, con tablas y formas sencillas, así como diagramas intuitivos.

Los requerimientos del sistema son versiones extendidas de los requerimientos del usuario que los ingenieros de software usan como punto de partida para el diseño del sistema. Añaden detalles y explican cómo el sistema debe brindar los requerimientos del usuario. Se pueden usar como parte del contrato para la implementación del sistema y, por lo tanto, deben ser una especificación completa y detallada de todo el sistema.

Idealmente, los requerimientos del sistema deben describir de manera simple el comportamiento externo del sistema y sus restricciones operacionales. No tienen que ocuparse de cómo se diseña o implementa el sistema. Sin embargo, al nivel de detalle requerido para especificar por completo un sistema de software complejo, es prácticamente imposible excluir toda la información de diseño. Para ello existen varias razones:

1. Tal vez se tenga que diseñar una arquitectura inicial del sistema para ayudar a estructurar la especificación de requerimientos. Los requerimientos del sistema se organizan

Notación	Descripción
Enunciados en lenguaje natural	Los requerimientos se escriben al usar enunciados numerados en lenguaje natural. Cada enunciado debe expresar un requerimiento.
Lenguaje natural estructurado	Los requerimientos se escriben en lenguaje natural en una forma o plantilla estándar. Cada campo ofrece información de un aspecto del requerimiento.
Lenguajes de descripción de diseño	Este enfoque usa un lenguaje como un lenguaje de programación, pero con características más abstractas para especificar los requerimientos al definir un modelo operacional del sistema. Aunque en la actualidad este enfoque se usa raras veces, aún tiene utilidad para especificaciones de interfaz.
Anotaciones gráficas	Los modelos gráficos, complementados con anotaciones de texto, sirven para definir los requerimientos funcionales del sistema; los casos de uso del UML y los diagramas de secuencia se emplean de forma común.
Especificaciones matemáticas	Dichas anotaciones se basan en conceptos matemáticos como máquinas o conjuntos de estado finito. Aunque tales especificaciones sin ambigüedades pueden reducir la imprecisión en un documento de requerimientos, la mayoría de los clientes no comprenden una especificación formal. No pueden comprobar que representa lo que quieren y por ello tienen reticencia para aceptarlo como un contrato de sistema.

Figura 4.8 Formas de escribir una especificación de requerimientos del sistema

de acuerdo con los diferentes subsistemas que constituyen el sistema. Como veremos en los capítulos 6 y 18, esta definición arquitectónica es esencial si usted quiere reutilizar componentes de software al implementar el sistema.

2. En la mayoría de los casos, los sistemas deben interoperar con los sistemas existentes, lo cual restringe el diseño e impone requerimientos sobre el nuevo sistema.
3. Quizá sea necesario el uso de una arquitectura específica para cubrir los requerimientos no funcionales (como la programación N-versión para lograr fiabilidad, que se estudia en el capítulo 13). Un regulador externo, que precise certificar que dicho sistema es seguro, puede especificar que se utilice un diseño arquitectónico ya avalado.

Los requerimientos del usuario se escriben casi siempre en lenguaje natural, complementado con diagramas y tablas adecuados en el documento de requerimientos. Los requerimientos del sistema se escriben también en lenguaje natural, pero de igual modo se utilizan otras notaciones basadas en formas, modelos gráficos del sistema o modelos matemáticos del sistema. La figura 4.8 resume las posibles anotaciones que podrían usarse para escribir requerimientos del sistema.

Los modelos gráficos son más útiles cuando es necesario mostrar cómo cambia un estado o al describir una secuencia de acciones. Los gráficos de secuencia UML y los gráficos de estado, que se explican en el capítulo 5, exponen la secuencia de acciones que ocurren en respuesta a cierto mensaje o evento. En ocasiones, se usan especificaciones matemáticas formales con la finalidad de describir los requerimientos para sistemas de protección o seguridad críticos, aunque rara vez se usan en otras circunstancias. Este enfoque para escribir especificaciones se explica en el capítulo 12.

3.2 Si se requiere, cada 10 minutos el sistema medirá el azúcar en la sangre y administrará insulina. (*Los cambios de azúcar en la sangre son relativamente lentos, de manera que no son necesarias mediciones más frecuentes; la medición menos periódica podría conducir a niveles de azúcar innecesariamente elevados.*)

3.6 Cada minuto, el sistema debe correr una rutina de autoevaluación, con las condiciones a probar y las acciones asociadas definidas en la tabla 1. (*Una rutina de autoevaluación puede detectar problemas de hardware y software, y prevenir al usuario sobre el hecho de que la operación normal puede ser imposible.*)

Figura 4.9 Ejemplo de requerimientos para el sistema de software de la bomba de insulina

4.3.1 Especificación en lenguaje natural

Desde los albores de la ingeniería de software, el lenguaje natural se usa para escribir los requerimientos de software. Es expresivo, intuitivo y universal. También es potencialmente vago, ambiguo y su significado depende de los antecedentes del lector. Como resultado, hay muchas propuestas para formas alternativas de escribir los requerimientos. Sin embargo, ninguna se ha adoptado de manera amplia, por lo que el lenguaje natural seguirá siendo la forma más usada para especificar los requerimientos del sistema y del software.

Para minimizar la interpretación errónea al escribir los requerimientos en lenguaje natural, se recomienda seguir algunos lineamientos sencillos:

1. Elabore un formato estándar y asegúrese de que todas las definiciones de requerimientos se adhieran a dicho formato. Al estandarizar el formato es menos probable cometer omisiones y más sencillo comprobar los requerimientos. El formato que usa el autor expresa el requerimiento en una sola oración. A cada requerimiento de usuario se asocia un enunciado de razones para explicar por qué se propuso el requerimiento. Las razones también pueden incluir información sobre quién planteó el requerimiento (la fuente del requerimiento), de modo que usted conozca a quién consultar en caso de que cambie el requerimiento.
2. Utilice el lenguaje de manera clara para distinguir entre requerimientos obligatorios y deseables. Los primeros son requerimientos que el sistema debe soportar y, por lo general, se escriben en futuro "debe ser". En tanto que los requerimientos deseables no son necesarios y se escriben en tiempo pospretérito o como condicional "debería ser".
3. Use texto resaltado (negrilla, cursiva o color) para seleccionar partes clave del requerimiento.
4. No deduzca que los lectores entienden el lenguaje técnico de la ingeniería de software. Es fácil que se malinterpreten palabras como "arquitectura" y "módulo". Por lo tanto, debe evitar el uso de jerga, abreviaturas y acrónimos.
5. Siempre que sea posible, asocie una razón con cada requerimiento de usuario. La razón debe explicar por qué se incluyó el requerimiento. Es particularmente útil cuando los requerimientos cambian, pues ayuda a decidir cuáles cambios serían indeseables.

La figura 4.9 ilustra cómo se usan dichos lineamientos. Incluye dos requerimientos para el software embebido para la bomba de insulina automatizada, que se introdujo en el capítulo 1. Usted puede descargar la especificación completa de los requerimientos de la bomba de insulina en las páginas Web del libro.

Bomba de insulina/Software de control/SRS/3.3.2

Función	Calcula dosis de insulina: nivel seguro de azúcar.
Descripción	Calcula la dosis de insulina que se va a suministrar cuando la medición del nivel de azúcar actual esté en zona segura entre 3 y 7 unidades.
Entradas	Lectura del azúcar actual (r2), las dos lecturas previas (r0 y r1).
Fuente	Lectura del azúcar actual del sensor. Otras lecturas de la memoria.
Salidas	CompDose: la dosis de insulina a administrar.
Destino	Ciclo de control principal.
Acción	CompDose es cero si es estable el nivel de azúcar, o cae o si aumenta el nivel pero disminuye la tasa de aumento. Si el nivel se eleva y la tasa de aumento crece, CompDose se calcula entonces al dividir la diferencia entre el nivel de azúcar actual y el nivel previo entre 4 y redondear el resultado. Si la suma se redondea a cero, en tal caso CompDose se establece en la dosis mínima que puede entregarse.
Requerimientos	Dos lecturas previas, de modo que puede calcularse la tasa de cambio del nivel de azúcar.
Precondición	El depósito de insulina contiene al menos la dosis individual de insulina máxima permitida.
Postcondición	r0 se sustituye con r1, luego r1 se sustituye con r2.
Efectos colaterales	Ninguno.

Figura 4.10 Especificación estructurada de un requerimiento para una bomba de insulina

4.3.2 Especificaciones estructuradas

El lenguaje natural estructurado es una manera de escribir requerimientos del sistema, donde está limitada la libertad del escritor de requerimientos y todos éstos se anotan en una forma estándar. Aunque este enfoque conserva la mayoría de la expresividad y comprensibilidad del lenguaje natural, asegura que haya cierta uniformidad sobre la especificación. Las anotaciones en lenguaje estructurado emplean plantillas para especificar requerimientos del sistema. La especificación utiliza constructos de lenguaje de programación para mostrar alternativas e iteración, y destaca elementos clave con el uso de sombreado o de fuentes distintas.

Los Robertson (Robertson y Robertson, 1999), en su libro del método de ingeniería de requerimientos VOLERE, recomiendan que se escriban los requerimientos del usuario inicialmente en tarjetas, un requerimiento por tarjeta. Proponen algunos campos en cada tarjeta, tales como razones de los requerimientos, dependencias en otros requerimientos, fuente de los requerimientos, materiales de apoyo, etcétera. Lo anterior es similar al enfoque utilizado en el ejemplo de la especificación estructurada que se muestra en la figura 4.10.

Para usar un enfoque estructurado que especifique los requerimientos de sistema, hay que definir una o más plantillas estándar para requerimientos, y representar dichas plantillas como formas estructuradas. La especificación puede estructurarse sobre los objetos manipulados por el sistema, las funciones que el sistema realiza o los eventos procesados por el sistema. En la figura 4.10 se muestra un ejemplo de una especificación basada en la forma, en este caso, una que define cómo calcular la dosis de insulina a administrar cuando el azúcar en la sangre está dentro de una banda segura.

Condición	Acción
Nivel de azúcar en descenso (r2 < r1)	CompDose = 0
Nivel de azúcar estable (r2 = r1)	CompDose = 0
Nivel de azúcar creciente y tasa de incremento decreciente ((r2 − r1) < (r1 − r0))	CompDose = 0
Nivel de azúcar creciente y tasa de incremento estable o creciente ((r2 − r1) ≥ (r1 − r0))	CompDose = round ((r2 − r1)/4) If resultado redondeado = 0 then CompDose = MinimumDose

Figura 4.11 Especificación tabular del cálculo para una bomba de insulina

Cuando use una forma estándar para especificar requerimientos funcionales, debe incluir la siguiente información:

1. Una descripción de la función o entidad a especificar.
2. Una descripción de sus entradas y sus procedencias.
3. Una descripción de sus salidas y a dónde se dirigen.
4. Información sobre los datos requeridos para el cálculo u otras entidades en el sistema que se utilizan (la parte "requiere").
5. Una descripción de la acción que se va a tomar.
6. Si se usa un enfoque funcional, una precondición establece lo que debe ser verdadero antes de llamar a la función, y una postcondición especifica lo que es verdadero después de llamar a la función.
7. Una descripción de los efectos colaterales (si acaso hay alguno) de la operación.

Al usar especificaciones estructuradas se eliminan algunos de los problemas de la especificación en lenguaje natural. La variabilidad en la especificación se reduce y los requerimientos se organizan de forma más efectiva. Sin embargo, en ocasiones todavía es difícil escribir requerimientos sin ambigüedades, en particular cuando deben especificarse cálculos complejos (por ejemplo, cómo calcular la dosis de insulina).

Para enfrentar este problema se puede agregar información extra a los requerimientos en lenguaje natural, por ejemplo, con el uso de tablas o modelos gráficos del sistema. Éstos pueden mostrar cómo proceden los cálculos, cambia el estado del sistema, interactúan los usuarios con el sistema y se realizan las secuencias de acciones.

Las tablas son particularmente útiles cuando hay algunas posibles situaciones alternas y se necesita describir las acciones a tomar en cada una de ellas. La bomba de insulina fundamenta sus cálculos del requerimiento de insulina, en la tasa de cambio de los niveles de azúcar en la sangre. Las tasas de cambio se calculan con las lecturas, actual y anterior. La figura 4.11 es una descripción tabular de cómo se usa la tasa de cambio del azúcar en la sangre, para calcular la cantidad de insulina por suministrar.

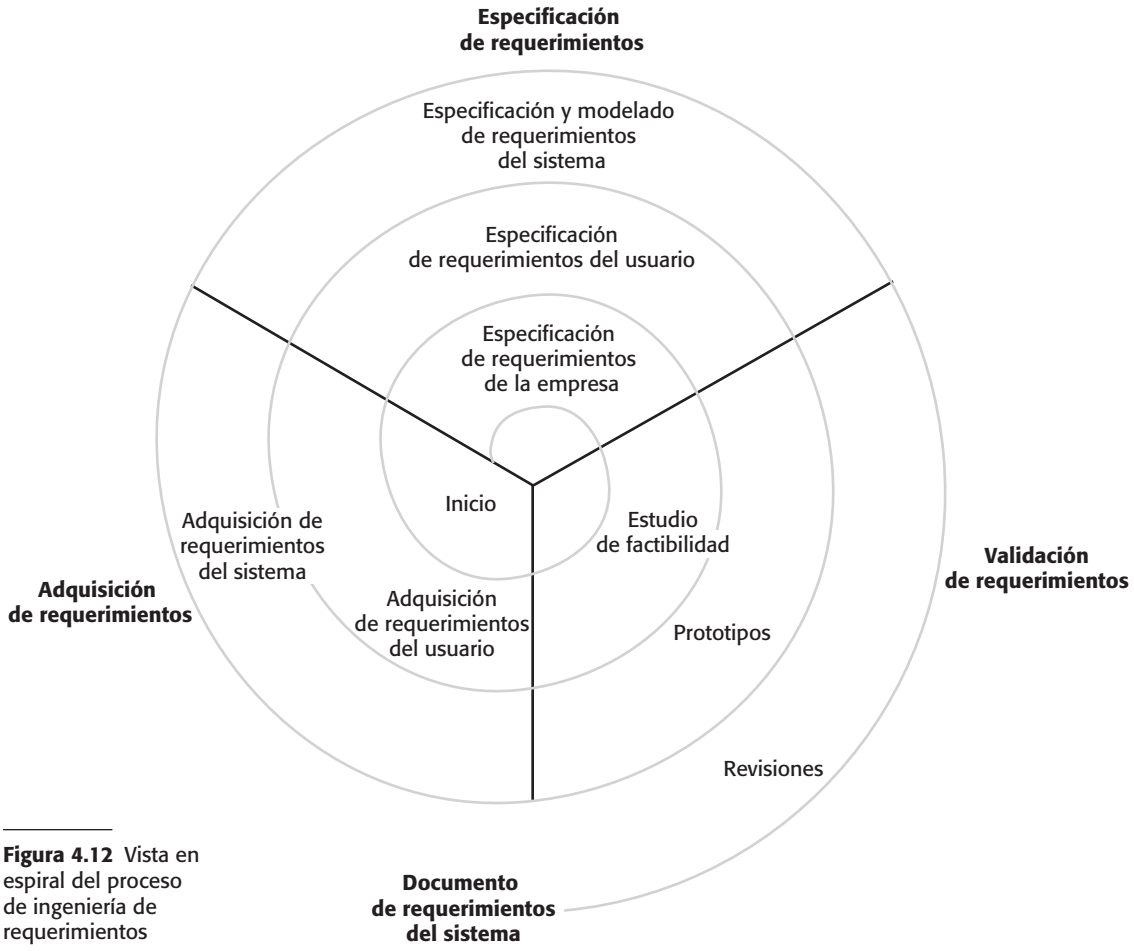


Figura 4.12 Vista en espiral del proceso de ingeniería de requerimientos

4.4 Procesos de ingeniería de requerimientos

Como vimos en el capítulo 2, los procesos de ingeniería de requerimientos incluyen cuatro actividades de alto nivel. Éstas se enfocan en valorar si el sistema es útil para la empresa (estudio de factibilidad), descubrir requerimientos (adquisición y análisis), convertir dichos requerimientos en alguna forma estándar (especificación) y comprobar que los requerimientos definan realmente el sistema que quiere el cliente (validación). En la figura 2.6 se mostró esto como proceso secuencial; sin embargo, en la práctica, la ingeniería de requerimientos es un proceso iterativo donde las actividades están entrelazadas.

La figura 4.12 presenta este entrelazamiento. Las actividades están organizadas como un proceso iterativo alrededor de una espiral, y la salida es un documento de requerimientos del sistema. La cantidad de tiempo y esfuerzo dedicados a cada actividad en cada iteración depende de la etapa del proceso global y el tipo de sistema que está siendo desarrollado. En el inicio del proceso, se empleará más esfuerzo para comprender los requerimientos empresariales de alto nivel y los no funcionales, así como los requerimientos del

Estudios de factibilidad

Un estudio de factibilidad es un breve estudio enfocado que debe realizarse con oportunidad en el proceso de IR. Debe responder tres preguntas clave: *a*) ¿El sistema contribuye con los objetivos globales de la organización? *b*) ¿El sistema puede implementarse dentro de la fecha y el presupuesto usando la tecnología actual? *c*) ¿El sistema puede integrarse con otros sistemas que se utilicen?

Si la respuesta a cualquiera de estas preguntas es negativa, probablemente no sea conveniente continuar con el proyecto.

http://www.SoftwareEngineering-9.com/Web/Requirements/FeasibilityStudy.html

usuario para el sistema. Más adelante en el proceso, en los anillos exteriores de la espiral, se dedicará más esfuerzo a la adquisición y comprensión de los requerimientos detallados del sistema.

Este modelo en espiral acomoda enfoques al desarrollo, donde los requerimientos se elaboraron con diferentes niveles de detalle. El número de iteraciones de la espiral tiende a variar, de modo que la espiral terminará después de adquirir algunos o todos los requerimientos del usuario. Se puede usar el desarrollo ágil en vez de la creación de prototipos, de manera que se diseñen en conjunto los requerimientos y la implementación del sistema.

Algunas personas consideran la ingeniería de requerimientos como el proceso de aplicar un método de análisis estructurado, tal como el análisis orientado a objetos (Larman, 2002). Esto implica analizar el sistema y desarrollar un conjunto de modelos gráficos del sistema, como los modelos de caso de uso, que luego sirven como especificación del sistema. El conjunto de modelos describe el comportamiento del sistema y se anota con información adicional que describe, por ejemplo, el rendimiento o la fiabilidad requeridos del sistema.

Aunque los métodos estructurados desempeñan un papel en el proceso de ingeniería de requerimientos, hay mucho más ingeniería de requerimientos de la que se cubre con dichos métodos. La adquisición de requerimientos, en particular, es una actividad centrada en la gente, y a las personas no les gustan las restricciones impuestas por modelos de sistema rígidos.

Prácticamente en todos los sistemas cambian los requerimientos. Las personas implicadas desarrollan una mejor comprensión de qué quieren que haga el software; la organización que compra el sistema cambia; se hacen modificaciones al hardware, al software y al entorno organizacional del sistema. Al proceso de administrar tales requerimientos cambiantes se le llama administración de requerimientos, tema que se trata en la sección 4.7.

4.5 Adquisición y análisis de requerimientos

Después de un estudio de factibilidad inicial, la siguiente etapa del proceso de ingeniería de requerimientos es la adquisición y el análisis de requerimientos. En esta actividad, los ingenieros de software trabajan con clientes y usuarios finales del sistema para descubrir el dominio de aplicación, qué servicios debe proporcionar el sistema, el desempeño requerido de éste, las restricciones de hardware, etcétera.

Figura 4.13 El proceso de adquisición y análisis de requerimientos

En una organización, la adquisición y el análisis de requerimientos pueden involucrar a diversas clases de personas. Un participante en el sistema es quien debe tener alguna influencia directa o indirecta sobre los requerimientos del mismo. Los participantes incluyen a usuarios finales que interactuarán con el sistema, y a cualquiera en una organización que resultará afectada por él. Otros participantes del sistema pueden ser los ingenieros que desarrollan o mantienen otros sistemas relacionados, administradores de negocios, expertos de dominio y representantes de asociaciones sindicales.

En la figura 4.13 se muestra un modelo del proceso de adquisición y análisis. Cada organización tendrá su versión o ejemplificación de este modelo general, dependiendo de factores locales, tales como experiencia del personal, tipo de sistema a desarrollar, estándares usados, etcétera.

Las actividades del proceso son:

1. *Descubrimiento de requerimientos* Éste es el proceso de interactuar con los participantes del sistema para descubrir sus requerimientos. También los requerimientos de dominio de los participantes y la documentación se descubren durante esta actividad. Existen numerosas técnicas complementarias que pueden usarse para el descubrimiento de requerimientos, las cuales se estudian más adelante en esta sección.
2. *Clasificación y organización de requerimientos* Esta actividad toma la compilación no estructurada de requerimientos, agrupa requerimientos relacionados y los organiza en grupos coherentes. La forma más común de agrupar requerimientos es usar un modelo de la arquitectura del sistema, para identificar subsistemas y asociar los requerimientos con cada subsistema. En la práctica, la ingeniería de requerimientos y el diseño arquitectónico no son actividades separadas completamente.
3. *Priorización y negociación de requerimientos* Inevitablemente, cuando intervienen diversos participantes, los requerimientos entrarán en conflicto. Esta actividad se preocupa por priorizar los requerimientos, así como por encontrar y resolver conflictos de requerimientos mediante la negociación. Por lo general, los participantes tienen que reunirse para resolver las diferencias y estar de acuerdo con el compromiso de los requerimientos.

4. *Especificación de requerimientos* Los requerimientos se documentan e ingresan en la siguiente ronda de la espiral. Pueden producirse documentos de requerimientos formales o informales, como se estudia en la sección 4.3.

La figura 4.13 muestra que la adquisición y el análisis de requerimientos es un proceso iterativo con retroalimentación continua de cada actividad a otras actividades. El ciclo del proceso comienza con el descubrimiento de requerimientos y termina con la documentación de los requerimientos. La comprensión de los requerimientos por parte del analista mejora con cada ronda del ciclo. El ciclo concluye cuando está completo el documento de requerimientos.

La adquisición y la comprensión de los requerimientos por parte de los participantes del sistema es un proceso difícil por diferentes razones:

1. Los participantes con frecuencia no saben lo que quieren de un sistema de cómputo, excepto en términos muy generales; pueden encontrar difícil articular qué quieren que haga el sistema; pueden hacer peticiones inalcanzables porque no saben qué es factible y qué no lo es.
2. Los participantes en un sistema expresan naturalmente los requerimientos con sus términos y conocimientos implícitos de su trabajo. Los ingenieros de requerimientos, sin experiencia en el dominio del cliente, podrían no entender dichos requerimientos.
3. Diferentes participantes tienen distintos requerimientos y pueden expresarlos en variadas formas. Los ingenieros de requerimientos deben descubrir todas las fuentes potenciales de requerimientos e identificar similitudes y conflictos.
4. Factores políticos llegan a influir en los requerimientos de un sistema. Los administradores pueden solicitar requerimientos específicos del sistema, porque éstos les permitirán aumentar su influencia en la organización.
5. El ambiente económico y empresarial donde ocurre el análisis es dinámico. Inevitablemente cambia durante el proceso de análisis. Puede cambiar la importancia de requerimientos particulares; o bien, tal vez surjan nuevos requerimientos de nuevos participantes a quienes no se consultó originalmente.

Resulta ineludible que diferentes participantes tengan diversas visiones de la importancia y prioridad de los requerimientos y, algunas veces, dichas visiones están en conflicto. Durante el proceso, usted deberá organizar negociaciones regulares con los participantes, de forma que se alcancen compromisos. Es imposible complacer por completo a cada participante, pero, si algunos suponen que sus visiones no se consideraron de forma adecuada, quizás intenten deliberadamente socavar el proceso de IR.

En la etapa de especificación de requerimientos, los requerimientos adquiridos hasta el momento se documentan de tal forma que puedan usarse para ayudar al hallazgo de requerimientos. En esta etapa, podría generarse una primera versión del documento de requerimientos del sistema, con secciones faltantes y requerimientos incompletos. De modo alternativo, los requerimientos pueden documentarse en una forma completamente diferente (por ejemplo, en una hoja de cálculo o en tarjetas). Escribir requerimientos en tarjetas suele ser muy efectivo, ya que los participantes las administran, cambian y organizan con facilidad.

Puntos de vista

Un punto de vista es una forma de recopilar y organizar un conjunto de requerimientos de un grupo de participantes que cuentan con algo en común. Por lo tanto, cada punto de vista incluye una serie de requerimientos del sistema. Los puntos de vista pueden provenir de usuarios finales, administradores, etcétera. Ayudan a identificar a los individuos que brindan información sobre sus requerimientos y a estructurar los requerimientos para análisis.

http://www.SoftwareEngineering-9.com/Web/Requirements/Viewpoints.html

4.5.1 Descubrimiento de requerimientos

El descubrimiento de requerimientos (llamado a veces adquisición de requerimientos) es el proceso de recopilar información sobre el sistema requerido y los sistemas existentes, así como de separar, a partir de esta información, los requerimientos del usuario y del sistema. Las fuentes de información durante la fase de descubrimiento de requerimientos incluyen documentación, participantes del sistema y especificaciones de sistemas similares. La interacción con los participantes es a través de entrevistas y observaciones, y pueden usarse escenarios y prototipos para ayudar a los participantes a entender cómo será el sistema.

Los participantes varían desde administradores y usuarios finales de un sistema hasta participantes externos como los reguladores, quienes certifican la aceptabilidad del sistema. Por ejemplo, los participantes que se incluyen para el sistema de información de pacientes en atención a la salud mental son:

1. Pacientes cuya información se registra en el sistema.
2. Médicos que son responsables de valorar y tratar a los pacientes.
3. Enfermeros que coordinan, junto con los médicos, las consultas y suministran algunos tratamientos.
4. Recepcionistas que administran las citas médicas de los pacientes.
5. Personal de TI que es responsable de instalar y mantener el sistema.
6. Un director de ética médica que debe garantizar que el sistema cumpla con los lineamientos éticos actuales de la atención al paciente.
7. Encargados de atención a la salud que obtienen información administrativa del sistema.
8. Personal de archivo médico que es responsable de garantizar que la información del sistema se conserve, y se implementen de manera adecuada los procedimientos de mantenimiento del archivo.

Además de los participantes del sistema, se observa que los requerimientos también pueden venir del dominio de aplicación y de otros sistemas que interactúan con el sistema a especificar. Todos ellos deben considerarse durante el proceso de adquisición de requerimientos.

Todas estas diferentes fuentes de requerimientos (participantes, dominio, sistemas) se representan como puntos de vista del sistema, y cada visión muestra un subconjunto de los

requerimientos para el sistema. Diferentes puntos de vista de un problema enfocan el problema de diferentes formas. Sin embargo, sus perspectivas no son totalmente independientes, sino que por lo general se traslapan, de manera que tienen requerimientos comunes. Usted puede usar estos puntos de vista para estructurar tanto el descubrimiento como la documentación de los requerimientos del sistema.

4.5.2 Entrevistas

Las entrevistas formales o informales con participantes del sistema son una parte de la mayoría de los procesos de ingeniería de requerimientos. En estas entrevistas, el equipo de ingeniería de requerimientos formula preguntas a los participantes sobre el sistema que actualmente usan y el sistema que se va a desarrollar. Los requerimientos se derivan de las respuestas a dichas preguntas. Las entrevistas son de dos tipos:

1. Entrevistas cerradas, donde los participantes responden a un conjunto de preguntas preestablecidas.
2. Entrevistas abiertas, en las cuales no hay agenda predefinida. El equipo de ingeniería de requerimientos explora un rango de conflictos con los participantes del sistema y, como resultado, desarrolla una mejor comprensión de sus necesidades.

En la práctica, las entrevistas con los participantes son por lo general una combinación de ambas. Quizá se deba obtener la respuesta a ciertas preguntas, pero eso a menudo conduce a otros temas que se discuten en una forma menos estructurada. Rara vez funcionan bien las discusiones completamente abiertas. Con frecuencia debe plantear algunas preguntas para comenzar y mantener la entrevista enfocada en el sistema que se va a desarrollar.

Las entrevistas son valiosas para lograr una comprensión global sobre qué hacen los participantes, cómo pueden interactuar con el nuevo sistema y las dificultades que enfrentan con los sistemas actuales. A las personas les gusta hablar acerca de sus trabajos, así que por lo general están muy dispuestas a participar en entrevistas. Sin embargo, las entrevistas no son tan útiles para comprender los requerimientos desde el dominio de la aplicación.

Por dos razones resulta difícil asimilar el conocimiento de dominio a través de entrevistas:

1. Todos los especialistas en la aplicación usan terminología y jerga que son específicos de un dominio. Es imposible que ellos discutan los requerimientos de dominio sin usar este tipo de lenguaje. Por lo general, usan la terminología en una forma precisa y sutil, que para los ingenieros de requerimientos es fácil de malinterpretar.
2. Cierto conocimiento del dominio es tan familiar a los participantes que encuentran difícil de explicarlo, o bien, creen que es tan fundamental que no vale la pena mencionarlo. Por ejemplo, para un bibliotecario no es necesario decir que todas las adquisiciones deben catalogarse antes de agregarlas al acervo. Sin embargo, esto quizá no sea obvio para el entrevistador y, por lo tanto, es posible que no lo tome en cuenta en los requerimientos.

Las entrevistas tampoco son una técnica efectiva para adquirir conocimiento sobre los requerimientos y las restricciones de la organización, porque existen relaciones sutiles de poder entre los diferentes miembros en la organización. Las estructuras publicadas de

la organización rara vez coinciden con la realidad de la toma de decisiones en una organización, pero los entrevistados quizá no deseen revelar a un extraño la estructura real, sino la teórica. En general, la mayoría de las personas se muestran renuentes a discutir los conflictos políticos y organizacionales que afecten los requerimientos.

Los entrevistadores efectivos poseen dos características:

1. Tienen mentalidad abierta, evitan ideas preconcebidas sobre los requerimientos y escuchan a los participantes. Si el participante aparece con requerimientos sorprendentes, entonces tienen disposición para cambiar su mentalidad acerca del sistema.
2. Instan al entrevistado con una pregunta de trampolín para continuar la plática, dar una propuesta de requerimientos o trabajar juntos en un sistema de prototipo. Cuando se pregunta al individuo "dime qué quieres" es improbable que alguien consiga información útil. Encuentran mucho más sencillo hablar en un contexto definido que en términos generales.

La información de las entrevistas se complementa con otra información del sistema de documentación que describe los procesos empresariales o los sistemas existentes, las observaciones del usuario, etcétera. En ocasiones, además de los documentos del sistema, la información de la entrevista puede ser la única fuente de datos sobre los requerimientos del sistema. Sin embargo, la entrevista por sí misma está expuesta a perder información esencial y, por consiguiente, debe usarse junto con otras técnicas de adquisición de requerimientos.

4.5.3 Escenarios

Por lo general, las personas encuentran más sencillo vincularse con ejemplos reales que con descripciones abstractas. Pueden comprender y criticar un escenario sobre cómo interactuar con un sistema de software. Los ingenieros de requerimientos usan la información obtenida de esta discusión para formular los verdaderos requerimientos del sistema.

Los escenarios son particularmente útiles para detallar un bosquejo de descripción de requerimientos. Se trata de ejemplos sobre descripciones de sesiones de interacción. Cada escenario abarca comúnmente una interacción o un número pequeño de interacciones posibles. Se desarrollan diferentes formas de escenarios y se ofrecen varios tipos de información con diversos niveles de detalle acerca del sistema. Las historias que se usan en programación extrema, estudiadas en el capítulo 3, son un tipo de escenario de requerimientos.

Un escenario comienza con un bosquejo de la interacción. Durante el proceso de adquisición, se suman detalles a éste para crear una representación completa de dicha interacción. En su forma más general, un escenario puede incluir:

1. Una descripción de qué esperan el sistema y los usuarios cuando inicia el escenario.
2. Una descripción en el escenario del flujo normal de los eventos.
3. Una descripción de qué puede salir mal y cómo se manejaría.
4. Información de otras actividades que estén en marcha al mismo tiempo.
5. Una descripción del estado del sistema cuando termina el escenario.

SUPOSICIÓN INICIAL:
El paciente observa a un auxiliar médico que elabora un registro en el sistema y recaba información personal de aquél (nombre, dirección, edad, etcétera). Una enfermera ingresa en el sistema y obtiene la historia médica.

NORMAL:
La enfermera busca al paciente por su nombre completo. Si hay más de un paciente con el mismo apellido, para identificarlo se usa el nombre y la fecha de nacimiento.

La enfermera elige la opción de menú y añade la historia médica.

Inmediatamente la enfermera sigue una serie de indicadores (*prompt*) del sistema para ingresar información de consultas en otras instituciones, sobre problemas de salud mental (entrada libre de texto), condiciones médicas existentes (la enfermera selecciona las condiciones del menú), medicamentos administrados actualmente (seleccionados del menú), alergias (texto libre) y vida familiar (formato).

QUÉ PUEDE SALIR MAL:
Si no existe el registro del paciente o no puede encontrarse, la enfermera debe crear un nuevo registro e ingresar información personal.

Las condiciones o los medicamentos del paciente no se ingresan en el menú. La enfermera debe elegir la opción "otro" e ingresar texto libre que describa la condición/medicamento.

El paciente no puede/no proporciona información acerca de su historia médica. La enfermera tiene que ingresar a texto libre que registre la incapacidad/renuencia a brindar información. El sistema debe imprimir el formato de exclusión estándar que menciona que la falta de información podría significar que el tratamiento esté limitado o demorado. Esto tiene que firmarlo el paciente.

OTRAS ACTIVIDADES:
Mientras se ingresa la información, otros miembros del personal pueden consultar los registros, pero no editarlos.

ESTADO DEL SISTEMA A COMPLETAR:
Ingreso del usuario. El registro del paciente, incluida su historia médica, se integra en la base de datos, se agrega un registro a la bitácora (log) del sistema que indica el tiempo de inicio y terminación de la sesión y la enfermera a cargo.

Figura 4.14 Escenario para recabar historia médica en MHC-PMS

La adquisición basada en escenario implica trabajar con los participantes para identificar escenarios y captar detalles a incluir en dichos escenarios. Estos últimos pueden escribirse como texto, complementarse con diagramas, tomas de pantallas, etcétera. De forma alternativa, es posible usar un enfoque más estructurado, como los escenarios de evento o casos de uso.

Como ejemplo de un simple escenario de texto, considere cómo usaría el MHC-PMS para ingresar datos de un nuevo paciente (figura 4.14). Cuando un nuevo paciente asiste a una clínica, un auxiliar médico crea un nuevo registro y agrega información personal (nombre, edad, etcétera). Después, una enfermera entrevista al paciente y recaba su historia médica. Luego, el paciente tiene una consulta inicial con un médico que lo diagnostica y, si es adecuado, recomienda un tratamiento. El escenario muestra lo que sucede cuando se recaba la historia médica.

4.5.4 Casos de uso

Los casos de uso son una técnica de descubrimiento de requerimientos que se introdujo por primera vez en el método Objectory (Jacobson *et al*., 1993). Ahora se ha convertido

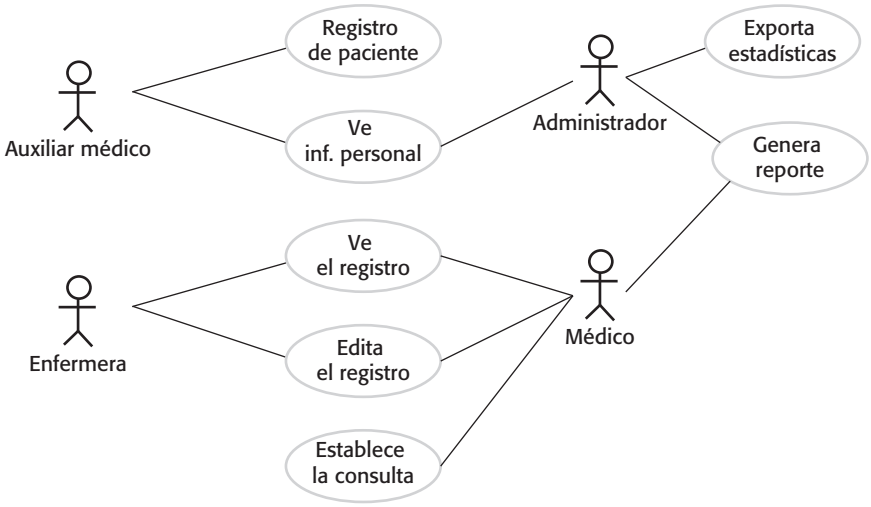


Figura 4.15 Casos de uso para el MHC-PMS

en una característica fundamental del modelado de lenguaje unificado. En su forma más sencilla, un caso de uso identifica a los actores implicados en una interacción, y nombra el tipo de interacción. Entonces, esto se complementa con información adicional que describe la interacción con el sistema. La información adicional puede ser una descripción textual, o bien, uno o más modelos gráficos como una secuencia UML o un gráfico de estado.

Los casos de uso se documentan con el empleo de un diagrama de caso de uso de alto nivel. El conjunto de casos de uso representa todas las interacciones posibles que se describirán en los requerimientos del sistema. Los actores en el proceso, que pueden ser individuos u otros sistemas, se representan como figuras sencillas. Cada clase de interacción se constituye como una elipse con etiqueta. Líneas vinculan a los actores con la interacción. De manera opcional, se agregan puntas de flecha a las líneas para mostrar cómo se inicia la interacción. Esto se ilustra en la figura 4.15, que presenta algunos de los casos de uso para el sistema de información del paciente.

No hay distinción tajante y rápida entre escenarios y casos de uso. Algunas personas consideran que cada caso de uso es un solo escenario; otras, como sugieren Stevens y Pooley (2006), encapsulan un conjunto de escenarios en un solo caso de uso. Cada escenario es un solo hilo a través del caso de uso. Por lo tanto, habría un escenario para la interacción normal, más escenarios para cada posible excepción. En la práctica, es posible usarlos en cualquier forma.

Los casos de uso identifican las interacciones individuales entre el sistema y sus usuarios u otros sistemas. Cada caso de uso debe documentarse con una descripción textual. Entonces pueden vincularse con otros modelos en el UML que desarrollará el escenario con más detalle. Por ejemplo, una breve descripción del caso de uso *Establece la consulta* de la figura 4.15 sería:

> *El establecimiento de consulta permite que dos o más médicos, que trabajan en diferentes consultorios, vean el mismo registro simultáneamente. Un médico inicia la consulta al elegir al individuo involucrado de un menú desplegable de médicos que estén en línea. Entonces el registro del paciente se despliega en sus pantallas,*

pero sólo el médico que inicia puede editar el registro. Además, se crea una ventana de chat de texto para ayudar a coordinar las acciones. Se supone que, de manera separada, se establecerá una conferencia telefónica para comunicación por voz.

Los escenarios y los casos de uso son técnicas efectivas para adquirir requerimientos de los participantes que interactúan directamente con el sistema. Cada tipo de interacción puede representarse como caso de uso. Sin embargo, debido a que se enfocan en interacciones con el sistema, no son tan efectivas para adquirir restricciones o requerimientos empresariales y no funcionales de alto nivel, ni para descubrir requerimientos de dominio.

El UML es un estándar *de facto* para modelado orientado a objetos, así que los casos de uso y la adquisición basada en casos ahora se utilizan ampliamente para adquisición de requerimientos. Los casos de uso se estudian en el capítulo 5, y se muestra cómo se emplean junto con otros modelos del sistema para documentar el diseño de un sistema.

4.5.5 Etnografía

Los sistemas de software no existen aislados. Se usan en un contexto social y organizacional, y dicho escenario podría derivar o restringir los requerimientos del sistema de software. A menudo satisfacer dichos requerimientos sociales y organizacionales es crítico para el éxito del sistema. Una razón por la que muchos sistemas de software se entregan, y nunca se utilizan, es que sus requerimientos no consideran de manera adecuada cómo afectaría el contexto social y organizacional la operación práctica del sistema.

La etnografía es una técnica de observación que se usa para entender los procesos operacionales y ayudar a derivar requerimientos de apoyo para dichos procesos. Un analista se adentra en el ambiente laboral donde se usará el sistema. Observa el trabajo diario y toma notas acerca de las tareas existentes en que intervienen los participantes. El valor de la etnografía es que ayuda a descubrir requerimientos implícitos del sistema que reflejan las formas actuales en que trabaja la gente, en vez de los procesos formales definidos por la organización.

Las personas con frecuencia encuentran muy difícil articular los detalles de su trabajo, porque es una segunda forma de vida para ellas. Entienden su trabajo, pero tal vez no su relación con otras funciones en la organización. Los factores sociales y organizacionales que afectan el trabajo, que no son evidentes para los individuos, sólo se vuelven claros cuando los percibe un observador sin prejuicios. Por ejemplo, un grupo de trabajo puede organizarse de modo que sus miembros conozcan el trabajo de los demás y se suplan entre sí cuando alguien se ausenta. Es probable que esto no se mencione durante una entrevista, pues el grupo podría no verlo como una parte integral de su función.

Suchman (1987) fue una de las primeras en usar la etnografía para estudiar el trabajo en la oficina. Ella descubrió que las prácticas reales del trabajo son más ricas, más complejas y más dinámicas que los modelos simples supuestos por los sistemas de automatización administrativa. La diferencia entre el trabajo supuesto y el real fue la razón más importante por la que dichos sistemas de oficina no tenían un efecto significativo sobre la productividad. Crabtree (2003) analiza desde entonces una amplia gama de estudios, y describe, en

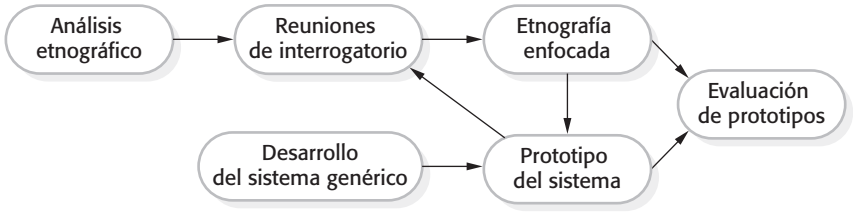


Figura 4.16 Etnografía y creación de prototipos para análisis de requerimientos

general, el uso de la etnografía en el diseño de sistemas. El autor ha investigado métodos para integrar la etnografía en el proceso de ingeniería de software, mediante su vinculación con los métodos de la ingeniería de requerimientos (Viller y Sommerville, 1999; Viller y Sommerville, 2000) y patrones para documentar la interacción en sistemas cooperativos (Martin *et al*., 2001; Martin *et al*., 2002; Martin y Sommerville, 2004).

La etnografía es muy efectiva para descubrir dos tipos de requerimientos:

1. Requerimientos que se derivan de la forma en que realmente trabaja la gente, en vez de la forma en la cual las definiciones del proceso indican que debería trabajar. Por ejemplo, los controladores de tráfico aéreo pueden desactivar un sistema de alerta de conflicto que detecte una aeronave con trayectoria de vuelo que se cruza, aun cuando los procedimientos de control normales especifiquen que es obligatorio usar tal sistema. Ellos deliberadamente dejan a la aeronave sobre la ruta de conflicto durante breves momentos, para ayudarse a dirigir el espacio aéreo. Su estrategia de control está diseñada para garantizar que dichas aeronaves se desvíen antes de que haya problemas, y consideran que la alarma de alerta de conflicto los distrae de su trabajo.

2. Requerimientos que se derivan de la cooperación y el conocimiento de las actividades de otras personas. Por ejemplo, los controladores de tráfico aéreo pueden usar el conocimiento del trabajo de otros controladores para predecir el número de aeronaves que entrarán a su sector de control. Entonces, modifican sus estrategias de control dependiendo de dicha carga de trabajo prevista. Por lo tanto, un sistema ATC automatizado debería permitir a los controladores en un sector tener cierta visibilidad del trabajo en sectores adyacentes.

La etnografía puede combinarse con la creación de prototipos (figura 4.16). La etnografía informa del desarrollo del prototipo, de modo que se requieren menos ciclos de refinamiento del prototipo. Más aún, la creación de prototipos se enfoca en la etnografía al identificar problemas y preguntas que entonces pueden discutirse con el etnógrafo. Siendo así, éste debe buscar las respuestas a dichas preguntas durante la siguiente fase de estudio del sistema (Sommerville *et al*., 1993).

Los estudios etnográficos pueden revelar detalles críticos de procesos, que con frecuencia se pierden con otras técnicas de adquisición de requerimientos. Sin embargo, debido a su enfoque en el usuario final, no siempre es adecuado para descubrir requerimientos de la organización o de dominio. No en todos los casos se identifican nuevas características que deben agregarse a un sistema. En consecuencia, la etnografía no es un enfoque completo para la adquisición por sí misma, y debe usarse para complementar otros enfoques, como el análisis de casos de uso.

Revisiones de requerimientos

Una revisión de requerimientos es un proceso donde un grupo de personas del cliente del sistema y el desarrollador del sistema leen con detalle el documento de requerimientos y buscan errores, anomalías e inconsistencias. Una vez detectados y registrados, recae en el cliente y el desarrollador la labor de negociar cómo resolver los problemas identificados.

http://www.SoftwareEngineering-9.com/Web/Requirements/Reviews.html

4.6 Validación de requerimientos

La validación de requerimientos es el proceso de verificar que los requerimientos definan realmente el sistema que en verdad quiere el cliente. Se traslapa con el análisis, ya que se interesa por encontrar problemas con los requerimientos. La validación de requerimientos es importante porque los errores en un documento de requerimientos pueden conducir a grandes costos por tener que rehacer, cuando dichos problemas se descubren durante el desarrollo del sistema o después de que éste se halla en servicio.

En general, el costo por corregir un problema de requerimientos al hacer un cambio en el sistema es mucho mayor que reparar los errores de diseño o codificación. La razón es que un cambio a los requerimientos significa generalmente que también deben cambiar el diseño y la implementación del sistema. Más aún, el sistema debe entonces ponerse a prueba de nuevo.

Durante el proceso de validación de requerimientos, tienen que realizarse diferentes tipos de comprobaciones sobre los requerimientos contenidos en el documento de requerimientos. Dichas comprobaciones incluyen:

1. *Comprobaciones de validez* Un usuario quizá crea que necesita un sistema para realizar ciertas funciones. Sin embargo, con mayor consideración y análisis se logra identificar las funciones adicionales o diferentes que se requieran. Los sistemas tienen diversos participantes con diferentes necesidades, y cualquier conjunto de requerimientos es inevitablemente un compromiso a través de la comunidad de participantes.
2. *Comprobaciones de consistencia* Los requerimientos en el documento no deben estar en conflicto. Esto es, no debe haber restricciones contradictorias o descripciones diferentes de la misma función del sistema.
3. *Comprobaciones de totalidad* El documento de requerimientos debe incluir requerimientos que definan todas las funciones y las restricciones pretendidas por el usuario del sistema.
4. *Comprobaciones de realismo* Al usar el conocimiento de la tecnología existente, los requerimientos deben comprobarse para garantizar que en realidad pueden implementarse. Dichas comprobaciones también tienen que considerar el presupuesto y la fecha para el desarrollo del sistema.
5. *Verificabilidad* Para reducir el potencial de disputas entre cliente y contratista, los requerimientos del sistema deben escribirse siempre de manera que sean verificables. Esto significa que usted debe ser capaz de escribir un conjunto de pruebas que demuestren que el sistema entregado cumpla cada requerimiento especificado.

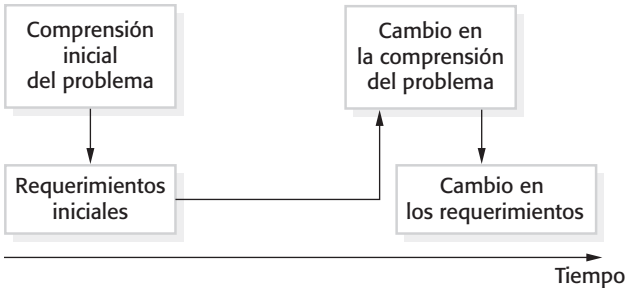


Figura 4.17
Evolución de los requerimientos

Hay algunas técnicas de validación de requerimientos que se usan individualmente o en conjunto con otras:

1. *Revisiones de requerimientos* Los requerimientos se analizan sistemáticamente usando un equipo de revisores que verifican errores e inconsistencias.
2. *Creación de prototipos* En esta aproximación a la validación, se muestra un modelo ejecutable del sistema en cuestión a los usuarios finales y clientes. Así, ellos podrán experimentar con este modelo para constatar si cubre sus necesidades reales.
3. *Generación de casos de prueba* Los requerimientos deben ser comprobables. Si las pruebas para los requerimientos se diseñan como parte del proceso de validación, esto revela con frecuencia problemas en los requerimientos. Si una prueba es difícil o imposible de diseñar, esto generalmente significa que los requerimientos serán difíciles de implementar, por lo que deberían reconsiderarse. El desarrollo de pruebas a partir de los requerimientos del usuario antes de escribir cualquier código es una pieza integral de la programación extrema.

No hay que subestimar los problemas incluidos en la validación de requerimientos. A final de cuentas, es difícil demostrar que un conjunto de requerimientos, de hecho, no cubre las necesidades de los usuarios. Estos últimos necesitan una imagen del sistema en operación, así como comprender la forma en que dicho sistema se ajustará a su trabajo. Es difícil, incluso para profesionales de la computación experimentados, realizar este tipo de análisis abstracto, y más aún para los usuarios del sistema. Como resultado, rara vez usted encontrará todos los problemas de requerimientos durante el proceso de validación de requerimientos. Es inevitable que haya más cambios en los requerimientos para corregir omisiones y malas interpretaciones, después de acordar el documento de requerimientos.

4.7 Administración de requerimientos

Los requerimientos para los grandes sistemas de software siempre cambian. Una razón es que dichos sistemas se desarrollaron por lo general para resolver problemas "horrorosos": aquellos problemas que no se pueden definir por completo. Como el problema no se logra definir por completo, los requerimientos del software están condenados también a estar incompletos. Durante el proceso de software, la comprensión que los participantes tienen de los problemas cambia constantemente (figura 4.17). Entonces, los requerimientos del sistema también deben evolucionar para reflejar esa visión cambiante del problema.

Requerimientos duraderos y volátiles

Algunos requerimientos son más susceptibles a cambiar que otros. Los requerimientos duraderos son los requerimientos que se asocian con las actividades centrales, de lento cambio, de una organización. También estos requerimientos se relacionan con actividades laborales fundamentales. Por el contrario, los requerimientos volátiles tienen más probabilidad de cambio. Se asocian por lo general con actividades de apoyo que reflejan cómo la organización hace su trabajo más que el trabajo en sí.

http://www.SoftwareEngineering-9.com/Web/Requirements/EnduringReq.html

Una vez que se instala un sistema, y se utiliza con regularidad, surgirán inevitablemente nuevos requerimientos. Es difícil que los usuarios y clientes del sistema anticipen qué efectos tendrá el nuevo sistema sobre sus procesos de negocios y la forma en que se hace el trabajo. Una vez que los usuarios finales experimentan el sistema, descubrirán nuevas necesidades y prioridades. Existen muchas razones por las que es inevitable el cambio:

1. Los ambientes empresarial y técnico del sistema siempre cambian después de la instalación. Puede introducirse nuevo hardware, y quizá sea necesario poner en interfaz el sistema con otros sistemas, cambiar las prioridades de la empresa (con los consecuentes cambios en el sistema de apoyo requerido) e introducir nuevas leyes y regulaciones que el sistema deba cumplir cabalmente.
2. Los individuos que pagan por un sistema y los usuarios de dicho sistema, por lo general, no son los mismos. Los clientes del sistema imponen requerimientos debido a restricciones organizativas y presupuestales. Esto podría estar en conflicto con los requerimientos del usuario final y, después de la entrega, probablemente deban agregarse nuevas características para apoyar al usuario, si el sistema debe cubrir sus metas.
3. Los sistemas grandes tienen regularmente una comunidad de usuarios diversa, en la cual muchos individuos tienen diferentes requerimientos y prioridades que quizás estén en conflicto o sean contradictorios. Los requerimientos finales del sistema inevitablemente tienen un compromiso entre sí y, con la experiencia, a menudo se descubre que el equilibrio de apoyo brindado a diferentes usuarios tiene que cambiar.

La administración de requerimientos es el proceso de comprender y controlar los cambios en los requerimientos del sistema. Es necesario seguir la pista de requerimientos individuales y mantener los vínculos entre los requerimientos dependientes, de manera que pueda valorarse el efecto del cambio en los requerimientos. También es preciso establecer un proceso formal para hacer cambios a las propuestas y vincular éstos con los requerimientos del sistema. El proceso formal de la administración de requerimientos debe comenzar tan pronto como esté disponible un borrador del documento de requerimientos. Sin embargo, hay que empezar a planear cómo administrar el cambio en los requerimientos durante el proceso de adquisición de los mismos.

4.7.1 Planeación de la administración de requerimientos

La planeación es una primera etapa esencial en el proceso de administración de requerimientos. Esta etapa establece el nivel de detalle que se requiere en la administración de requerimientos. Durante la etapa de administración de requerimientos, usted tiene que decidir sobre:

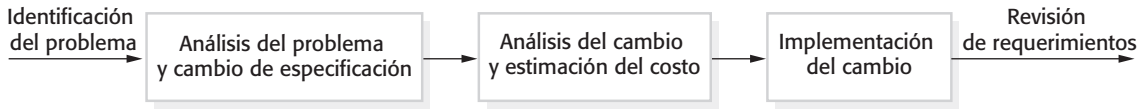


Figura 4.18 Administración del cambio de requerimientos

1. *Identificación de requerimientos* Cada requerimiento debe identificarse de manera exclusiva, de forma que pueda tener referencia cruzada con otros requerimientos y usarse en las evaluaciones de seguimiento.
2. *Un proceso de administración del cambio* Éste es el conjunto de actividades que valoran el efecto y costo de los cambios. En la siguiente sección se estudia con más detalle este proceso.
3. *Políticas de seguimiento* Dichas políticas definen las relaciones entre cada requerimiento, así como entre los requerimientos y el diseño del sistema que debe registrarse. La política de seguimiento también tiene que definir cómo mantener dichos registros.
4. *Herramientas de apoyo* La administración de requerimientos incluye el procesamiento de grandes cantidades de información acerca de los requerimientos. Las herramientas disponibles varían desde sistemas especializados de administración de requerimientos, hasta hojas de cálculo y sistemas de bases de datos simples.

La administración de requerimientos necesita apoyo automatizado y herramientas de software, para lo cual deben seleccionarse durante la fase de planeación. Se necesitan herramientas de apoyo para:

1. *Almacenamiento de requerimientos* Los requerimientos tienen que mantenerse en un almacén de datos administrado y seguro, que sea accesible para todos quienes intervienen en el proceso de ingeniería de requerimientos.
2. *Administración del cambio* El proceso de administración del cambio (figura 4.18) se simplifica si está disponible la herramienta de apoyo activa.
3. *Administración del seguimiento* Como se estudió anteriormente, la herramienta de apoyo para el seguimiento permite la identificación de requerimientos relacionados. Algunas herramientas que están disponibles usan técnicas de procesamiento en lenguaje natural, para ayudar a descubrir posibles relaciones entre los requerimientos.

Para sistemas pequeños, quizá no sea necesario usar herramientas especializadas de administración de requerimientos. El proceso de administración de requerimientos puede apoyarse con el uso de funciones disponibles en procesadores de texto, hojas de cálculo y bases de datos de PC. Sin embargo, para sistemas más grandes se requieren herramientas de apoyo más especializadas. En las páginas Web del libro se incluyen vínculos a información acerca de herramientas de administración de requerimientos.

4.7.2 Administración del cambio en los requerimientos

La administración del cambio en los requerimientos (figura 4.18) debe aplicarse a todos los cambios propuestos a los requerimientos de un sistema, después de aprobarse el documento de requerimientos. La administración del cambio es esencial porque es necesario determinar si los beneficios de implementar nuevos requerimientos están justificados por

Seguimiento de requerimientos

Es necesario seguir la huella de las relaciones entre requerimientos, sus fuentes y el diseño del sistema, de modo que usted pueda analizar las razones para los cambios propuestos, así como el efecto que dichos cambios tengan probablemente sobre otras partes del sistema. Es necesario poder seguir la pista de cómo un cambio se propaga hacia el sistema. ¿Por qué?

http://www.SoftwareEngineering-9.com/Web/Requirements/ReqTraceability.html

los costos de la implementación. La ventaja de usar un proceso formal para la administración del cambio es que todas las propuestas de cambio se tratan de manera consistente y los cambios al documento de requerimientos se realizan en una forma controlada.

Existen tres etapas principales de un proceso de administración del cambio:

1. *Análisis del problema y especificación del cambio* El proceso comienza con la identificación de un problema en los requerimientos o, en ocasiones, con una propuesta de cambio específica. Durante esta etapa, el problema o la propuesta de cambio se analizan para comprobar que es válida. Este análisis retroalimenta al solicitante del cambio, quien responderá con una propuesta de cambio de requerimientos más específica, o decidirá retirar la petición.

2. *Análisis del cambio y estimación del costo* El efecto del cambio propuesto se valora usando información de seguimiento y conocimiento general de los requerimientos del sistema. El costo por realizar el cambio se estima en términos de modificaciones al documento de requerimientos y, si es adecuado, al diseño y la implementación del sistema. Una vez completado este análisis, se toma una decisión acerca de si se procede o no con el cambio de requerimientos.

3. *Implementación del cambio* Se modifican el documento de requerimientos y, donde sea necesario, el diseño y la implementación del sistema. Hay que organizar el documento de requerimientos de forma que sea posible realizar cambios sin reescritura o reorganización extensos. Conforme a los programas, la variabilidad en los documentos se logra al minimizar las referencias externas y al hacer las secciones del documento tan modulares como sea posible. De esta manera, secciones individuales pueden modificarse y sustituirse sin afectar otras partes del documento.

Si un nuevo requerimiento tiene que implementarse urgentemente, siempre existe la tentación para cambiar el sistema y luego modificar de manera retrospectiva el documento de requerimientos. Hay que tratar de evitar esto, pues casi siempre conducirá a que la especificación de requerimientos y la implementación del sistema se salgan de ritmo. Una vez realizados los cambios al sistema, es fácil olvidar la inclusión de dichos cambios en el documento de requerimientos, o bien, agregar información al documento de requerimientos que sea inconsistente con la implementación.

Los procesos de desarrollo ágil, como la programación extrema, se diseñaron para enfrentar los requerimientos que cambian durante el proceso de desarrollo. En dichos procesos, cuando un usuario propone un cambio de requerimientos, éste no pasa por un proceso de administración del cambio formal. En vez de ello, el usuario tiene que priorizar dicho cambio y, si es de alta prioridad, decidir qué características del sistema planeadas para la siguiente iteración pueden eliminarse.

PUNTOS CLAVE

- Los requerimientos para un sistema de software establecen lo que debe hacer el sistema y definen las restricciones sobre su operación e implementación.
- Los requerimientos funcionales son enunciados de los servicios que debe proporcionar el sistema, o descripciones de cómo deben realizarse algunos cálculos.
- Los requerimientos no funcionales restringen con frecuencia el sistema que se va a desarrollar y el proceso de desarrollo a usar. Éstos pueden ser requerimientos del producto, requerimientos organizacionales o requerimientos externos. A menudo se relacionan con las propiedades emergentes del sistema y, por lo tanto, se aplican al sistema en su conjunto.
- El documento de requerimientos de software es un enunciado acordado sobre los requerimientos del sistema. Debe organizarse de forma que puedan usarlo tanto los clientes del sistema como los desarrolladores del software.
- El proceso de ingeniería de requerimientos incluye un estudio de factibilidad, adquisición y análisis de requerimientos, especificación de requerimientos, validación de requerimientos y administración de requerimientos.
- La adquisición y el análisis de requerimientos es un proceso iterativo que se representa como una espiral de actividades: descubrimiento de requerimientos, clasificación y organización de requerimientos, negociación de requerimientos y documentación de requerimientos.
- La validación de requerimientos es el proceso de comprobar la validez, la consistencia, la totalidad, el realismo y la verificabilidad de los requerimientos.
- Los cambios empresariales, organizacionales y técnicos conducen inevitablemente a cambios en los requerimientos para un sistema de software. La administración de requerimientos es el proceso de gestionar y controlar dichos cambios.

LECTURAS SUGERIDAS

Software Requirements, 2nd edition. Este libro, diseñado para escritores y usuarios de requerimientos, analiza las buenas prácticas en la ingeniería de requerimientos. (K. M. Weigers, 2003, Microsoft Press.)

"Integrated requirements engineering: A tutorial". Se trata de un ensayo de tutoría en el que se analizan las actividades de la ingeniería de requerimientos y cómo pueden adaptarse para ajustarse a las prácticas modernas de la ingeniería de software. (I. Sommerville, IEEE Software, 22(1), Jan–Feb 2005.) http://dx.doi.org/10.1109/MS.2005.13.

Mastering the Requirements Process, 2nd edition. Un libro bien escrito, fácil de leer, que se basa en un método particular (VOLERE), pero que también incluye múltiples buenos consejos generales acerca de la ingeniería de requerimientos. (S. Robertson y J. Robertson, 2006, Addison-Wesley.)

"Research Directions in Requirements Engineering". Un buen estudio de la investigación en ingeniería de requerimientos que destaca los futuros retos en la investigación en el área, con la finalidad de enfrentar conflictos como la escala y la agilidad. (B. H. C. Cheng y J. M. Atlee, Proc. Conf on Future of Software Engineering, IEEE Computer Society, 2007.) http://dx.doi.org/10.1109/FOSE.2007.17.

EJERCICIOS

4.1. Identifique y describa brevemente cuatro tipos de requerimientos que puedan definirse para un sistema basado en computadora.

4.2. Descubra las ambigüedades u omisiones en el siguiente enunciado de requerimientos de un sistema de emisión de boletos:

Un sistema automatizado de emisión de boletos vende boletos de ferrocarril. Los usuarios seleccionan su destino e ingresan un número de tarjeta de crédito y uno de identificación personal. El boleto de ferrocarril se emite y se carga en su cuenta de tarjeta de crédito. Cuando el usuario oprime el botón *start*, se activa una pantalla de menú con los posibles destinos, junto con un mensaje que pide al usuario seleccionar un destino. Una vez seleccionado el destino, se solicita a los usuarios ingresar su tarjeta de crédito. Se comprueba su validez y luego se pide al usuario ingresar un identificador personal. Cuando se valida la transacción crediticia, se emite el boleto.

4.3. Vuelva a escribir la descripción anterior usando el enfoque estructurado referido en este capítulo. Resuelva las ambigüedades identificadas de forma adecuada.

4.4. Escriba un conjunto de requerimientos no funcionales para el sistema de emisión de boletos, y establezca su fiabilidad y tiempo de respuesta esperados.

4.5. Con la técnica aquí sugerida, en que las descripciones en lenguaje natural se presentan en formato estándar, escriba requerimientos de usuario plausibles para las siguientes funciones:

- Un sistema de bombeo de petróleo (gasolina) no asistido que incluya un lector de tarjeta de crédito. El cliente pasa la tarjeta en el lector, luego especifica la cantidad de combustible requerido. Se suministra el combustible y se deduce de la cuenta del cliente.
- La función de dispensar efectivo en un cajero automático.
- La función de revisión y corrección ortográfica en un procesador de textos.

4.6. Sugiera cómo un ingeniero responsable de redactar una especificación de requerimientos de sistema puede seguir la huella de las relaciones entre requerimientos funcionales y no funcionales.

4.7. Con su conocimiento de cómo se usa un cajero automático, desarrolle un conjunto de casos de uso que pudieran servir como base para comprender los requerimientos para el sistema de un cajero automático.

4.8. ¿Quién debería involucrarse en una revisión de requerimientos? Dibuje un modelo del proceso que muestre cómo podría organizarse una revisión de requerimientos.

4.9. Cuando tienen que hacerse cambios de emergencia a los sistemas, es posible que deba modificarse el software del sistema antes de aprobar los cambios a los requerimientos. Sugiera un modelo de un proceso para realizar dichas modificaciones, que garantice que el documento de requerimientos y la implementación del sistema no serán inconsistentes.

4.10. Usted acepta un empleo con un usuario de software, quien contrató a su empleador anterior con la finalidad de desarrollar un sistema para ellos. Usted descubre que la interpretación de los requerimientos de su compañía es diferente de la interpretación tomada por su antiguo empleador. Discuta qué haría en tal situación. Usted sabe que los costos para su actual empleador aumentarán si no se resuelven las ambigüedades. Sin embargo, también tiene una responsabilidad de confidencialidad con su empleador anterior.

REFERENCIAS

Beck, K. (1999). "Embracing Change with Extreme Programming". *IEEE Computer,* **32** (10), 70–8.

Crabtree, A. (2003). *Designing Collaborative Systems: A Practical Guide to Ethnography.* London: Springer-Verlag.

Davis, A. M. (1993). *Software Requirements: Objects, Functions and States.* Englewood Cliffs, NJ: Prentice Hall.

IEEE. (1998). "IEEE Recommended Practice for Software Requirements Specifications". En *IEEE Software Engineering Standards Collection.* Los Alamitos, Ca.: IEEE Computer Society Press.

Jacobson, I., Christerson, M., Jonsson, P. y Overgaard, G. (1993). *Object-Oriented Software Engineering.* Wokingham: Addison-Wesley.

Kotonya, G. y Sommerville, I. (1998). *Requirements Engineering: Processes and Techniques.* Chichester, UK: John Wiley and Sons.

Larman, C. (2002). *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process.* Englewood Cliff, NJ: Prentice Hall.

Martin, D., Rodden, T., Rouncefield, M., Sommerville, I. y Viller, S. (2001). "Finding Patterns in the Fieldwork". *Proc. ECSCW' 01.* Bonn: Kluwer. 39–58.

Martin, D., Rouncefield, M. y Sommerville, I. (2002). "Applying patterns of interaction to work (re) design: E-government and planning". *Proc. ACM CHI' 2002,* ACM Press. 235–42.

Martin, D. y Sommerville, I. (2004). "Patterns of interaction: Linking ethnomethodology and design". *ACM Trans. on Computer-Human Interaction,* **11** (1), 59–89.

Robertson, S. y Robertson, J. (1999). *Mastering the Requirements Process.* Harlow, UK: Addison-Wesley.

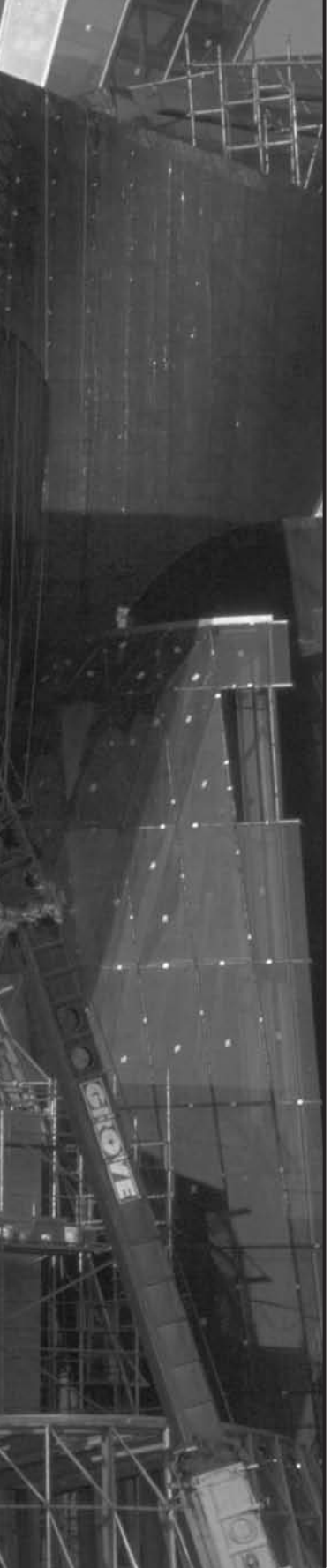
Sommerville, I., Rodden, T., Sawyer, P., Bentley, R. y Twidale, M. (1993). "Integrating ethnography into the requirements engineering process". *Proc. RE' 93,* San Diego CA.: IEEE Computer Society Press. 165–73.

Stevens, P. y Pooley, R. (2006). *Using UML: Software Engineering with Objects and Components, 2nd ed.* Harlow, UK: Addison Wesley.

Suchman, L. (1987). *Plans and Situated Actions.* Cambridge: Cambridge University Press.

Viller, S. y Sommerville, I. (1999). "Coherence: An Approach to Representing Ethnographic Analyses in Systems Design". *Human-Computer Interaction,* **14** (1 & 2), 9–41.

Viller, S. y Sommerville, I. (2000). "Ethnographically informed analysis for software engineers". *Int. J. of Human-Computer Studies,* **53** (1), 169–96.



6

Diseño arquitectónico

Objetivos

El objetivo de este capítulo es introducir los conceptos de arquitectura de software y diseño arquitectónico. Al estudiar este capítulo:

- comprenderá por qué es importante el diseño arquitectónico del software;
- conocerá las decisiones que deben tomarse sobre la arquitectura de software durante el proceso de diseño arquitectónico;
- asimilará la idea de los patrones arquitectónicos, formas bien reconocidas de organización de las arquitecturas del sistema, que pueden reutilizarse en los diseños del sistema;
- identificará los patrones arquitectónicos usados frecuentemente en diferentes tipos de sistemas de aplicación, incluidos los sistemas de procesamiento de transacción y los sistemas de procesamiento de lenguaje.

Contenido

El diseño arquitectónico se interesa por entender cómo debe organizarse un sistema y cómo tiene que diseñarse la estructura global de ese sistema. En el modelo del proceso de desarrollo de software, como se mostró en el capítulo 2, el diseño arquitectónico es la primera etapa en el proceso de diseño del software. Es el enlace crucial entre el diseño y la ingeniería de requerimientos, ya que identifica los principales componentes estructurales en un sistema y la relación entre ellos. La salida del proceso de diseño arquitectónico consiste en un modelo arquitectónico que describe la forma en que se organiza el sistema como un conjunto de componentes en comunicación.

En los procesos ágiles, por lo general se acepta que una de las primeras etapas en el proceso de desarrollo debe preocuparse por establecer una arquitectura global del sistema. Usualmente no resulta exitoso el desarrollo incremental de arquitecturas. Mientras que la refactorización de componentes en respuesta a los cambios suele ser relativamente fácil, tal vez resulte costoso refactorizar una arquitectura de sistema.

Para ayudar a comprender lo que se entiende por arquitectura del sistema, tome en cuenta la figura 6.1. En ella se presenta un modelo abstracto de la arquitectura para un sistema de robot de empaquetado, que indica los componentes que tienen que desarrollarse. Este sistema robótico empaca diferentes clases de objetos. Usa un componente de visión para recoger los objetos de una banda transportadora, identifica la clase de objeto y selecciona el tipo correcto de empaque. Luego, el sistema mueve los objetos que va a empacar de la banda transportadora de entrega y coloca los objetos empacados en otro transportador. El modelo arquitectónico presenta dichos componentes y los vínculos entre ellos.

En la práctica, hay un significativo traslape entre los procesos de ingeniería de requerimientos y el diseño arquitectónico. De manera ideal, una especificación de sistema no debe incluir cierta información de diseño. Esto no es realista, excepto para sistemas muy pequeños. La descomposición arquitectónica es por lo general necesaria para estructurar y organizar la especificación. Por lo tanto, como parte del proceso de ingeniería de requerimientos, usted podría proponer una arquitectura de sistema abstracta donde se asocien grupos de funciones de sistemas o características con componentes o subsistemas a gran escala. Luego, puede usar esta descomposición para discutir con los participantes sobre los requerimientos y las características del sistema.

Las arquitecturas de software se diseñan en dos niveles de abstracción, que en este texto se llaman *arquitectura en pequeño* y *arquitectura en grande*:

1. La arquitectura en pequeño se interesa por la arquitectura de programas individuales. En este nivel, uno se preocupa por la forma en que el programa individual se separa en componentes. Este capítulo se centra principalmente en arquitecturas de programa.
2. La arquitectura en grande se interesa por la arquitectura de sistemas empresariales complejos que incluyen otros sistemas, programas y componentes de programa. Tales sistemas empresariales se distribuyen a través de diferentes computadoras, que diferentes compañías administran y poseen. En los capítulos 18 y 19 se cubren las arquitecturas grandes; en ellos se estudiarán las arquitecturas de los sistemas distribuidos.

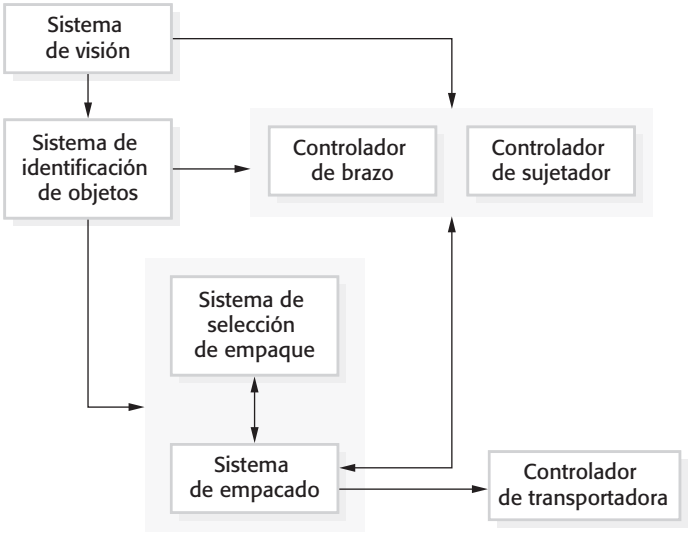


Figura 6.1 Arquitectura de un sistema de control para un robot empacador

La arquitectura de software es importante porque afecta el desempeño y la potencia, así como la capacidad de distribución y mantenimiento de un sistema (Bosch, 2000). Como afirma Bosch, los componentes individuales implementan los requerimientos funcionales del sistema. Los requerimientos no funcionales dependen de la arquitectura del sistema, es decir, la forma en que dichos componentes se organizan y comunican. En muchos sistemas, los requerimientos no funcionales están también influidos por componentes individuales, pero no hay duda de que la arquitectura del sistema es la influencia dominante.

Bass y sus colaboradores (2003) analizan tres ventajas de diseñar y documentar de manera explícita la arquitectura de software:

1. *Comunicación con los participantes* La arquitectura es una presentación de alto nivel del sistema, que puede usarse como un enfoque para la discusión de un amplio número de participantes.
2. *Análisis del sistema* En una etapa temprana en el desarrollo del sistema, aclarar la arquitectura del sistema requiere cierto análisis. Las decisiones de diseño arquitectónico tienen un efecto profundo sobre si el sistema puede o no cubrir requerimientos críticos como rendimiento, fiabilidad y mantenibilidad.
3. *Reutilización a gran escala* Un modelo de una arquitectura de sistema es una descripción corta y manejable de cómo se organiza un sistema y cómo interoperan sus componentes. Por lo general, la arquitectura del sistema es la misma para sistemas con requerimientos similares y, por lo tanto, puede soportar reutilización de software a gran escala. Como se explica en el capítulo 16, es posible desarrollar arquitecturas de línea de productos donde la misma arquitectura se reutilice mediante una amplia gama de sistemas relacionados.

Hofmeister y sus colaboradores (2000) proponen que una arquitectura de software sirve en primer lugar como un plan de diseño para la negociación de requerimientos de sistema y, en segundo lugar, como un medio para establecer discusiones con clientes, desarrolladores y administradores. También sugieren que es una herramienta esencial para la administración de la complejidad. Oculta los detalles y permite a los diseñadores enfocarse en las abstracciones clave del sistema.

Las arquitecturas de sistemas se modelan con frecuencia usando diagramas de bloques simples, como en la figura 6.1. Cada recuadro en el diagrama representa un componente. Los recuadros dentro de recuadros indican que el componente se dividió en subcomponentes. Las flechas significan que los datos y/o señales de control pasan de un componente a otro en la dirección de las flechas. Hay varios ejemplos de este tipo de modelo arquitectónico en el catálogo de arquitectura de software de Booch (Booch, 2009).

Los diagramas de bloque presentan una imagen de alto nivel de la estructura del sistema e incluyen fácilmente a individuos de diferentes disciplinas que intervienen en el proceso de desarrollo del sistema. No obstante su amplio uso, Bass y sus colaboradores (2003) no están de acuerdo con los diagramas de bloque informales para describir una arquitectura. Afirman que tales diagramas informales son representaciones arquitectónicas deficientes, pues no muestran ni el tipo de relaciones entre los componentes del sistema ni las propiedades externamente visibles de los componentes.

Las aparentes contradicciones entre práctica y teoría arquitectónica surgen porque hay dos formas en que se utiliza un modelo arquitectónico de un programa:

1. *Como una forma de facilitar la discusión acerca del diseño del sistema* Una visión arquitectónica de alto nivel de un sistema es útil para la comunicación con los participantes de un sistema y la planeación del proyecto, ya que no se satura con detalles. Los participantes pueden relacionarse con él y entender una visión abstracta del sistema. En tal caso, analizan el sistema como un todo sin confundirse por los detalles. El modelo arquitectónico identifica los componentes clave que se desarrollarán, de modo que los administradores pueden asignar a individuos para planear el desarrollo de dichos sistemas.
2. *Como una forma de documentar una arquitectura que se haya diseñado* La meta aquí es producir un modelo de sistema completo que muestre los diferentes componentes en un sistema, sus interfaces y conexiones. El argumento para esto es que tal descripción arquitectónica detallada facilita la comprensión y la evolución del sistema.

Los diagramas de bloque son una forma adecuada para describir la arquitectura del sistema durante el proceso de diseño, pues son una buena manera de soportar las comunicaciones entre las personas involucradas en el proceso. En muchos proyectos, suele ser la única documentación arquitectónica que existe. Sin embargo, si la arquitectura de un sistema debe documentarse ampliamente, entonces es mejor usar una notación con semántica bien definida para la descripción arquitectónica. No obstante, tal como se estudia en la sección 6.2, algunas personas consideran que la documentación detallada ni es útil ni vale realmente la pena el costo de su desarrollo.

6.1 Decisiones en el diseño arquitectónico

El diseño arquitectónico es un proceso creativo en el cual se diseña una organización del sistema que cubrirá los requerimientos funcionales y no funcionales de éste. Puesto que se trata de un proceso creativo, las actividades dentro del proceso dependen del tipo de sistema que se va a desarrollar, los antecedentes y la experiencia del arquitecto del sistema, así como de los requerimientos específicos del sistema. Por lo tanto, es útil pensar en el diseño arquitectónico como un conjunto de decisiones a tomar en vez de una secuencia de actividades.

Durante el proceso de diseño arquitectónico, los arquitectos del sistema deben tomar algunas decisiones estructurales que afectarán profundamente el sistema y su proceso de desarrollo. Con base en su conocimiento y experiencia, deben considerar las siguientes preguntas fundamentales sobre el sistema:

1. ¿Existe alguna arquitectura de aplicación genérica que actúe como plantilla para el sistema que se está diseñando?
2. ¿Cómo se distribuirá el sistema a través de algunos núcleos o procesadores?
3. ¿Qué patrones o estilos arquitectónicos pueden usarse?
4. ¿Cuál será el enfoque fundamental usado para estructurar el sistema?
5. ¿Cómo los componentes estructurales en el sistema se separarán en subcomponentes?
6. ¿Qué estrategia se usará para controlar la operación de los componentes en el sistema?
7. ¿Cuál organización arquitectónica es mejor para entregar los requerimientos no funcionales del sistema?
8. ¿Cómo se evaluará el diseño arquitectónico?
9. ¿Cómo se documentará la arquitectura del sistema?

Aunque cada sistema de software es único, los sistemas en el mismo dominio de aplicación tienen normalmente arquitecturas similares que reflejan los conceptos fundamentales del dominio. Por ejemplo, las líneas de producto de aplicación son aplicaciones que se construyen en torno a una arquitectura central con variantes que cubren requerimientos específicos del cliente. Cuando se diseña una arquitectura de sistema, debe decidirse qué tienen en común el sistema y las clases de aplicación más amplias, con la finalidad de determinar cuánto conocimiento se puede reutilizar de dichas arquitecturas de aplicación. En la sección 6.4 se estudian las arquitecturas de aplicación genéricas y en el capítulo 16 las líneas de producto de aplicación.

Para sistemas embebidos y sistemas diseñados para computadoras personales, por lo general, hay un solo procesador y no tendrá que diseñar una arquitectura distribuida para el sistema. Sin embargo, los sistemas más grandes ahora son sistemas distribuidos donde el software de sistema se distribuye a través de muchas y diferentes computadoras. La elección de arquitectura de distribución es una decisión clave que afecta el rendimiento y

la fiabilidad del sistema. Éste es un tema importante por derecho propio y se trata por separado en el capítulo 18.

La arquitectura de un sistema de software puede basarse en un patrón o un estilo arquitectónico particular. Un patrón arquitectónico es una descripción de una organización del sistema (Garlan y Shaw, 1993), tal como una organización cliente-servidor o una arquitectura por capas. Los patrones arquitectónicos captan la esencia de una arquitectura que se usó en diferentes sistemas de software. Usted tiene que conocer tanto los patrones comunes, en que éstos se usen, como sus fortalezas y debilidades cuando se tomen decisiones sobre la arquitectura de un sistema. En la sección 6.3 se analizan algunos patrones de uso frecuente.

La noción de un estilo arquitectónico de Garlan y Shaw (estilo y patrón llegaron a significar lo mismo) cubre las preguntas 4 a 6 de la lista anterior. Es necesario elegir la estructura más adecuada, como cliente-servidor o estructura en capas, que le permita satisfacer los requerimientos del sistema. Para descomponer las unidades del sistema estructural, usted opta por la estrategia de separar los componentes en subcomponentes. Los enfoques que pueden usarse permiten la implementación de diferentes tipos de arquitectura. Finalmente, en el proceso de modelado de control, se toman decisiones sobre cómo se controla la ejecución de componentes. Se desarrolla un modelo general de las relaciones de control entre las diferentes partes del sistema.

Debido a la estrecha relación entre los requerimientos no funcionales y la arquitectura de software, el estilo y la estructura arquitectónicos particulares que se elijan para un sistema dependerán de los requerimientos de sistema no funcionales:

1. *Rendimiento* Si el rendimiento es un requerimiento crítico, la arquitectura debe diseñarse para localizar operaciones críticas dentro de un pequeño número de componentes, con todos estos componentes desplegados en la misma computadora en vez de distribuirlos por la red. Esto significaría usar algunos componentes relativamente grandes, en vez de pequeños componentes de grano fino, lo cual reduce el número de comunicaciones entre componentes. También puede considerar organizaciones del sistema en tiempo de operación que permitan a éste ser replicable y ejecutable en diferentes procesadores.

2. *Seguridad* Si la seguridad es un requerimiento crítico, será necesario usar una estructura en capas para la arquitectura, con los activos más críticos protegidos en las capas más internas, y con un alto nivel de validación de seguridad aplicado a dichas capas.

3. *Protección* Si la protección es un requerimiento crítico, la arquitectura debe diseñarse de modo que las operaciones relacionadas con la protección se ubiquen en algún componente individual o en un pequeño número de componentes. Esto reduce los costos y problemas de validación de la protección, y hace posible ofrecer sistemas de protección relacionados que, en caso de falla, desactiven con seguridad el sistema.

4. *Disponibilidad* Si la disponibilidad es un requerimiento crítico, la arquitectura tiene que diseñarse para incluir componentes redundantes de manera que sea posible sustituir y actualizar componentes sin detener el sistema. En el capítulo 13 se describen dos arquitecturas de sistema tolerantes a fallas en sistemas de alta disponibilidad.

5. *Mantenibilidad* Si la mantenibilidad es un requerimiento crítico, la arquitectura del sistema debe diseñarse usando componentes autocontenidos de grano fino que

puedan cambiarse con facilidad. Los productores de datos tienen que separarse de los consumidores y hay que evitar compartir las estructuras de datos.

Evidentemente, hay un conflicto potencial entre algunas de estas arquitecturas. Por ejemplo, usar componentes grandes mejora el rendimiento, y utilizar componentes pequeños de grano fino aumenta la mantenibilidad. Si tanto el rendimiento como la mantenibilidad son requerimientos importantes del sistema, entonces debe encontrarse algún compromiso. Esto en ocasiones se logra usando diferentes patrones o estilos arquitectónicos para distintas partes del sistema.

Evaluar un diseño arquitectónico es difícil porque la verdadera prueba de una arquitectura es qué tan bien el sistema cubre sus requerimientos funcionales y no funcionales cuando está en uso. Sin embargo, es posible hacer cierta evaluación al comparar el diseño contra arquitecturas de referencia o patrones arquitectónicos genéricos. Para ayudar con la evaluación arquitectónica, también puede usarse la descripción de Bosch (2000) de las características no funcionales de los patrones arquitectónicos.

6.2 Vistas arquitectónicas

En la introducción a este capítulo se explicó que los modelos arquitectónicos de un sistema de software sirven para enfocar la discusión sobre los requerimientos o el diseño del software. De manera alternativa, pueden emplearse para documentar un diseño, de modo que se usen como base en el diseño y la implementación más detallados, así como en la evolución futura del sistema. En esta sección se estudian dos temas que son relevantes para ambos:

1. ¿Qué vistas o perspectivas son útiles al diseñar y documentar una arquitectura del sistema?
2. ¿Qué notaciones deben usarse para describir modelos arquitectónicos?

Es imposible representar toda la información relevante sobre la arquitectura de un sistema en un solo modelo arquitectónico, ya que cada uno presenta únicamente una vista o perspectiva del sistema. Ésta puede mostrar cómo un sistema se descompone en módulos, cómo interactúan los procesos de tiempo de operación o las diferentes formas en que los componentes del sistema se distribuyen a través de una red. Todo ello es útil en diferentes momentos de manera que, para el diseño y la documentación, por lo general se necesita presentar múltiples vistas de la arquitectura de software.

Existen diferentes opiniones relativas a qué vistas se requieren. Krutchen (1995), en su bien conocido modelo de vista 4+1 de la arquitectura de software, sugiere que deben existir cuatro vistas arquitectónicas fundamentales, que se relacionan usando casos de uso o escenarios. Las vistas que él sugiere son:

1. Una vista lógica, que indique las abstracciones clave en el sistema como objetos o clases de objeto. En este tipo de vista se tienen que relacionar los requerimientos del sistema con entidades.

2. Una vista de proceso, que muestre cómo, en el tiempo de operación, el sistema está compuesto de procesos en interacción. Esta vista es útil para hacer juicios acerca de las características no funcionales del sistema, como el rendimiento y la disponibilidad.
3. Una vista de desarrollo, que muestre cómo el software está descompuesto para su desarrollo, esto es, indica la descomposición del software en elementos que se implementen mediante un solo desarrollador o equipo de desarrollo. Esta vista es útil para administradores y programadores de software.
4. Una vista física, que exponga el hardware del sistema y cómo los componentes de software se distribuyen a través de los procesadores en el sistema. Esta vista es útil para los ingenieros de sistemas que planean una implementación de sistema.

Hofmeister y sus colaboradores (2000) sugieren el uso de vistas similares, pero a éstas agregan la noción de vista conceptual. Esta última es una vista abstracta del sistema que puede ser la base para descomponer los requerimientos de alto nivel en especificaciones más detalladas, ayudar a los ingenieros a tomar decisiones sobre componentes que puedan reutilizarse, y representar una línea de producto (que se estudia en el capítulo 16) en vez de un solo sistema. La figura 6.1, que describe la arquitectura de un robot de empacado, es un ejemplo de una vista conceptual del sistema.

En la práctica, las vistas conceptuales casi siempre se desarrollan durante el proceso de diseño y se usan para apoyar la toma de decisiones arquitectónicas. Son una forma de comunicar a diferentes participantes la esencia de un sistema. Durante el proceso de diseño, también pueden desarrollarse algunas de las otras vistas, al tiempo que se discuten diferentes aspectos del sistema, aunque no haya necesidad de una descripción completa desde todas las perspectivas. Además se podrían asociar patrones arquitectónicos, estudiados en la siguiente sección, con las diferentes vistas de un sistema.

Hay diferentes opiniones respecto de si los arquitectos de software deben o no usar el UML para una descripción arquitectónica (Clements *et al*., 2002). Un estudio en 2006 (Lange *et al*., 2006) demostró que, cuando se usa el UML, se aplica principalmente en una forma holgada e informal. Los autores de dicho ensayo argumentan que esto era incorrecto. El autor no está de acuerdo con esta visión. El UML se diseñó para describir sistemas orientados a objetos y, en la etapa de diseño arquitectónico, uno quiere describir con frecuencia sistemas en un nivel superior de abstracción. Las clases de objetos están muy cerca de la implementación, como para ser útiles en la descripción arquitectónica.

Para el autor, el UML no es útil durante el proceso de diseño en sí y prefiere notaciones informales que sean más rápidas de escribir y puedan dibujarse fácilmente en un pizarrón. El UML es de más valor cuando se documenta una arquitectura a detalle o se usa un desarrollo dirigido por modelo, como se estudió en el capítulo 5.

Algunos investigadores proponen el uso de lenguajes de descripción arquitectónica (ADL, por las siglas de *Architectural Description Languages*) más especializados (Bass *et al*., 2003) para describir arquitecturas del sistema. Los elementos básicos de los ADL son componentes y conectores, e incluyen reglas y lineamientos para arquitecturas bien formadas. Sin embargo, debido a su naturaleza especializada, los expertos de dominio y aplicación tienen dificultad para entender y usar los ADL. Esto dificulta la valoración de su utilidad para la ingeniería práctica del software. Los ADL diseñados para un dominio particular (por ejemplo, sistemas automotores) pueden usarse como una base

Nombre	MVC (modelo de vista del controlador)
Descripción	Separa presentación e interacción de los datos del sistema. El sistema se estructura en tres componentes lógicos que interactúan entre sí. El componente Modelo maneja los datos del sistema y las operaciones asociadas a esos datos. El componente Vista define y gestiona cómo se presentan los datos al usuario. El componente Controlador dirige la interacción del usuario (por ejemplo, teclas oprimidas, clics del mouse, etcétera) y pasa estas interacciones a Vista y Modelo. Véase la figura 6.3.
Ejemplo	La figura 6.4 muestra la arquitectura de un sistema de aplicación basado en la Web, que se organiza con el uso del patrón MVC.
Cuándo se usa	Se usa cuando existen múltiples formas de ver e interactuar con los datos. También se utiliza al desconocerse los requerimientos futuros para la interacción y presentación.
Ventajas	Permite que los datos cambien de manera independiente de su representación y viceversa. Soporta en diferentes formas la presentación de los mismos datos, y los cambios en una representación se muestran en todos ellos.
Desventajas	Puede implicar código adicional y complejidad de código cuando el modelo de datos y las interacciones son simples.

Figura 6.2 Patrón modelo de vista del controlador (MVC)

para el desarrollo dirigido por modelo. Sin embargo, se considera que los modelos y las notaciones informales, como el UML, seguirán siendo las formas de uso más común para documentar las arquitecturas del sistema.

Los usuarios de métodos ágiles afirman que, por lo general, no se utiliza la documentación detallada del diseño. Por lo tanto, desarrollarla es un desperdicio de tiempo y dinero. El autor está en gran medida de acuerdo con esta visión y considera que, para la mayoría de los sistemas, no vale la pena desarrollar una descripción arquitectónica detallada desde estas cuatro perspectivas. Uno debe desarrollar las vistas que sean útiles para la comunicación sin preocuparse si la documentación arquitectónica está completa o no. Sin embargo, una excepción es al desarrollar sistemas críticos, cuando es necesario realizar un análisis de confiabilidad detallado del sistema. Tal vez se deba convencer a reguladores externos de que el sistema se hizo conforme a sus regulaciones y, en consecuencia, puede requerirse una documentación arquitectónica completa.

6.3 Patrones arquitectónicos

La idea de los patrones como una forma de presentar, compartir y reutilizar el conocimiento sobre los sistemas de software se usa ahora ampliamente. El origen de esto fue la publicación de un libro acerca de patrones de diseño orientados a objetos (Gamma *et al*., 1995), que incitó el desarrollo de otros tipos de patrón, como los patrones para el diseño organizacional (Coplien y Harrison, 2004), patrones de usabilidad (Usability Group, 1998), interacción (Martin y Sommerville, 2004), administración de la configuración

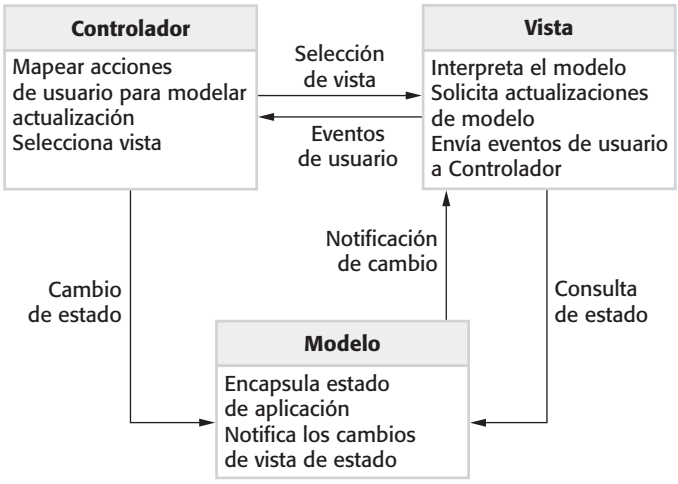


Figura 6.3 La organización del MVC

(Berczuk y Appleton, 2002), etcétera. Los patrones arquitectónicos se propusieron en la década de 1990, con el nombre de "estilos arquitectónicos" (Shaw y Garlan, 1996), en una serie de cinco volúmenes de manuales sobre arquitectura de software orientada a patrones, publicados entre 1996 y 2007 (Buschmann *et al*., 1996; Buschmann *et al*., 2007a; Buschmann *et al*., 2007b; Kircher y Jain, 2004; Schmidt *et al*., 2000).

En esta sección se introducen los patrones arquitectónicos y se describe brevemente una selección de patrones arquitectónicos de uso común en diferentes tipos de sistemas. Para más información de patrones y su uso, debe remitirse a los manuales de patrones publicados.

Un patrón arquitectónico se puede considerar como una descripción abstracta estilizada de buena práctica, que se ensayó y puso a prueba en diferentes sistemas y entornos. De este modo, un patrón arquitectónico debe describir una organización de sistema que ha tenido éxito en sistemas previos. Debe incluir información sobre cuándo es y cuándo no es adecuado usar dicho patrón, así como sobre las fortalezas y debilidades del patrón.

Por ejemplo, la figura 6.2 describe el muy conocido patrón Modelo-Vista-Controlador. Este patrón es el soporte del manejo de la interacción en muchos sistemas basados en la Web. La descripción del patrón estilizado incluye el nombre del patrón, una breve descripción (con un modelo gráfico asociado) y un ejemplo del tipo de sistema donde se usa el patrón (de nuevo, quizá con un modelo gráfico). También debe incluir información sobre cuándo hay que usar el patrón, así como sobre sus ventajas y desventajas. En las figuras 6.3 y 6.4 se presentan los modelos gráficos de la arquitectura asociada con el patrón MVC. En ellas se muestra la arquitectura desde diferentes vistas: la figura 6.3 es una vista conceptual; en tanto que la figura 6.4 ilustra una posible arquitectura en tiempo de operación, cuando este patrón se usa para el manejo de la interacción en un sistema basado en la Web.

En una breve sección de un capítulo general es imposible describir todos los patrones genéricos que se usan en el desarrollo de software. En cambio, se presentan algunos ejemplos seleccionados de patrones que se utilizan ampliamente y captan buenos principios de diseño arquitectónico. En las páginas Web del libro se incluyen más ejemplos acerca de patrones arquitectónicos genéricos.

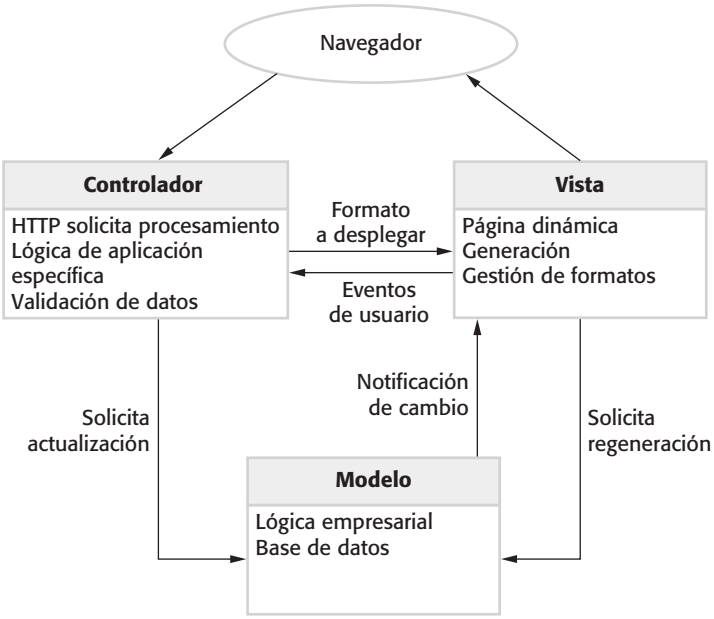


Figura 6.4 Arquitectura de aplicación Web con el patrón MVC

6.3.1 Arquitectura en capas

Las nociones de separación e independencia son fundamentales para el diseño arquitectónico porque permiten localizar cambios. El patrón MVC, que se muestra en la figura 6.2, separa elementos de un sistema, permitiéndoles cambiar de forma independiente. Por ejemplo, agregar una nueva vista o cambiar una vista existente puede hacerse sin modificación alguna a los datos subyacentes en el modelo. El patrón de arquitectura en capas es otra forma de lograr separación e independencia. Este patrón se ilustra en la figura 6.5. Aquí, la funcionalidad del sistema está organizada en capas separadas, y cada una se apoya sólo en las facilidades y los servicios ofrecidos por la capa inmediatamente debajo de ella.

Este enfoque en capas soporta el desarrollo incremental de sistemas. Conforme se desarrolla una capa, algunos de los servicios proporcionados por esta capa deben quedar a disposición de los usuarios. La arquitectura también es cambiable y portátil. En tanto su interfaz no varíe, una capa puede sustituirse por otra equivalente. Más aún, cuando las interfaces de capa cambian o se agregan nuevas facilidades a una capa, sólo resulta afectada la capa adyacente. A medida que los sistemas en capas localizan dependencias de máquina en capas más internas, se facilita el ofrecimiento de implementaciones multiplataforma de un sistema de aplicación. Sólo las capas más internas dependientes de la máquina deben reimplantarse para considerar las facilidades de un sistema operativo o base de datos diferentes.

La figura 6.6 es un ejemplo de una arquitectura en capas con cuatro capas. La capa inferior incluye software de soporte al sistema, por lo general soporte de base de datos y sistema operativo. La siguiente capa es la de aplicación, que comprende los componentes relacionados con la funcionalidad de la aplicación, así como los componentes de utilidad que usan otros componentes de aplicación. La tercera capa se relaciona con la gestión de interfaz del usuario y con brindar autenticación y autorización al usuario, mientras que la

Nombre	Arquitectura en capas
Descripción	Organiza el sistema en capas con funcionalidad relacionada con cada capa. Una capa da servicios a la capa de encima, de modo que las capas de nivel inferior representan servicios núcleo que es probable se utilicen a lo largo de todo el sistema. Véase la figura 6.6.
Ejemplo	Un modelo en capas de un sistema para compartir documentos con derechos de autor se tiene en diferentes bibliotecas, como se ilustra en la figura 6.7.
Cuándo se usa	Se usa al construirse nuevas facilidades encima de los sistemas existentes; cuando el desarrollo se dispersa a través de varios equipos de trabajo, y cada uno es responsable de una capa de funcionalidad; cuando exista un requerimiento para seguridad multinivel.
Ventajas	Permite la sustitución de capas completas en tanto se conserve la interfaz. Para aumentar la confiabilidad del sistema, en cada capa pueden incluirse facilidades redundantes (por ejemplo, autenticación).
Desventajas	En la práctica, suele ser difícil ofrecer una separación limpia entre capas, y es posible que una capa de nivel superior deba interactuar directamente con capas de nivel inferior, en vez de que sea a través de la capa inmediatamente abajo de ella. El rendimiento suele ser un problema, debido a múltiples niveles de interpretación de una solicitud de servicio mientras se procesa en cada capa.

Figura 6.5 Patrón de arquitectura en capas

capa superior proporciona facilidades de interfaz de usuario. Desde luego, es arbitrario el número de capas. Cualquiera de las capas en la figura 6.6 podría dividirse en dos o más capas.

La figura 6.7 es un ejemplo de cómo puede aplicarse este patrón de arquitectura en capas a un sistema de biblioteca llamado LIBSYS, que permite el acceso electrónico controlado a material con derechos de autor de un conjunto de bibliotecas universitarias. Tiene una arquitectura de cinco capas y, en la capa inferior, están las bases de datos individuales en cada biblioteca.

En la figura 6.17 (que se encuentra en la sección 6.4) se observa otro ejemplo de patrón de arquitectura en capas. Muestra la organización del sistema para atención a la salud mental (MHC-PMS) que se estudió en capítulos anteriores.

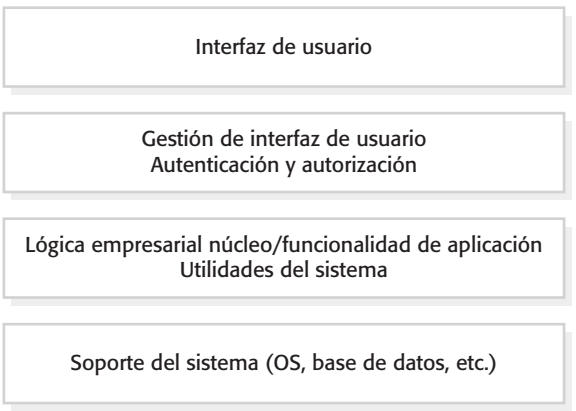


Figura 6.6 Arquitectura genérica en capas

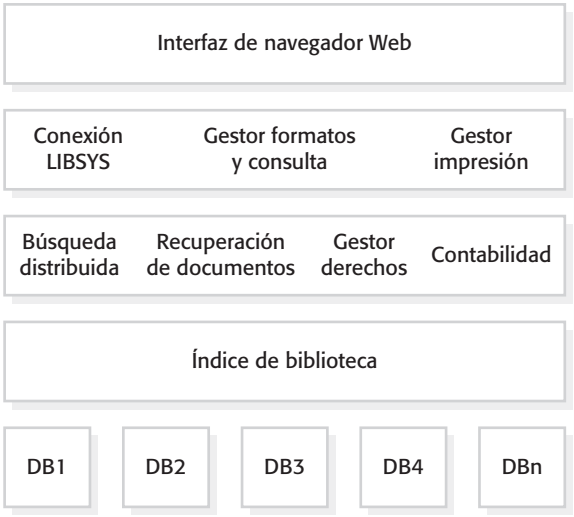


Figura 6.7 Arquitectura del sistema LIBSYS

6.3.2 Arquitectura de repositorio

Los patrones de arquitectura en capas y MVC son ejemplos de patrones en que la vista presentada es la organización conceptual de un sistema. El siguiente ejemplo, el patrón de repositorio (figura 6.8), describe cómo comparte datos un conjunto de componentes en interacción.

La mayoría de los sistemas que usan grandes cantidades de datos se organizan sobre una base de datos o un repositorio compartido. Por lo tanto, este modelo es adecuado

Figura 6.8 El patrón de repositorio

Nombre	Repositorio
Descripción	Todos los datos en un sistema se gestionan en un repositorio central, accesible a todos los componentes del sistema. Los componentes no interactúan directamente, sino tan sólo a través del repositorio.
Ejemplo	La figura 6.9 es un ejemplo de un IDE donde los componentes usan un repositorio de información de diseño de sistema. Cada herramienta de software genera información que, en ese momento, está disponible para uso de otras herramientas.
Cuándo se usa	Este patrón se usa cuando se tiene un sistema donde los grandes volúmenes de información generados deban almacenarse durante mucho tiempo. También puede usarse en sistemas dirigidos por datos, en los que la inclusión de datos en el repositorio active una acción o herramienta.
Ventajas	Los componentes pueden ser independientes, no necesitan conocer la existencia de otros componentes. Los cambios hechos por un componente se pueden propagar hacia todos los componentes. La totalidad de datos se puede gestionar de manera consistente (por ejemplo, respaldos realizados al mismo tiempo), pues todos están en un lugar.
Desventajas	El repositorio es un punto de falla único, de modo que los problemas en el repositorio afectan a todo el sistema. Es posible que haya ineficiencias al organizar toda la comunicación a través del repositorio. Quizá sea difícil distribuir el repositorio por medio de varias computadoras.

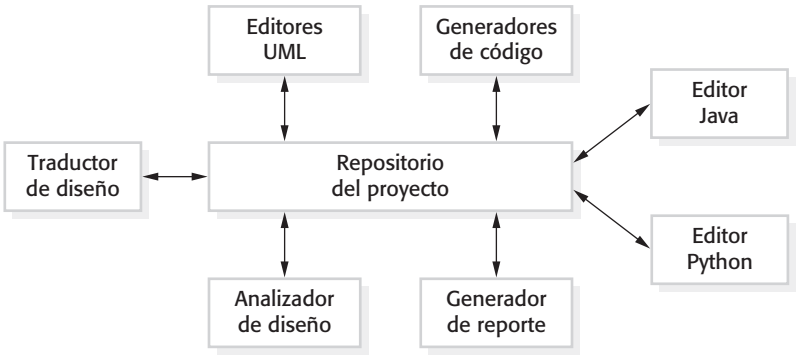


Figura 6.9 Arquitectura de repositorio para un IDE

para aplicaciones en las que un componente genere datos y otro los use. Los ejemplos de este tipo de sistema incluyen sistemas de comando y control, sistemas de información administrativa, sistemas CAD y entornos de desarrollo interactivo para software.

La figura 6.9 ilustra una situación en la que puede usarse un repositorio. Este diagrama muestra un IDE que incluye diferentes herramientas para soportar desarrollo dirigido por modelo. En este caso, el repositorio puede ser un entorno controlado por versión (como se estudia en el capítulo 25) que hace un seguimiento de los cambios al software y permite regresar (*rollback*) a versiones anteriores.

Organizar herramientas alrededor de un repositorio es una forma eficiente de compartir grandes cantidades de datos. No hay necesidad de transmitir explícitamente datos de un componente a otro. Sin embargo, los componentes deben operar en torno a un modelo de repositorio de datos acordado. Inevitablemente, éste es un compromiso entre las necesidades específicas de cada herramienta y sería difícil o imposible integrar nuevos componentes, si sus modelos de datos no se ajustan al esquema acordado. En la práctica, llega a ser complicado distribuir el repositorio sobre un número de máquinas. Aunque es posible distribuir un repositorio lógicamente centralizado, puede haber problemas con la redundancia e inconsistencia de los datos.

En el ejemplo que se muestra en la figura 6.9, el repositorio es pasivo, y el control es responsabilidad de los componentes que usan el repositorio. Un enfoque alternativo, que se derivó para sistemas IA, utiliza un modelo “blackboard” (pizarrón) que activa componentes cuando los datos particulares se tornan disponibles. Esto es adecuado cuando la forma de los datos del repositorio está menos estructurada. Las decisiones sobre cuál herramienta activar puede hacerse sólo cuando se hayan analizado los datos. Este modelo lo introdujo Nii (1986). Bosch (2000) incluye un buen análisis de cómo este estilo se relaciona con los atributos de calidad del sistema.

6.3.3 Arquitectura cliente-servidor

El patrón de repositorio se interesa por la estructura estática de un sistema sin mostrar su organización en tiempo de operación. El siguiente ejemplo ilustra una organización en tiempo de operación, de uso muy común para sistemas distribuidos. En la figura 6.10 se describe el patrón cliente-servidor.

Nombre	Cliente-servidor
Descripción	En una arquitectura cliente-servidor, la funcionalidad del sistema se organiza en servicios, y cada servicio lo entrega un servidor independiente. Los clientes son usuarios de dichos servicios y para utilizarlos ingresan a los servidores.
Ejemplo	La figura 6.11 es un ejemplo de una filmoteca y videoteca (videos/DVD) organizada como un sistema cliente-servidor.
Cuándo se usa	Se usa cuando, desde varias ubicaciones, se tiene que ingresar a los datos en una base de datos compartida. Como los servidores se pueden replicar, también se usan cuando la carga de un sistema es variable.
Ventajas	La principal ventaja de este modelo es que los servidores se pueden distribuir a través de una red. La funcionalidad general (por ejemplo, un servicio de impresión) estaría disponible a todos los clientes, así que no necesita implementarse en todos los servicios.
Desventajas	Cada servicio es un solo punto de falla, de modo que es susceptible a ataques de rechazo de servicio o a fallas del servidor. El rendimiento resultará impredecible porque depende de la red, así como del sistema. Quizás haya problemas administrativos cuando los servidores sean propiedad de diferentes organizaciones.

Figura 6.10 Patrón cliente-servidor

Un sistema que sigue el patrón cliente-servidor se organiza como un conjunto de servicios y servidores asociados, y de clientes que acceden y usan los servicios. Los principales componentes de este modelo son:

1. Un conjunto de servidores que ofrecen servicios a otros componentes. Ejemplos de éstos incluyen servidores de impresión; servidores de archivo que brindan servicios de administración de archivos, y un servidor compilador, que proporciona servicios de compilación de lenguaje de programación.
2. Un conjunto de clientes que solicitan los servicios que ofrecen los servidores. Habrá usualmente varias instancias de un programa cliente que se ejecuten de manera concurrente en diferentes computadoras.
3. Una red que permite a los clientes acceder a dichos servicios. La mayoría de los sistemas cliente-servidor se implementan como sistemas distribuidos, conectados mediante protocolos de Internet.

Las arquitecturas cliente-servidor se consideran a menudo como arquitecturas de sistemas distribuidos; sin embargo, el modelo lógico de servicios independientes que opera en servidores separados puede implementarse en una sola computadora. De nuevo, un beneficio importante es la separación e independencia. Los servicios y servidores pueden cambiar sin afectar otras partes del sistema.

Es posible que los clientes deban conocer los nombres de los servidores disponibles, así como los servicios que proporcionan. Sin embargo, los servidores no necesitan conocer la identidad de los clientes o cuántos clientes acceden a sus servicios. Los clientes acceden a los servicios que proporciona un servidor, a través de llamadas a procedimiento remoto usando un protocolo solicitud-respuesta, como el protocolo http utilizado

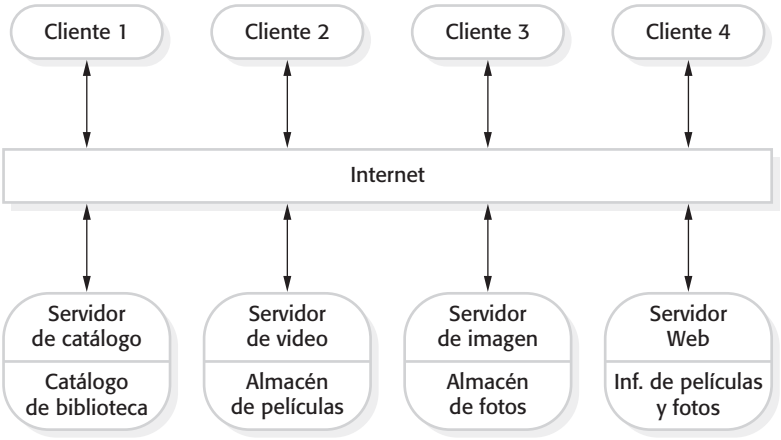


Figura 6.11 Arquitectura cliente-servidor para una filmoteca

en la WWW. En esencia, un cliente realiza una petición a un servidor y espera hasta que recibe una respuesta.

La figura 6.11 es un ejemplo de un sistema que se basa en el modelo cliente-servidor. Se trata de un sistema multiusuario basado en la Web, para ofrecer un repertorio de películas y fotografías. En este sistema, varios servidores manejan y despliegan los diferentes tipos de medios. Los cuadros de video necesitan transmitirse rápidamente y en sincronía, aunque a una resolución relativamente baja. Tal vez estén comprimidos en un almacén, de manera que el servidor de video puede manipular en diferentes formatos la compresión y descompresión del video. Sin embargo, las imágenes fijas deben conservarse en una resolución alta, por lo que es adecuado mantenerlas en un servidor independiente.

El catálogo debe manejar una variedad de consultas y ofrecer vínculos hacia el sistema de información Web, que incluye datos acerca de las películas y los videos, así

Figura 6.12 Patrón de tubería y filtro (*pipe and filter*)

Nombre	Tubería y filtro (*pipe and filter*)
Descripción	El procesamiento de datos en un sistema se organiza de forma que cada componente de procesamiento (filtro) sea discreto y realice un tipo de transformación de datos. Los datos fluyen (como en una tubería) de un componente a otro para su procesamiento.
Ejemplo	La figura 6.13 es un ejemplo de un sistema de tubería y filtro usado para el procesamiento de facturas.
Cuándo se usa	Se suele utilizar en aplicaciones de procesamiento de datos (tanto basadas en lotes [*batch*] como en transacciones), donde las entradas se procesan en etapas separadas para generar salidas relacionadas.
Ventajas	Fácil de entender y soporta reutilización de transformación. El estilo del flujo de trabajo coincide con la estructura de muchos procesos empresariales. La evolución al agregar transformaciones es directa. Puede implementarse como un sistema secuencial o como uno concurrente.
Desventajas	El formato para la transferencia de datos debe acordarse entre las transformaciones que se comunican. Cada transformación debe analizar sus entradas y sintetizar sus salidas al formato acordado. Esto aumenta la carga del sistema, y puede significar que sea imposible reutilizar transformaciones funcionales que usen estructuras de datos incompatibles.

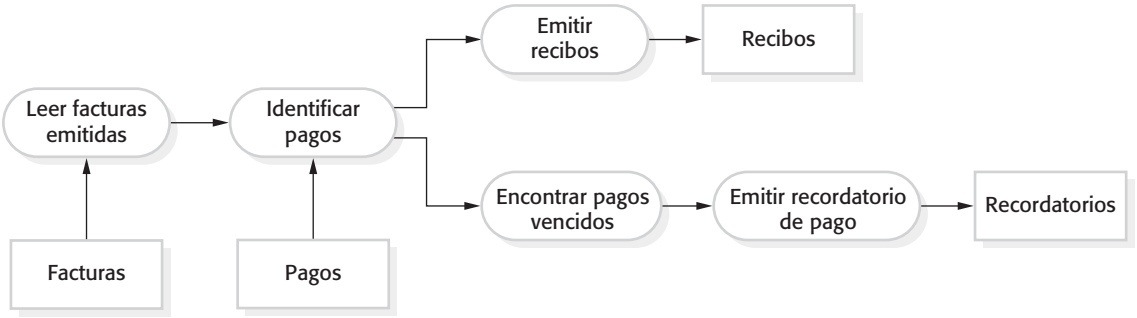


Figura 6.13 Ejemplo de arquitectura de tubería y filtro

como un sistema de comercio electrónico que soporte la venta de fotografías, películas y videos. El programa cliente es simplemente una interfaz integrada de usuario, construida mediante un navegador Web, para acceder a dichos servicios.

La ventaja más importante del modelo cliente-servidor consiste en que es una arquitectura distribuida. Éste puede usarse de manera efectiva en sistemas en red con distintos procesadores distribuidos. Es fácil agregar un nuevo servidor e integrarlo al resto del sistema, o bien, actualizar de manera clara servidores sin afectar otras partes del sistema. En el capítulo 18 se estudian las arquitecturas distribuidas, incluidas las arquitecturas cliente-servidor y las arquitecturas de objeto distribuidas.

6.3.4 Arquitectura de tubería y filtro

El ejemplo final de un patrón arquitectónico es el patrón tubería y filtro (*pipe and filter*). Éste es un modelo de la organización en tiempo de operación de un sistema, donde las transformaciones funcionales procesan sus entradas y producen salidas. Los datos fluyen de uno a otro y se transforman conforme se desplazan a través de la secuencia. Cada paso de procesamiento se implementa como un transformador. Los datos de entrada fluyen por medio de dichos transformadores hasta que se convierten en salida. Las transformaciones pueden ejecutarse secuencialmente o en forma paralela. Es posible que los datos se procesen por cada transformador ítem por ítem o en un solo lote.

El nombre "tubería y filtro" proviene del sistema Unix original, donde era posible vincular procesos empleando "tuberías". Por ellas pasaba una secuencia de texto de un proceso a otro. Los sistemas conformados con este modelo pueden implementarse al combinar comandos Unix, usando las tuberías y las instalaciones de control del intérprete de comandos Unix. Se usa el término "filtro" porque una transformación "filtra" los datos que puede procesar de su secuencia de datos de entrada.

Se han utilizado variantes de este patrón desde que se usaron por primera vez computadoras para el procesamiento automático de datos. Cuando las transformaciones son secuenciales, con datos procesados en lotes, este modelo arquitectónico de tubería y filtro se convierte en un modelo secuencial en lote, una arquitectura común para sistemas de procesamiento de datos (por ejemplo, un sistema de facturación). La arquitectura de un sistema embebido puede organizarse también como un proceso por entubamiento, donde cada proceso se ejecuta de manera concurrente. En el capítulo 20 se estudia el uso de este patrón en sistemas embebidos.

En la figura 6.13 se muestra un ejemplo de este tipo de arquitectura de sistema, que se usa en una aplicación de procesamiento en lote. Una organización emite facturas a los clientes. Una vez a la semana, los pagos efectuados se incorporan a las facturas. Para

Patrones arquitectónicos para control

Existen patrones arquitectónicos específicos que reflejan formas usadas comúnmente para organizar el control en un sistema. En ellos se incluyen el control centralizado, basado en un componente que llama otros componentes, y el control con base en un evento, donde el sistema reacciona a eventos externos.

http://www.SoftwareEngineering-9.com/Web/Architecture/ArchPatterns/

las facturas pagadas se emite un recibo. Para las facturas no saldadas dentro del plazo de pago se emite un recordatorio.

Los sistemas interactivos son difíciles de escribir con el modelo tubería y filtro, debido a la necesidad de procesar una secuencia de datos. Aunque las entradas y salidas textuales simples pueden modelarse de esta forma, las interfaces gráficas de usuario tienen formatos I/O más complejos, así como una estrategia de control que se basa en eventos como clics del mouse o selecciones del menú. Es difícil traducir esto en una forma compatible con el modelo *pipelining* (entubamiento).

6.4 Arquitecturas de aplicación

Los sistemas de aplicación tienen la intención de cubrir las necesidades de una empresa u organización. Todas las empresas tienen mucho en común: necesitan contratar personal, emitir facturas, llevar la contabilidad, etcétera. Las empresas que operan en el mismo sector usan aplicaciones comunes específicas para el sector. De esta forma, además de las funciones empresariales generales, todas las compañías telefónicas necesitan sistemas para conectar llamadas, administrar sus redes y emitir facturas a los clientes, entre otros. En consecuencia, también los sistemas de aplicación que utilizan dichas empresas tienen mucho en común.

Estos factores en común condujeron al desarrollo de arquitecturas de software que describen la estructura y la organización de tipos particulares de sistemas de software. Las arquitecturas de aplicación encapsulan las principales características de una clase de sistemas. Por ejemplo, en los sistemas de tiempo real puede haber modelos arquitectónicos genéricos de diferentes tipos de sistema, tales como sistemas de recolección de datos o sistemas de monitorización. Aunque las instancias de dichos sistemas difieren en detalle, la estructura arquitectónica común puede reutilizarse cuando se desarrollen nuevos sistemas del mismo tipo.

La arquitectura de aplicación puede reimplantarse cuando se desarrollen nuevos sistemas, pero, para diversos sistemas empresariales, la reutilización de aplicaciones es posible sin reimplementación. Esto se observa en el crecimiento de los sistemas de planeación de recursos empresariales (ERP, por las siglas de *Enterprise Resource Planning*) de compañías como SAP y Oracle, y paquetes de software vertical (COTS) para aplicaciones especializadas en diferentes áreas de negocios. En dichos sistemas, un sistema genérico se configura y adapta para crear una aplicación empresarial específica.

Arquitecturas de aplicación

En el sitio Web del libro existen muchos ejemplos de arquitecturas de aplicación. Se incluyen descripciones de sistemas de procesamiento de datos en lote, sistemas de asignación de recursos y sistemas de edición basados en eventos.

http://www.SoftwareEngineering-9.com/Web/Architecture/AppArch/

Por ejemplo, un sistema para suministrar la administración en cadena se adapta a diferentes tipos de proveedores, bienes y arreglos contractuales.

Como diseñador de software, usted puede usar modelos de arquitecturas de aplicación en varias formas:

1. *Como punto de partida para el proceso de diseño arquitectónico* Si no está familiarizado con el tipo de aplicación que desarrolla, podría basar su diseño inicial en una arquitectura de aplicación genérica. Desde luego, ésta tendrá que ser especializada para el sistema específico que se va a desarrollar, pero es un buen comienzo para el diseño.
2. *Como lista de verificación del diseño* Si usted desarrolló un diseño arquitectónico para un sistema de aplicación, puede comparar éste con la arquitectura de aplicación genérica y luego, verificar que su diseño sea consistente con la arquitectura genérica.
3. *Como una forma de organizar el trabajo del equipo de desarrollo* Las arquitecturas de aplicación identifican características estructurales estables de las arquitecturas del sistema y, en muchos casos, es posible desarrollar éstas en paralelo. Puede asignar trabajo a los miembros del grupo para implementar diferentes componentes dentro de la arquitectura.
4. *Como un medio para valorar los componentes a reutilizar* Si tiene componentes por reutilizar, compare éstos con las estructuras genéricas para saber si existen componentes similares en la arquitectura de aplicación.
5. *Como un vocabulario para hablar acerca de los tipos de aplicaciones* Si discute acerca de una aplicación específica o trata de comparar aplicaciones del mismo tipo, entonces puede usar los conceptos identificados en la arquitectura genérica para hablar sobre las aplicaciones.

Hay muchos tipos de sistema de aplicación y, en algunos casos, parecerían muy diferentes. Sin embargo, muchas de estas aplicaciones distintas superficialmente en realidad tienen mucho en común y, por ende, suelen representarse mediante una sola arquitectura de aplicación abstracta. Esto se ilustra aquí al describir las siguientes arquitecturas de dos tipos de aplicación:

1. *Aplicaciones de procesamiento de transacción* Este tipo de aplicaciones son aplicaciones centradas en bases de datos, que procesan los requerimientos del usuario mediante la información y actualizan ésta en una base de datos. Se trata del tipo más común de sistemas empresariales interactivos. Se organizan de tal forma que las acciones del usuario no pueden interferir unas con otras y se mantiene la integridad de la base

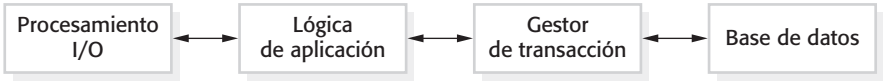


Figura 6.14 Estructura de las aplicaciones de procesamiento de transacción

de datos. Esta clase de sistema incluye los sistemas bancarios interactivos, sistemas de comercio electrónico, sistemas de información y sistemas de reservaciones.

2. *Sistemas de procesamiento de lenguaje* Son sistemas en los que las intenciones del usuario se expresan en un lenguaje formal (como Java). El sistema de procesamiento de lenguaje elabora este lenguaje en un formato interno y después interpreta dicha representación interna. Los sistemas de procesamiento de lenguaje mejor conocidos son los compiladores, que traducen los programas en lenguaje de alto nivel dentro de un código de máquina. Sin embargo, los sistemas de procesamiento de lenguaje se usan también en la interpretación de lenguajes de comandos para bases de datos y sistemas de información, así como de lenguajes de marcado como XML (Harold y Means, 2002; Hunter *et al*., 2007).

Se eligieron estos tipos particulares de sistemas porque una gran cantidad de sistemas empresariales basados en la Web son sistemas de procesamiento de transacciones, y todo el desarrollo del software se apoya en los sistemas de procesamiento de lenguaje.

6.4.1 Sistemas de procesamiento de transacciones

Los sistemas de procesamiento de transacciones (TP, por las siglas de *Transaction Processing*) están diseñados para procesar peticiones del usuario mediante la información de una base de datos, o los requerimientos para actualizar una base de datos (Lewis *et al*., 2003). Técnicamente, una transacción de base de datos es una secuencia de operaciones que se trata como una sola unidad (una unidad atómica). Todas las operaciones en una transacción tienen que completarse antes de que sean permanentes los cambios en la base de datos. Esto garantiza que la falla en las operaciones dentro de la transacción no conduzca a inconsistencias en la base de datos.

Desde una perspectiva de usuario, una transacción es cualquier secuencia coherente de operaciones que satisfacen un objetivo, como "encontrar los horarios de vuelos de Londres a París". Si la transacción del usuario no requiere el cambio en la base de datos, entonces sería innecesario empaquetar esto como una transacción técnica de base de datos.

Un ejemplo de una transacción es una petición de cliente para retirar dinero de una cuenta bancaria mediante un cajero automático. Esto incluye obtener detalles de la cuenta del cliente, verificar y modificar el saldo por la cantidad retirada y enviar comandos al cajero automático para entregar el dinero. Hasta que todos estos pasos se completan, la transacción permanece inconclusa y no cambia la base de datos de la cuenta del cliente.

Por lo general, los sistemas de procesamiento de transacción son sistemas interactivos donde los usuarios hacen peticiones asíncronas de servicios. La figura 6.14 ilustra la estructura arquitectónica conceptual de las aplicaciones de TP. Primero, un usuario hace una petición al sistema a través de un componente de procesamiento I/O. La petición se procesa mediante alguna lógica específica de la aplicación. Se crea una transacción y pasa hacia un gestor de transacciones que, por lo general, está embebido en el sistema de manejo de la

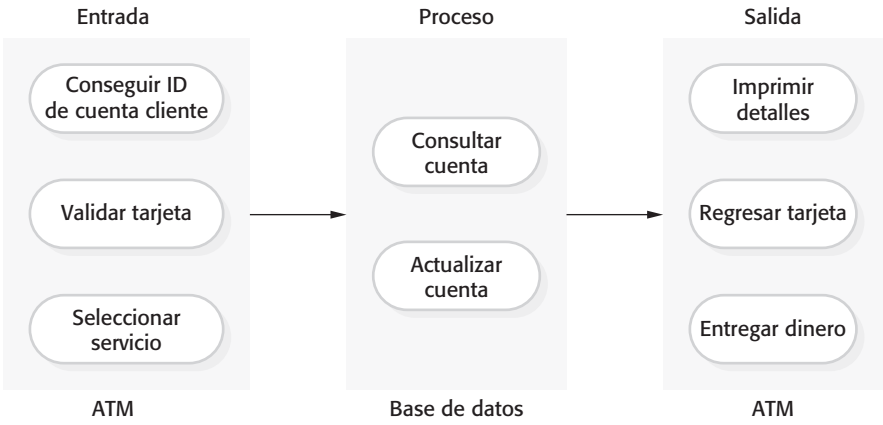


Figura 6.15 Arquitectura de software de un sistema ATM

base de datos. Después de que el gestor de transacciones asegura que la transacción se ha completado adecuadamente, señala a la aplicación que terminó el procesamiento.

Los sistemas de procesamiento de transacción pueden organizarse como una arquitectura "tubería y filtro" con componentes de sistema responsables de entradas, procesamiento y salida. Por ejemplo, considere un sistema bancario que permite a los clientes consultar sus cuentas y retirar dinero de un cajero automático. El sistema está constituido por dos componentes de software cooperadores: el software del cajero automático y el software de procesamiento de cuentas en el servidor de la base de datos del banco. Los componentes de entrada y salida se implementan como software en el cajero automático y el componente de procesamiento es parte del servidor de la base de datos del banco. La figura 6.15 muestra la arquitectura de este sistema e ilustra las funciones de los componentes de entrada, proceso y salida.

6.4.2 Sistemas de información

Todos los sistemas que incluyen interacción con una base de datos compartida se consideran sistemas de información basados en transacciones. Un sistema de información permite acceso controlado a una gran base de información, tales como un catálogo de biblioteca, un horario de vuelos o los registros de pacientes en un hospital. Cada vez más, los sistemas de información son sistemas basados en la Web, cuyo acceso es mediante un navegador Web.

La figura 6.16 presenta un modelo muy general de un sistema de información. El sistema se modela con un enfoque por capas (estudiado en la sección 6.3), donde la capa superior soporta la interfaz de usuario, y la capa inferior es la base de datos del sistema. La capa de comunicaciones con el usuario maneja todas las entradas y salidas de la interfaz de usuario, y la capa de recuperación de información incluye la lógica específica de aplicación para acceder y actualizar la base de datos. Como se verá más adelante, las capas en este modelo pueden trazarse directamente hacia servidores dentro de un sistema basado en Internet.

Como ejemplo de una instancia de este modelo en capas, la figura 6.17 muestra la arquitectura del MHC-PMS. Recuerde que este sistema mantiene y administra detalles de los pacientes que consultan médicos especialistas en problemas de salud mental. En el

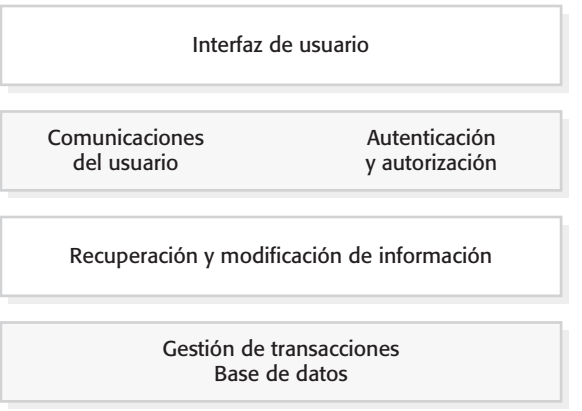


Figura 6.16 Arquitectura de sistema de información en capas

modelo se agregaron detalles a cada capa al identificar los componentes que soportan las comunicaciones del usuario, así como la recuperación y el acceso a la información:

1. La capa superior es responsable de implementar la interfaz de usuario. En este caso, la UI se implementó con el uso de un navegador Web.
2. La segunda capa proporciona la funcionalidad de interfaz de usuario que se entrega a través del navegador Web. Incluye componentes que permiten a los usuarios ingresar al sistema, y componentes de verificación para garantizar que las operaciones utilizadas estén permitidas de acuerdo con su rol. Esta capa incluye componentes de gestión de formato y menú que presentan información a los usuarios, así como componentes de validación de datos que comprueban la consistencia de la información.
3. La tercera capa implementa la funcionalidad del sistema y ofrece componentes que ponen en operación la seguridad del sistema, la creación y actualización de la información del paciente, la importación y exportación de datos del paciente desde otras bases de datos, y los generadores de reporte que elaboran informes administrativos.

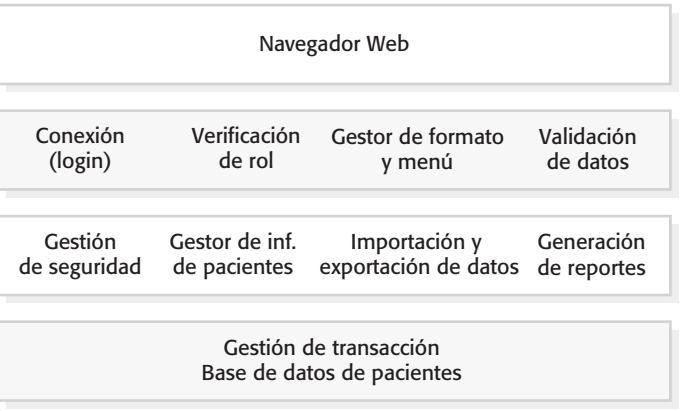


Figura 6.17 Arquitectura del MHC-PMS

4. Finalmente, la capa más baja, que se construye al usar un sistema comercial de gestión de base de datos, ofrece administración de transacciones y almacenamiento constante de datos.

Los sistemas de gestión de información y recursos, por lo general, son ahora sistemas basados en la Web donde las interfaces de usuario se implementan con el uso de un navegador Web. Por ejemplo, los sistemas de comercio electrónico son sistemas de gestión de recursos basados en Internet, que aceptan pedidos electrónicos por bienes o servicios y, luego, ordenan la entrega de dichos bienes o servicios al cliente. En un sistema de comercio electrónico, la capa específica de aplicación incluye funcionalidad adicional que soporta un "carrito de compras", donde los usuarios pueden colocar algunos objetos en transacciones separadas y, luego, pagarlos en una sola transacción.

La organización de servidores en dichos sistemas refleja usualmente el modelo genérico en cuatro capas presentadas en la figura 6.16. Dichos sistemas se suelen implementar como arquitecturas cliente-servidor de multinivel, como se estudia en el capítulo 18:

1. El servidor Web es responsable de todas las comunicaciones del usuario, y la interfaz de usuario se pone en función mediante un navegador Web;
2. El servidor de aplicación es responsable de implementar la lógica específica de la aplicación, así como del almacenamiento de la información y las peticiones de recuperación;
3. El servidor de la base de datos mueve la información hacia y desde la base de datos y, además, manipula la gestión de transacciones.

El uso de múltiples servidores permite un rendimiento elevado, al igual que posibilita la manipulación de cientos de transacciones por minuto. Conforme aumenta la demanda, pueden agregarse servidores en cada nivel, para lidiar con el procesamiento adicional implicado.

6.4.3 Sistemas de procesamiento de lenguaje

Los sistemas de procesamiento de lenguaje convierten un lenguaje natural o artificial en otra representación del lenguaje y, para lenguajes de programación, también pueden ejecutar el código resultante. En ingeniería de software, los compiladores traducen un lenguaje de programación artificial en código de máquina. Otros sistemas de procesamiento de lenguaje traducen una descripción de datos XML en comandos para consultar una base de datos o una representación XML alternativa. Los sistemas de procesamiento de lenguaje natural pueden transformar un lenguaje natural a otro, por ejemplo, francés a noruego.

En la figura 6.18 se ilustra una posible arquitectura para un sistema de procesamiento de lenguaje hacia un lenguaje de programación. Las instrucciones en el lenguaje fuente definen el programa a ejecutar, en tanto que un traductor las convierte en instrucciones para una máquina abstracta. Luego, dichas instrucciones se interpretan mediante otro componente que lee (*fetch*) las instrucciones para su ejecución y las ejecuta al usar (si es necesario) datos del entorno. La salida del proceso es el resultado de la interpretación de instrucciones en los datos de entrada.

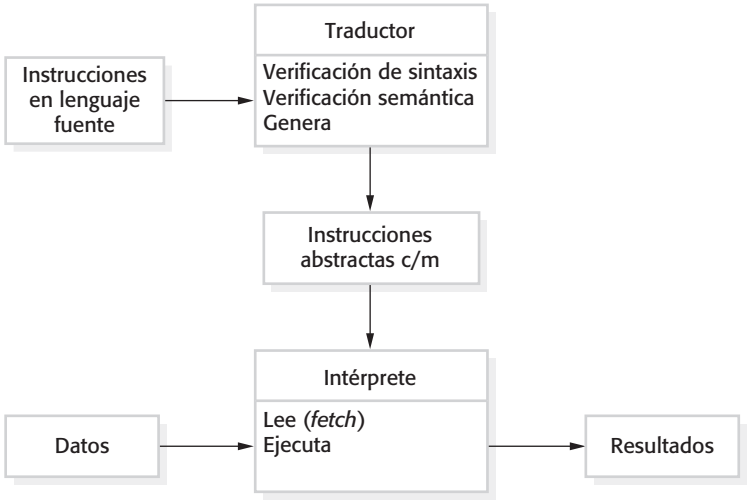


Figura 6.18 Arquitectura de un sistema de procesamiento de lenguaje

Desde luego, para muchos compiladores, el intérprete es una unidad de hardware que procesa instrucciones de la máquina, en tanto que la máquina abstracta es el procesador real. Sin embargo, para lenguajes escritos de manera dinámica, como Python, el intérprete puede ser un componente de software.

Los compiladores de lenguaje de programación que forman parte de un entorno de programación más general tienen una arquitectura genérica (figura 6.19) que incluye los siguientes componentes:

1. Un analizador léxico, que toma valores simbólicos (*tokens*) y los convierte en una forma interna.
2. Una tabla de símbolos, que contiene información de los nombres de las entidades (variables, de clase, de objeto, etcétera) usados en el texto que se traduce.
3. Un analizador de sintaxis, el cual verifica la sintaxis del lenguaje que se va a traducir. Emplea una gramática definida del lenguaje y construye un árbol de sintaxis.
4. Un árbol de sintaxis es una estructura interna que representa el programa a compilar.

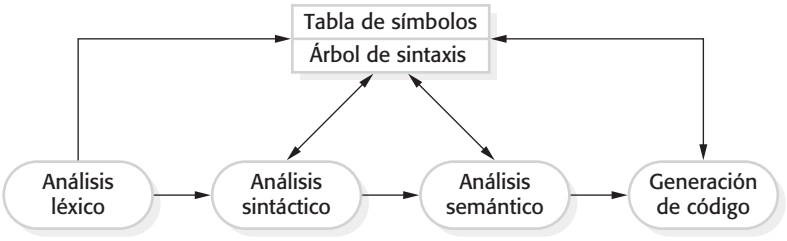


Figura 6.19 Una arquitectura de compilador de tubería y filtro

Arquitecturas de referencia

Las arquitecturas de referencia captan en un dominio características importantes de las arquitecturas del sistema. En esencia, incluyen todo lo que pueda estar en una arquitectura de aplicación aunque, en realidad, es muy improbable que alguna aplicación individual contenga todas las características mostradas en una arquitectura de referencia. El objetivo principal de las arquitecturas de referencia consiste en evaluar y comparar las propuestas de diseño y, en dicho dominio, educar a las personas sobre las características arquitectónicas.

http://www.SoftwareEngineering-9.com/Web/Architecture/RefArch.html

5. Un analizador semántico que usa información del árbol de sintaxis y la tabla de símbolos, para verificar la exactitud semántica del texto en lenguaje de entrada.
6. Un generador de código que "recorre" el árbol de sintaxis y genera un código de máquina abstracto.

También pueden incluirse otros componentes que analizan y transforman el árbol de sintaxis para mejorar la eficiencia y remover redundancia del código de máquina generado. En otros tipos de sistema de procesamiento de lenguaje, como un traductor de lenguaje natural, habrá componentes adicionales, por ejemplo, un diccionario, y el código generado en realidad es el texto de entrada traducido en otro lenguaje.

Existen patrones arquitectónicos alternativos que pueden usarse en un sistema de procesamiento de lenguaje (Garlan y Shaw, 1993). Pueden implementarse compiladores con una composición de un repositorio y un modelo de tubería y filtro. En una arquitectura de compilador, la tabla de símbolos es un repositorio para datos compartidos. Las fases de análisis léxico, sintáctico y semántico se organizan de manera secuencial, como se muestra en la figura 6.19, y se comunican a través de la tabla de símbolos compartida.

Este modelo de tubería y filtro de compilación de lenguaje es efectivo en entornos *batch*, donde los programas se compilan y ejecutan sin interacción del usuario; por ejemplo, en la traducción de un documento XML a otro. Es menos efectivo cuando un compilador se integra con otras herramientas de procesamiento de lenguaje, como un sistema de edición estructurado, un depurador interactivo o un programa de impresión estética (*prettyprinter*). En esta situación, los cambios de un componente deben reflejarse de inmediato en otros componentes. Por lo tanto, es mejor organizar el sistema en torno a un repositorio, como se muestra en la figura 6.20.

Esta figura ilustra cómo un sistema de procesamiento de lenguaje puede formar parte de un conjunto integrado de herramientas de soporte de programación. En este ejemplo, la tabla de símbolos y el árbol de sintaxis actúan como un almacén de información central. Las herramientas y los fragmentos de herramienta se comunican a través de él. Otra información que en ocasiones se incrusta en las herramientas, como la definición gramática y la definición del formato de salida para el programa, se toma de las herramientas y se coloca en el repositorio. En consecuencia, un editor enfocado en la sintaxis podría verificar que ésta sea correcta mientras se escribe un programa, y un *prettyprinter* puede crear listados del programa en un formato que sea fácil de leer.

Figura 6.20 Arquitectura de repositorio para un sistema de procesamiento de lenguaje

PUNTOS CLAVE

- Una arquitectura de software es una descripción de cómo se organiza un sistema de software. Las propiedades de un sistema, como rendimiento, seguridad y disponibilidad, están influidas por la arquitectura utilizada.
- Las decisiones de diseño arquitectónico incluyen decisiones sobre el tipo de aplicación, la distribución del sistema, los estilos arquitectónicos a usar y las formas en que la arquitectura debe documentarse y evaluarse.
- Las arquitecturas pueden documentarse desde varias perspectivas o diferentes vistas. Las posibles vistas incluyen la conceptual, la lógica, la de proceso, la de desarrollo y la física.
- Los patrones arquitectónicos son medios para reutilizar el conocimiento sobre las arquitecturas de sistemas genéricos. Describen la arquitectura, explican cuándo debe usarse, y exponen sus ventajas y desventajas.
- Los patrones arquitectónicos usados comúnmente incluyen el modelo de vista del controlador, arquitectura en capas, repositorio, cliente-servidor, y tubería y filtro.
- Los modelos genéricos de las arquitecturas de sistemas de aplicación ayudan a entender la operación de las aplicaciones, comparar aplicaciones del mismo tipo, validar diseños del sistema de aplicación y valorar componentes para reutilización a gran escala.
- Los sistemas de procesamiento de transacción son sistemas interactivos que permiten el acceso y la modificación remota de la información, en una base de datos por parte de varios usuarios. Los sistemas de información y los sistemas de gestión de recursos son ejemplos de sistemas de procesamiento de transacciones.
- Los sistemas de procesamiento de lenguaje se usan para traducir textos de un lenguaje a otro y para realizar las instrucciones especificadas en el lenguaje de entrada. Incluyen un traductor y una máquina abstracta que ejecuta el lenguaje generado.

LECTURAS SUGERIDAS

Software Architecture: Perspectives on an Emerging Discipline. Éste fue el primer libro sobre arquitectura de software e incluye una amplia discusión acerca de los diferentes estilos arquitectónicos. (M. Shaw y D. Garlan, Prentice-Hall, 1996.)

Software Architecture in Practice, 2nd ed. Se trata de una cuestión práctica de arquitecturas de software que no exagera los beneficios del diseño arquitectónico. Ofrece una clara razón empresarial sobre por qué son importantes las arquitecturas. (L. Bass, P. Clements y R. Kazman, Addison-Wesley, 2003.)

"The Golden Age of Software Architecture". Este ensayo estudia el desarrollo de la arquitectura de software, desde sus inicios en la década de 1980 hasta su uso actual. Aun cuando tiene poco contenido técnico, presenta un panorama histórico amplio e interesante. (M. Shaw y P. Clements, *IEEE Software,* **21** (2), marzo-abril 2006.) http://dx.doi.org/10.1109/MS.2006.58.

Handbook of Software Architecture. Éste es un trabajo en progreso de Grady Booch, uno de los primeros difusores de la arquitectura de software. Ha documentado las arquitecturas de varios sistemas de software, de manera que el lector puede ver la realidad en vez de abstracción académica. Disponible en la Web y con la intención de aparecer como libro. http://www.handbookofsoftwarearchitecture.com/.

EJERCICIOS

6.1. Cuando se describe un sistema, explique por qué es posible que deba diseñar la arquitectura del sistema antes de completar la especificación de requerimientos.

6.2. Se le pide preparar y entregar una presentación a un administrador no técnico para justificar la contratación de un arquitecto de sistemas para un nuevo proyecto. Escriba una lista que establezca los puntos clave de su presentación. Por supuesto, debe explicar qué se entiende por arquitecto de sistemas.

6.3. Exponga por qué pueden surgir conflictos de diseño cuando se desarrolla una arquitectura para la que tanto los requerimientos de disponibilidad como los de seguridad son los requerimientos no funcionales más importantes.

6.4. Dibuje diagramas que muestren una vista conceptual y una vista de proceso de las arquitecturas de los siguientes sistemas:

Un sistema automatizado de emisión de boletos que utilizan los pasajeros en una estación de ferrocarril.

Un sistema de videoconferencia controlado por computadora, que permita que los datos de video, audio y computadora sean al mismo tiempo visibles a muchos participantes.

Un robot limpiador de pisos cuya función sea asear espacios relativamente despejados, como corredores. El limpiador debe detectar las paredes y otros obstáculos.

6.5. Explique por qué usted usa normalmente muchos patrones arquitectónicos cuando diseña la arquitectura de un sistema grande. Además de la información sobre los patrones estudiados en este capítulo, ¿qué información adicional puede serle útil al diseñar sistemas grandes?

6.6. Sugiera una arquitectura para un sistema (como iTunes) que se use para vender y distribuir música por Internet. ¿Qué patrones arquitectónicos son la base para esta arquitectura?

6.7. Especifique cómo usaría el modelo de referencia de entornos CASE (disponibles en las páginas Web del libro), para comparar los IDE ofrecidos por diferentes proveedores de un lenguaje de programación como Java.

6.8. Con el modelo genérico de un sistema de procesamiento de lenguaje presentado aquí, diseñe la arquitectura de un sistema que acepte comandos en lenguaje natural y los traduzca en consultas de base de datos en un lenguaje como SQL.

6.9. Con el modelo básico de un sistema de información, como se presentó en la figura 6.16, sugiera los componentes que puedan ser parte de un sistema de información que permita a los usuarios consultar información de los vuelos que llegan y salen de un aeropuerto específico.

6.10. ¿Debe existir una profesión separada de "arquitecto de software", cuyo papel sea trabajar de manera independiente con un cliente para diseñar la arquitectura de un sistema de software? Entonces, una compañía de software aparte implementaría el sistema. ¿Cuáles serían las dificultades de establecer tal profesión?

REFERENCIAS

Bass, L., Clements, P. y Kazman, R. (2003). Software Architecture in Practice, 2a ed. Boston: Addison-Wesley.

Berczuk, S. P. y Appleton, B. (2002). Software Configuration Management Patterns: Effective Teamwork, Practical Integration. Boston: Addison-Wesley.

Booch, G. (2009). "Handbook of software architecture". Publicación Web. http://www.handbookofsoftwarearchitecture.com/.

Bosch, J. (2000). Design and Use of Software Architectures. Harlow, UK: Addison-Wesley.

Buschmann, F., Henney, K. y Schmidt, D. C. (2007a). Pattern-oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing. New York: John Wiley & Sons.

Buschmann, F., Henney, K. y Schmidt, D. C. (2007b). Pattern-oriented Software Architecture Volume 5: On Patterns and Pattern Languages. New York: John Wiley & Sons.

Buschmann, F., Meunier, R., Rohnert, H. y Sommerlad, P. (1996). Pattern-oriented Software Architecture Volume 1: A System of Patterns. New York: John Wiley & Sons.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. y Stafford, J. (2002). Documenting Software Architectures: Views and Beyond. Boston: Addison-Wesley.

Coplien, J. H. y Harrison, N. B. (2004). Organizational Patterns of Agile Software Development. Englewood Cliffs, NJ: Prentice Hall.

Gamma, E., Helm, R., Johnson, R. y Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Mass.: Addison-Wesley.

Garlan, D. y Shaw, M. (1993). "An introduction to software architecture". Advances in Software Engineering and Knowledge Engineering, 1 1–39.

Harold, E. R. y Means, W. S. (2002). XML in a Nutshell. Sebastopol. Calif.: O'Reilly.

Hofmeister, C., Nord, R. y Soni, D. (2000). Applied Software Architecture. Boston: Addison-Wesley.

Hunter, D., Rafter, J., Fawcett, J. y Van Der Vlist, E. (2007). Beginning XML, 4a ed. Indianapolis, Ind.: Wrox Press.

Kircher, M. y Jain, P. (2004). Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management. New York: John Wiley & Sons.

Krutchen, P. (1995). "The 4+1 view model of software architecture". IEEE Software, **12** (6), 42–50.

Lange, C. F. J., Chaudron, M. R. V. y Muskens, J. (2006). "UML software description and architecture description". IEEE Software, **23** (2), 40–6.

Lewis, P. M., Bernstein, A. J. y Kifer, M. (2003). Databases and Transaction Processing: An Application-oriented Approach. Boston: Addison-Wesley.

Martin, D. y Sommerville, I. (2004). "Patterns of interaction: Linking ethnomethodology and design". ACM Trans. on Computer-Human Interaction, **11** (1), 59–89.

Nii, H. P. (1986). "Blackboard systems, parts 1 and 2". AI Magazine, **7** (3 y 4), 38–53 y 62–9.

Schmidt, D., Stal, M., Rohnert, H. y Buschmann, F. (2000). Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects. New York: John Wiley & Sons.

Shaw, M. y Garlan, D. (1996). Software Architecture: Perspectives on an Emerging Discipline. Englewood Cliffs, NJ: Prentice Hall.

Usability group. (1998). "Usability patterns". Publicación Web. http://www.it.bton.ac.uk/cil/usability/patterns/.