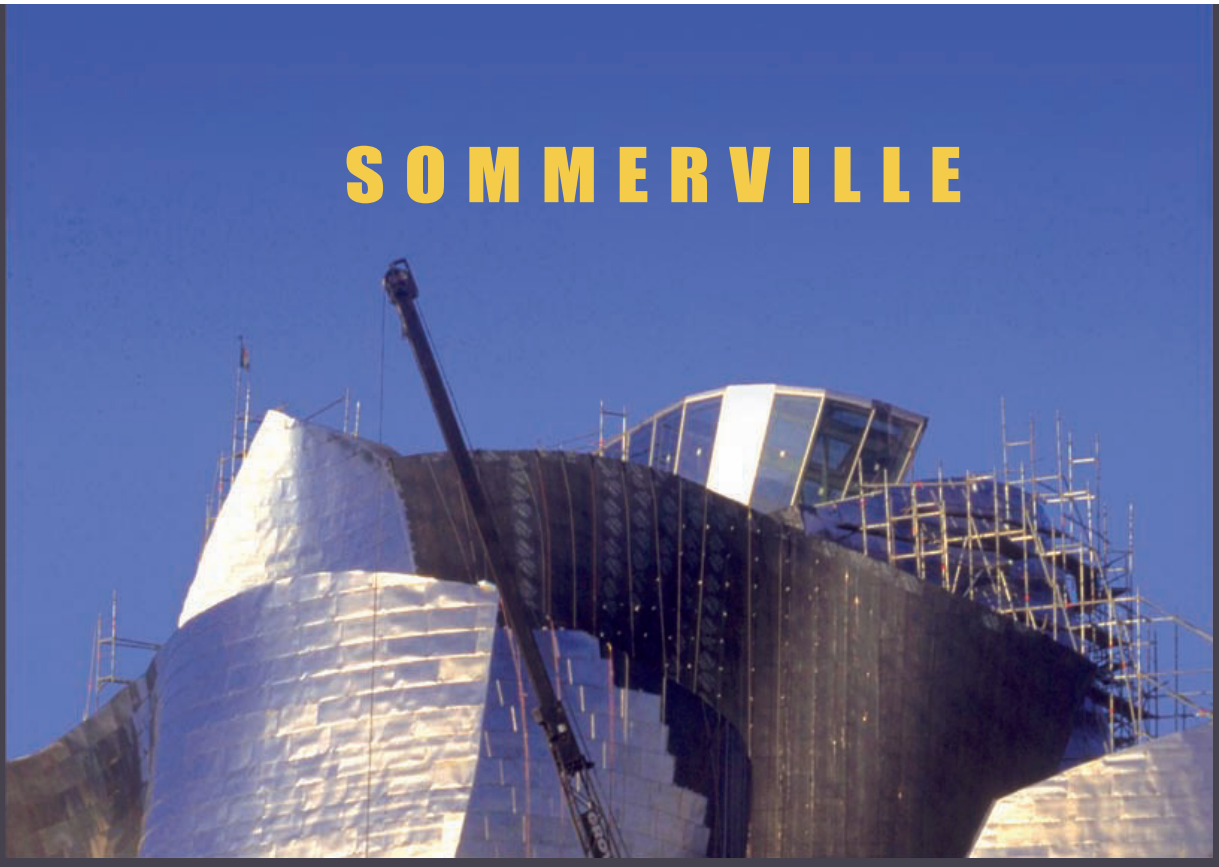
SOMMERVILLE



INGENIERÍA DE SOFTWARE

9

PEARSON

INGENIERÍA DE SOFTWARE

Novena edición

Ian Sommerville

Traducción:
Víctor Campos Olguín
Traductor especialista en Sistemas Computacionales

Revisión técnica:
Sergio Fuenlabrada Velázquez
Edna Martha Miranda Chávez
Miguel Ángel Torres Durán
Mario Alberto Sesma Martínez
Mario Oviedo Galdeano
José Luis López Goytia
Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas-Instituto Politécnico Nacional, México

Darío Guillermo Cardacci
Universidad Abierta Interamericana, Buenos Aires, Argentina

Marcelo Martín Marciszack
Universidad Tecnológica Nacional, Córdoba, Argentina

Addison-Wesley

México • Argentina • Brasil • Colombia • Costa Rica • Chile • Ecuador
España • Guatemala • Panamá • Perú • Puerto Rico • Uruguay • Venezuela

Datos de catalogación bibliográfica

Sommerville, Ian

Ingeniería de Software

PEARSON EDUCACIÓN, México, 2011

ISBN: 978-607-32-0603-7

Área: Computación

Formato: 18.5 × 23.5 cm Páginas: 792

ISBN 9780137035151

Edición en español

Editor: Luis M. Cruz Castillo
e-mail: luis.cruz@pearson.com
Editor de desarrollo: Felipe Hernández Carrasco
Supervisor de producción: Juan José García Guzmán

NOVENA EDICIÓN, 2011

Atlacomulco 500-5o. piso
Col. Industrial Atoto
53519, Naucalpan de Juárez, Estado de México

Cámara Nacional de la Industria Editorial Mexicana. Reg. núm. 1031.

ISBN VERSIÓN IMPRESA: 978-607-32-0603-7
ISBN VERSIÓN E-BOOK: 978-607-32-0604-4
ISBN E-CHAPTER: 978-607-32-0605-1

PRIMERA IMPRESIÓN

Impreso en México. *Printed in Mexico.*

1 2 3 4 5 6 7 8 9 0 - 14 13 12 11

Addison Wesley
es una marca de

PEARSON

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.

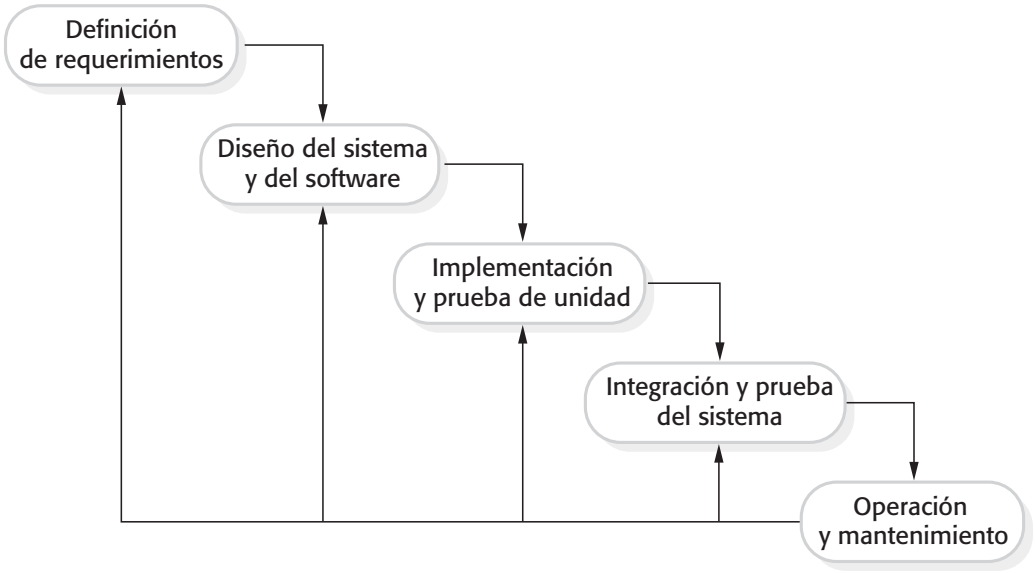


Figura 2.1 El modelo en cascada

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

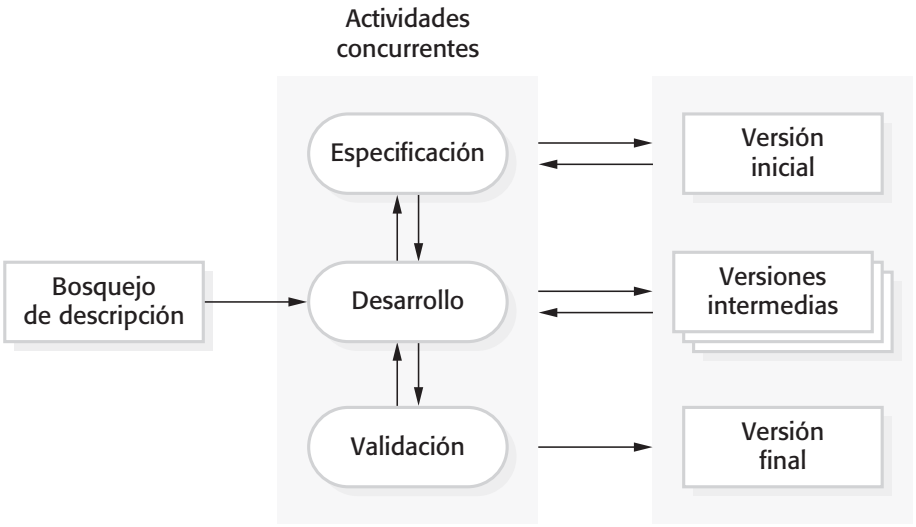


Figura 2.2 Desarrollo incremental

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.

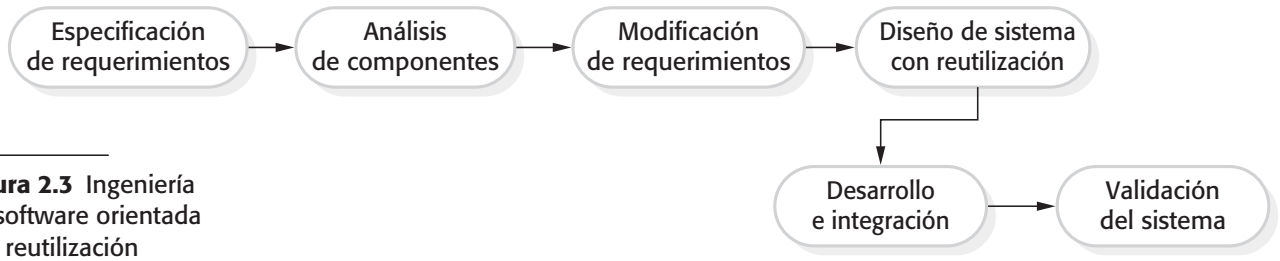


Figura 2.3 Ingeniería de software orientada a la reutilización

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.

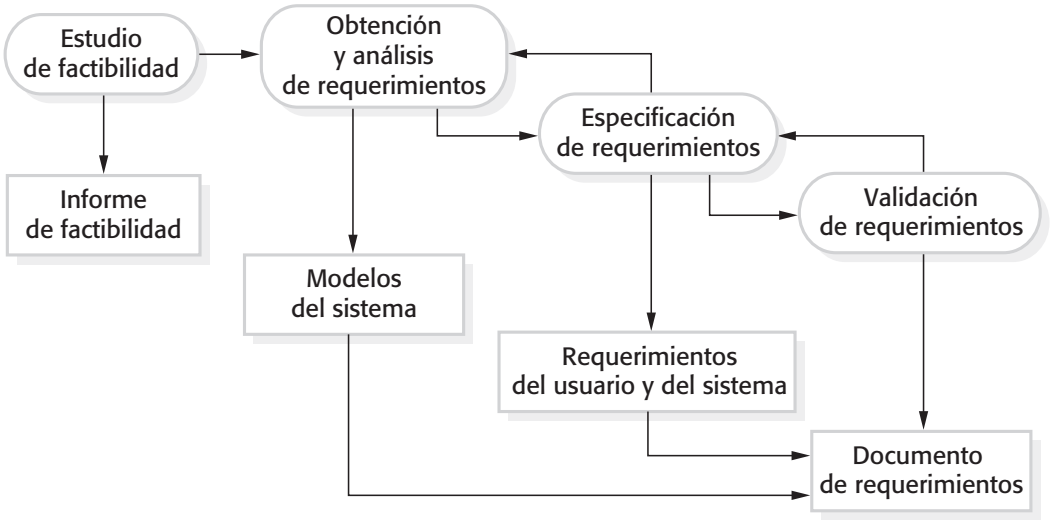


Figura 2.4 Proceso de ingeniería de requerimientos

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.

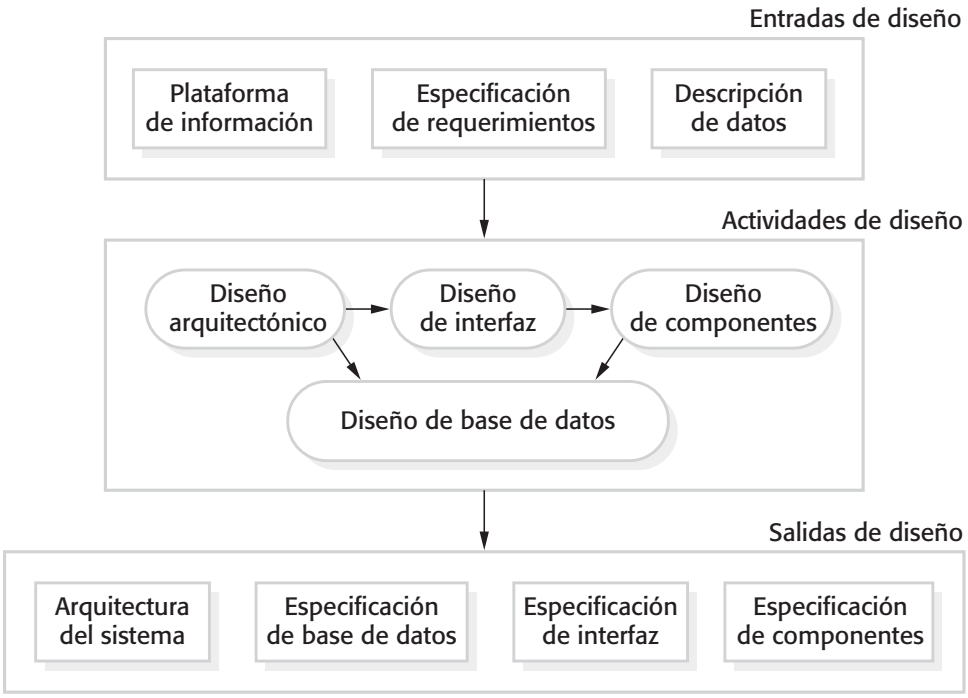


Figura 2.5 Modelo general del proceso de diseño

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,

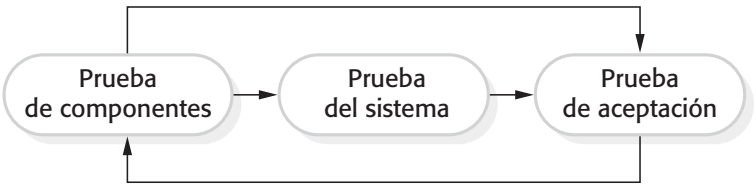


Figura 2.6 Etapas de pruebas

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.

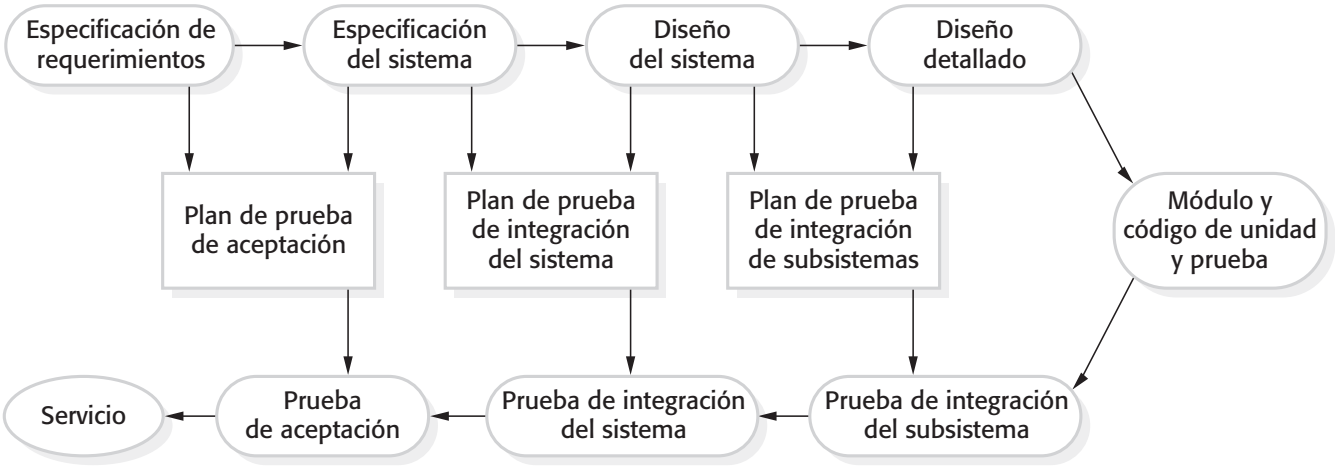


Figura 2.7 Probando fases en un proceso de software dirigido por un plan

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.

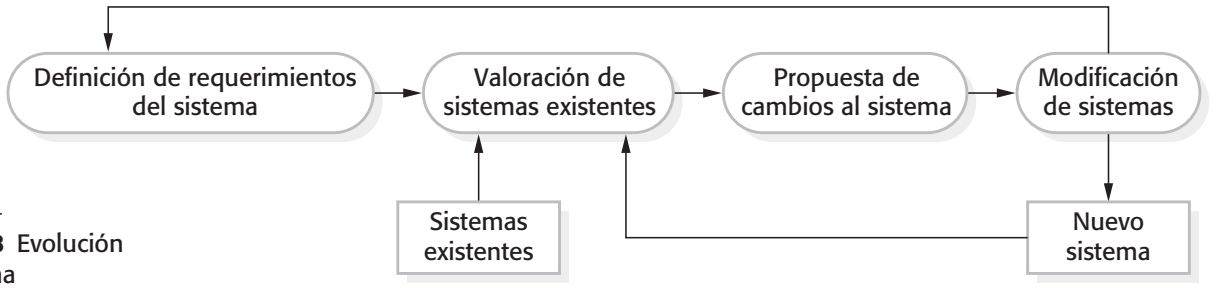


Figura 2.8 Evolución del sistema

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.

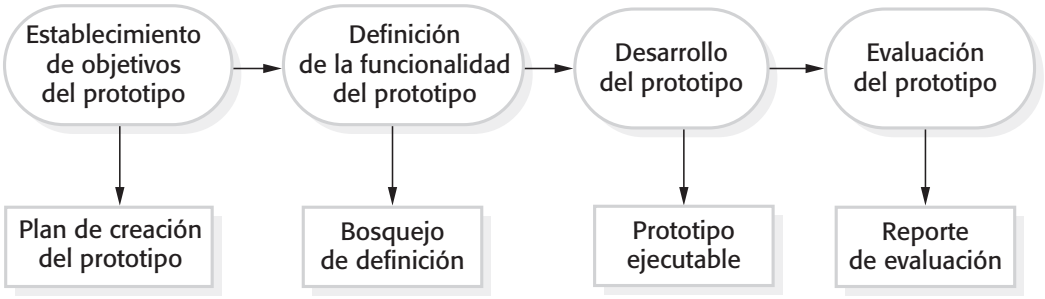


Figura 2.9 Proceso de desarrollo del prototipo

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.

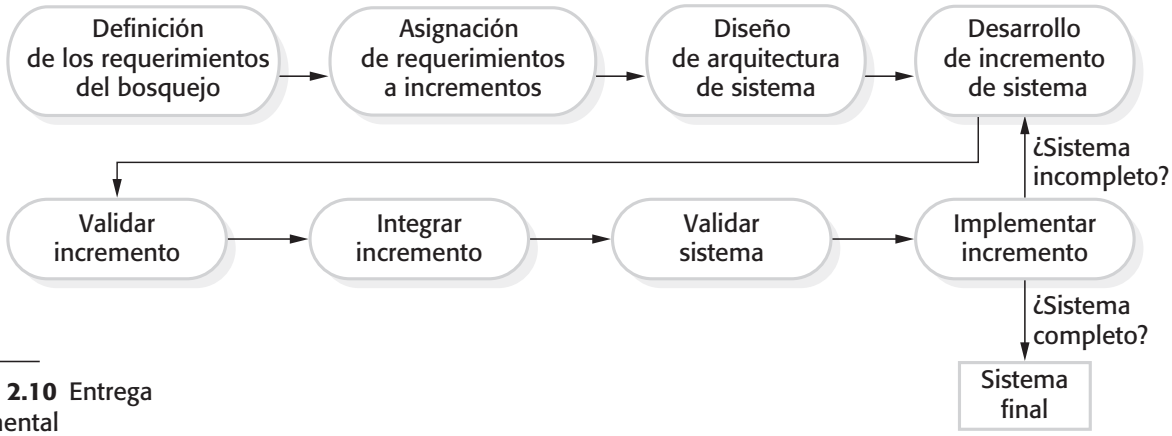


Figura 2.10 Entrega incremental

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.

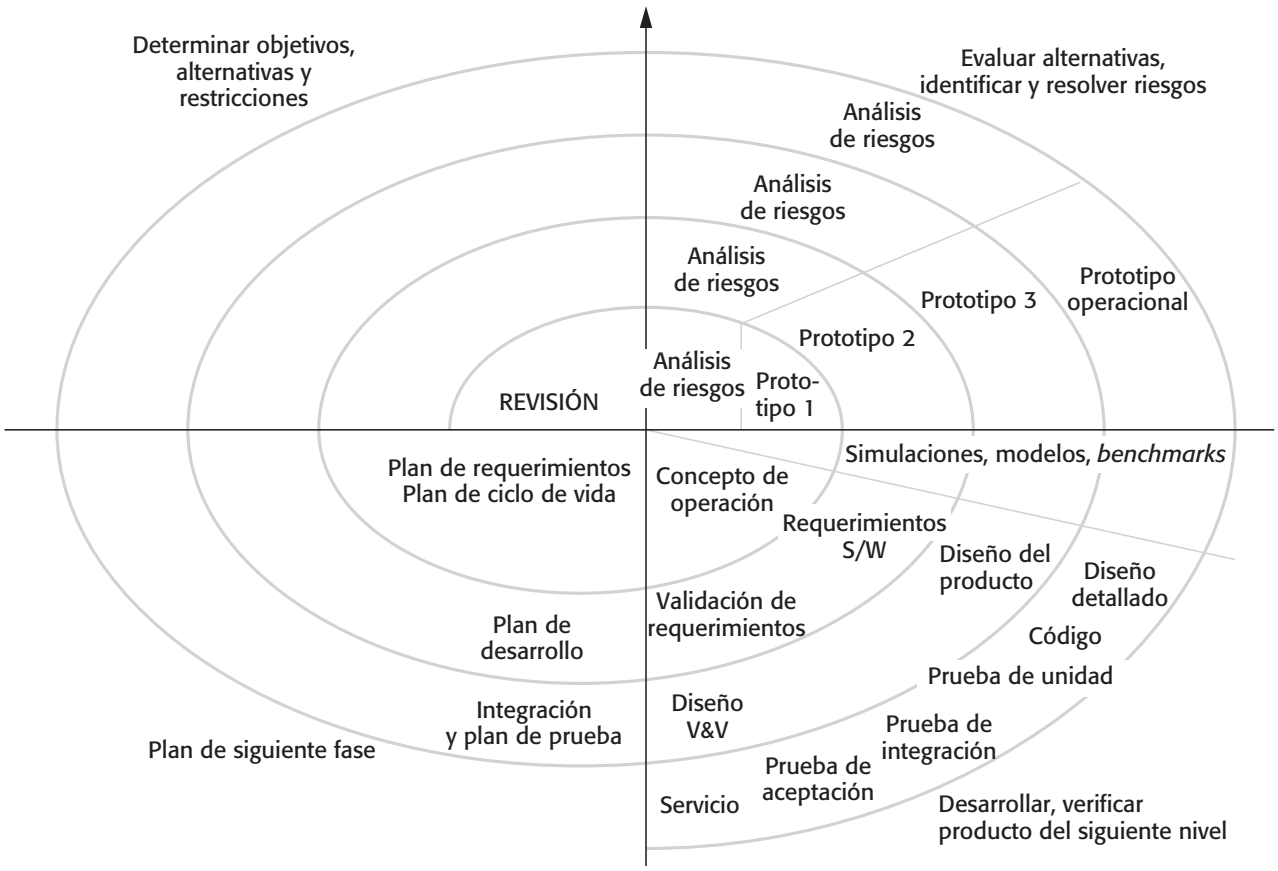


Figura 2.11 Modelo en espiral de Boehm del proceso de software (© IEEE, 1988)

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)

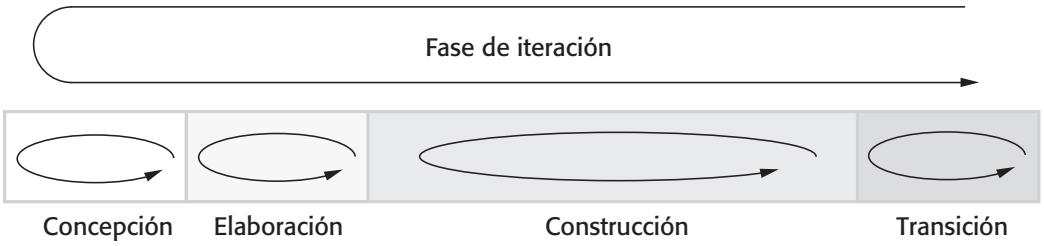


Figura 2.12 Fases en el Proceso Unificado Racional

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.

3

Desarrollo ágil de software

Objetivos

El objetivo de este capítulo es introducirlo a los métodos de desarrollo ágil de software. Al estudiar este capítulo:

- comprenderá las razones de los métodos de desarrollo ágil de software, el manifiesto ágil, así como las diferencias entre el desarrollo ágil y el dirigido por un plan;
- conocerá las prácticas clave en la programación extrema y cómo se relacionan con los principios generales de los métodos ágiles;
- entenderá el enfoque de Scrum para la administración de un proyecto ágil;
- reconocerá los conflictos y problemas de escalar los métodos de desarrollo ágil para el diseño de sistemas de software grandes.

Contenido

Las empresas operan ahora en un entorno global que cambia rápidamente. En ese sentido, deben responder frente a nuevas oportunidades y mercados, al cambio en las condiciones económicas, así como al surgimiento de productos y servicios competitivos. El software es parte de casi todas las operaciones industriales, de modo que el nuevo software se desarrolla rápidamente para aprovechar las actuales oportunidades, con la finalidad de responder ante la amenaza competitiva. En consecuencia, en la actualidad la entrega y el desarrollo rápidos son por lo general el requerimiento fundamental de los sistemas de software. De hecho, muchas empresas están dispuestas a negociar la calidad del software y el compromiso con los requerimientos, para lograr con mayor celeridad la implementación que necesitan del software.

Debido a que dichos negocios funcionan en un entorno cambiante, a menudo es prácticamente imposible derivar un conjunto completo de requerimientos de software estable. Los requerimientos iniciales cambian de modo inevitable, porque los clientes encuentran imposible predecir cómo un sistema afectará sus prácticas operacionales, cómo interactuará con otros sistemas y cuáles operaciones de usuarios se automatizarán. Es posible que sea sólo hasta después de entregar un sistema, y que los usuarios adquieran experiencia con éste, cuando se aclaren los requerimientos reales. Incluso, es probable que debido a factores externos, los requerimientos cambien rápida e impredeciblemente. En tal caso, el software podría ser obsoleto al momento de entregarse.

Los procesos de desarrollo de software que buscan especificar por completo los requerimientos y, luego, diseñar, construir y probar el sistema, no están orientados al desarrollo rápido de software. A medida que los requerimientos cambian, o se descubren problemas en los requerimientos, el diseño o la implementación del sistema tienen que reelaborarse y probarse de nuevo. En consecuencia, un proceso convencional en cascada o uno basado en especificación se prolongan con frecuencia, en tanto que el software final se entrega al cliente mucho después de lo que se especificó originalmente.

En algunos tipos de software, como los sistemas de control críticos para la seguridad, donde es esencial un análisis completo del sistema, resulta oportuno un enfoque basado en un plan. Sin embargo, en un ambiente empresarial de rápido movimiento, esto llega a causar verdaderos problemas. Al momento en que el software esté disponible para su uso, la razón original para su adquisición quizás haya variado tan radicalmente que el software sería inútil a todas luces. Por lo tanto, para sistemas empresariales, son esenciales en particular los procesos de diseño que se enfocan en el desarrollo y la entrega de software rápidos.

Durante algún tiempo, se reconoció la necesidad de desarrollo y de procesos de sistema rápidos que administraran los requerimientos cambiantes. IBM introdujo el desarrollo incremental en la década de 1980 (Mills *et al*., 1980). La entrada de los llamados lenguajes de cuarta generación, también en la misma década, apoyó la idea del software de desarrollo y entrega rápidos (Martin, 1981). Sin embargo, la noción prosperó realmente a finales de la década de 1990, con el desarrollo de la noción de enfoques ágiles como el DSDM (Stapleton, 1997), Scrum (Schwaber y Beedle, 2001) y la programación extrema (Beck, 1999; Beck, 2000).

Los procesos de desarrollo del software rápido se diseñan para producir rápidamente un software útil. El software no se desarrolla como una sola unidad, sino como una serie de incrementos, y cada uno de ellos incluye una nueva funcionalidad del sistema. Aun cuando existen muchos enfoques para el desarrollo de software rápido, comparten algunas características fundamentales:

1. Los procesos de especificación, diseño e implementación están entrelazados. No existe una especificación detallada del sistema, y la documentación del diseño se

minimiza o es generada automáticamente por el entorno de programación que se usa para implementar el sistema. El documento de requerimientos del usuario define sólo las características más importantes del sistema.

2. El sistema se desarrolla en diferentes versiones. Los usuarios finales y otros colaboradores del sistema intervienen en la especificación y evaluación de cada versión. Ellos podrían proponer cambios al software y nuevos requerimientos que se implementen en una versión posterior del sistema.

3. Las interfaces de usuario del sistema se desarrollan usando con frecuencia un sistema de elaboración interactivo, que permita que el diseño de la interfaz se cree rápidamente en cuanto se dibujan y colocan iconos en la interfaz. En tal situación, el sistema puede generar una interfaz basada en la Web para un navegador o una interfaz para una plataforma específica, como Microsoft Windows.

Los métodos ágiles son métodos de desarrollo incremental donde los incrementos son mínimos y, por lo general, se crean las nuevas liberaciones del sistema, y cada dos o tres semanas se ponen a disposición de los clientes. Involucran a los clientes en el proceso de desarrollo para conseguir una rápida retroalimentación sobre los requerimientos cambiantes. Minimizan la cantidad de documentación con el uso de comunicaciones informales, en vez de reuniones formales con documentos escritos.

3.1 Métodos ágiles

En la década de 1980 y a inicios de la siguiente, había una visión muy difundida de que la forma más adecuada para lograr un mejor software era mediante una cuidadosa planeación del proyecto, aseguramiento de calidad formalizada, el uso de métodos de análisis y el diseño apoyado por herramientas CASE, así como procesos de desarrollo de software rigurosos y controlados. Esta percepción proviene de la comunidad de ingeniería de software, responsable del desarrollo de grandes sistemas de software de larga duración, como los sistemas aeroespaciales y gubernamentales.

Este software lo desarrollaron grandes equipos que trabajaban para diferentes compañías. A menudo los equipos estaban geográficamente dispersos y laboraban por largos periodos en el software. Un ejemplo de este tipo de software es el sistema de control de una aeronave moderna, que puede tardar hasta 10 años desde la especificación inicial hasta la implementación. Estos enfoques basados en un plan incluyen costos operativos significativos en la planeación, el diseño y la documentación del sistema. Dichos gastos se justifican cuando debe coordinarse el trabajo de múltiples equipos de desarrollo, cuando el sistema es un sistema crítico y cuando numerosas personas intervendrán en el mantenimiento del software a lo largo de su vida.

Sin embargo, cuando este engorroso enfoque de desarrollo basado en la planeación se aplica a sistemas de negocios pequeños y medianos, los costos que se incluyen son tan grandes que dominan el proceso de desarrollo del software. Se invierte más tiempo en diseñar el sistema, que en el desarrollo y la prueba del programa. Conforme cambian los requerimientos del sistema, resulta esencial la reelaboración y, en principio al menos, la especificación y el diseño deben modificarse con el programa.

En la década de 1990 el descontento con estos enfoques engorrosos de la ingeniería de software condujo a algunos desarrolladores de software a proponer nuevos "métodos

ágiles", los cuales permitieron que el equipo de desarrollo se enfocara en el software en lugar del diseño y la documentación. Los métodos ágiles se apoyan universalmente en el enfoque incremental para la especificación, el desarrollo y la entrega del software. Son más adecuados para el diseño de aplicaciones en que los requerimientos del sistema cambian, por lo general, rápidamente durante el proceso de desarrollo. Tienen la intención de entregar con prontitud el software operativo a los clientes, quienes entonces propondrán requerimientos nuevos y variados para incluir en posteriores iteraciones del sistema. Se dirigen a simplificar el proceso burocrático al evitar trabajo con valor dudoso a largo plazo, y a eliminar documentación que quizá nunca se emplee.

La filosofía detrás de los métodos ágiles se refleja en el manifiesto ágil, que acordaron muchos de los desarrolladores líderes de estos métodos. Este manifiesto afirma:

> *Estamos descubriendo mejores formas para desarrollar software, al hacerlo y al ayudar a otros a hacerlo. Gracias a este trabajo llegamos a valorar:*
>
> > *A los individuos y las interacciones sobre los procesos y las herramientas*
> >
> > *Al software operativo sobre la documentación exhaustiva*
> >
> > *La colaboración con el cliente sobre la negociación del contrato*
> >
> > *La respuesta al cambio sobre el seguimiento de un plan*
>
> *Esto es, aunque exista valor en los objetos a la derecha, valoraremos más los de la izquierda.*

Probablemente el método ágil más conocido sea la programación extrema (Beck, 1999; Beck, 2000), descrita más adelante en este capítulo. Otros enfoques ágiles incluyen los de Scrum (Cohn, 2009; Schwaber, 2004; Schwaber y Beedle, 2001), de Crystal (Cockburn, 2001; Cockburn, 2004), de desarrollo de software adaptativo (Highsmith, 2000), de DSDM (Stapleton, 1997; Stapleton, 2003) y el desarrollo dirigido por características (Palmer y Felsing, 2002). El éxito de dichos métodos condujo a cierta integración con métodos más tradicionales de desarrollo, basados en el modelado de sistemas, lo cual resulta en la noción de modelado ágil (Ambler y Jeffries, 2002) y ejemplificaciones ágiles del Proceso Racional Unificado (Larman, 2002).

Aunque todos esos métodos ágiles se basan en la noción del desarrollo y la entrega incrementales, proponen diferentes procesos para lograrlo. Sin embargo, comparten una serie de principios, según el manifiesto ágil y, por ende, tienen mucho en común. Dichos principios se muestran en la figura 3.1. Diferentes métodos ágiles ejemplifican esos principios en diversas formas; sin embargo, no se cuenta con espacio suficiente para discutir todos los métodos ágiles. En cambio, este texto se enfoca en dos de los métodos usados más ampliamente: programación extrema (sección 3.3) y de Scrum (sección 3.4).

Los métodos ágiles han tenido mucho éxito para ciertos tipos de desarrollo de sistemas:

1. Desarrollo del producto, donde una compañía de software elabora un producto pequeño o mediano para su venta.
2. Diseño de sistemas a la medida dentro de una organización, donde hay un claro compromiso del cliente por intervenir en el proceso de desarrollo, y donde no existen muchas reglas ni regulaciones externas que afecten el software.

Principio	Descripción
Participación del cliente	Los clientes deben intervenir estrechamente durante el proceso de desarrollo. Su función consiste en ofrecer y priorizar nuevos requerimientos del sistema y evaluar las iteraciones del mismo.
Entrega incremental	El software se desarrolla en incrementos y el cliente especifica los requerimientos que se van a incluir en cada incremento.
Personas, no procesos	Tienen que reconocerse y aprovecharse las habilidades del equipo de desarrollo. Debe permitirse a los miembros del equipo desarrollar sus propias formas de trabajar sin procesos establecidos.
Adoptar el cambio	Esperar a que cambien los requerimientos del sistema y, de este modo, diseñar el sistema para adaptar dichos cambios.
Mantener simplicidad	Enfocarse en la simplicidad tanto en el software a desarrollar como en el proceso de desarrollo. Siempre que sea posible, trabajar de manera activa para eliminar la complejidad del sistema.

Figura 3.1 Los principios de los métodos ágiles

Como se analiza en la sección final de este capítulo, el éxito de los métodos ágiles se debe al interés considerable por usar dichos métodos para otros tipos de desarrollo del software. No obstante, dado su enfoque en equipos reducidos firmemente integrados, hay problemas en escalarlos hacia grandes sistemas. También se ha experimentado en el uso de enfoques ágiles para la ingeniería de sistemas críticos (Drobna *et al*., 2004). Sin embargo, a causa de las necesidades de seguridad, protección y análisis de confiabilidad en los sistemas críticos, los métodos ágiles requieren modificaciones significativas antes de usarse cotidianamente con la ingeniería de sistemas críticos.

En la práctica, los principios que subyacen a los métodos ágiles son a veces difíciles de cumplir:

1. Aunque es atractiva la idea del involucramiento del cliente en el proceso de desarrollo, su éxito radica en tener un cliente que desee y pueda pasar tiempo con el equipo de desarrollo, y éste represente a todos los participantes del sistema. Los representantes del cliente están comúnmente sujetos a otras presiones, así que no intervienen por completo en el desarrollo del software.
2. Quizás algunos miembros del equipo no cuenten con la personalidad adecuada para la participación intensa característica de los métodos ágiles y, en consecuencia, no podrán interactuar adecuadamente con los otros integrantes del equipo.
3. Priorizar los cambios sería extremadamente difícil, sobre todo en sistemas donde existen muchos participantes. Cada uno por lo general ofrece diversas prioridades a diferentes cambios.
4. Mantener la simplicidad requiere trabajo adicional. Bajo la presión de fechas de entrega, es posible que los miembros del equipo carezcan de tiempo para realizar las simplificaciones deseables al sistema.

5. Muchas organizaciones, especialmente las grandes compañías, pasan años cambiando su cultura, de tal modo que los procesos se definan y continúen. Para ellas, resulta difícil moverse hacia un modelo de trabajo donde los procesos sean informales y estén definidos por equipos de desarrollo.

Otro problema que no es técnico, es decir, que consiste en un problema general con el desarrollo y la entrega incremental, ocurre cuando el cliente del sistema acude a una organización externa para el desarrollo del sistema. Por lo general, el documento de requerimientos del software forma parte del contrato entre el cliente y el proveedor. Como la especificación incremental es inherente en los métodos ágiles, quizá sea difícil elaborar contratos para este tipo de desarrollo.

Como resultado, los métodos ágiles deben apoyarse en contratos, en los cuales el cliente pague por el tiempo requerido para el desarrollo del sistema, en vez de hacerlo por el desarrollo de un conjunto específico de requerimientos. En tanto todo marche bien, esto beneficia tanto al cliente como al desarrollador. No obstante, cuando surgen problemas, sería difícil discutir acerca de quién es culpable y quién debería pagar por el tiempo y los recursos adicionales requeridos para solucionar las dificultades.

La mayoría de los libros y ensayos que describen los métodos ágiles y las experiencias con éstos hablan del uso de dichos métodos para el desarrollo de nuevos sistemas. Sin embargo, como se explica en el capítulo 9, una enorme cantidad de esfuerzo en ingeniería de software se usa en el mantenimiento y la evolución de los sistemas de software existentes. Hay sólo un pequeño número de reportes de experiencia sobre el uso de métodos ágiles para el mantenimiento de software (Poole y Huisman, 2001). Se presentan entonces dos preguntas que deberían considerarse junto con los métodos y el mantenimiento ágiles:

1. ¿Los sistemas que se desarrollan usando un enfoque ágil se mantienen, a pesar del énfasis en el proceso de desarrollo de minimizar la documentación formal?
2. ¿Los métodos ágiles pueden usarse con efectividad para evolucionar un sistema como respuesta a requerimientos de cambio por parte del cliente?

Se estima que la documentación formal describe el sistema y, por lo tanto, facilita la comprensión a quienes cambian el sistema. Sin embargo, en la práctica, con frecuencia la documentación formal no se conserva actualizada y, por ende, no refleja con precisión el código del programa. Por esta razón, los apasionados de los métodos ágiles argumentan que escribir esta documentación es una pérdida de tiempo y que la clave para implementar software mantenible es producir un código legible de alta calidad. De esta manera, las prácticas ágiles enfatizan la importancia de escribir un código bien estructurado y destinar el esfuerzo en mejorar el código. En consecuencia, la falta de documentación no debe representar un problema para mantener los sistemas desarrollados con el uso de un enfoque ágil.

No obstante, según la experiencia del autor con el mantenimiento de sistemas, éste sugiere que el documento clave es el documento de requerimientos del sistema, el cual indica al ingeniero de software lo que se supone que debe hacer el sistema. Sin tal conocimiento, es difícil valorar el efecto de los cambios propuestos al sistema. Varios métodos ágiles recopilan los requerimientos de manera informal e incremental, aunque sin crear un documento coherente de requerimientos. A este respecto, es probable

que el uso de métodos ágiles haga más difícil y costoso el mantenimiento posterior del sistema.

Es factible que las prácticas ágiles, usadas en el proceso de mantenimiento en sí, resulten efectivas, ya sea que se utilice o no se utilice un enfoque ágil para el desarrollo del sistema. La entrega incremental, el diseño para el cambio y el mantenimiento de la simplicidad tienen sentido cuando se modifica el software. De hecho, se pensaría tanto en un proceso de desarrollo ágil como en un proceso de evolución del software.

Sin embargo, quizá la principal dificultad luego de entregar el software sea mantener al cliente interviniendo en el proceso. Aunque un cliente justifique la participación de tiempo completo de un representante durante el desarrollo del sistema, esto es menos probable en el mantenimiento, cuando los cambios no son continuos. Es posible que los representantes del cliente pierdan interés en el sistema. En consecuencia, es previsible que se requieran mecanismos alternativos, como las propuestas de cambio, descritas en el capítulo 25, para establecer los nuevos requerimientos del sistema.

El otro problema potencial tiene que ver con mantener la continuidad del equipo de desarrollo. Los métodos ágiles se apoyan en aquellos miembros del equipo que comprenden los aspectos del sistema sin que deban consultar la documentación. Si se separa un equipo de desarrollo ágil, entonces se pierde este conocimiento implícito y es difícil que los nuevos miembros del equipo acumulen la misma percepción del sistema y sus componentes.

Quienes apoyan los métodos ágiles han creído fielmente en la promoción de su uso y tienden a pasar por alto sus limitaciones. Esto alienta una respuesta igualmente extrema que, para el autor, exagera los problemas con este enfoque (Stephens y Rosenberg, 2003). Críticos más razonables como DeMarco y Boehm (DeMarco y Boehm, 2002) destacan tanto las ventajas como las desventajas de los métodos ágiles. Proponen un enfoque híbrido donde los métodos ágiles que incorporan algunas técnicas del desarrollo dirigido por un plan son la mejor forma de avanzar.

3.2 Desarrollo dirigido por un plan y desarrollo ágil

Los enfoques ágiles en el desarrollo de software consideran el diseño y la implementación como las actividades centrales en el proceso del software. Incorporan otras actividades en el diseño y la implementación, como la adquisición de requerimientos y pruebas. En contraste, un enfoque basado en un plan para la ingeniería de software identifica etapas separadas en el proceso de software con salidas asociadas a cada etapa. Las salidas de una etapa se usan como base para planear la siguiente actividad del proceso. La figura 3.2 muestra las distinciones entre los enfoques ágil y el basado en un plan para la especificación de sistemas.

En un enfoque basado en un plan, la iteración ocurre dentro de las actividades con documentos formales usados para comunicarse entre etapas del proceso. Por ejemplo, los requerimientos evolucionarán y, a final de cuentas, se producirá una especificación de aquéllos. Esto entonces es una entrada al proceso de diseño y la implementación. En un enfoque ágil, la iteración ocurre a través de las actividades. Por lo tanto, los requerimientos y el diseño se desarrollan en conjunto, no por separado.

Un proceso de software dirigido por un plan soporta el desarrollo y la entrega incrementales. Es perfectamente factible asignar requerimientos y planear tanto la fase de diseño y desarrollo como una serie de incrementos. Un proceso ágil no está inevitable-

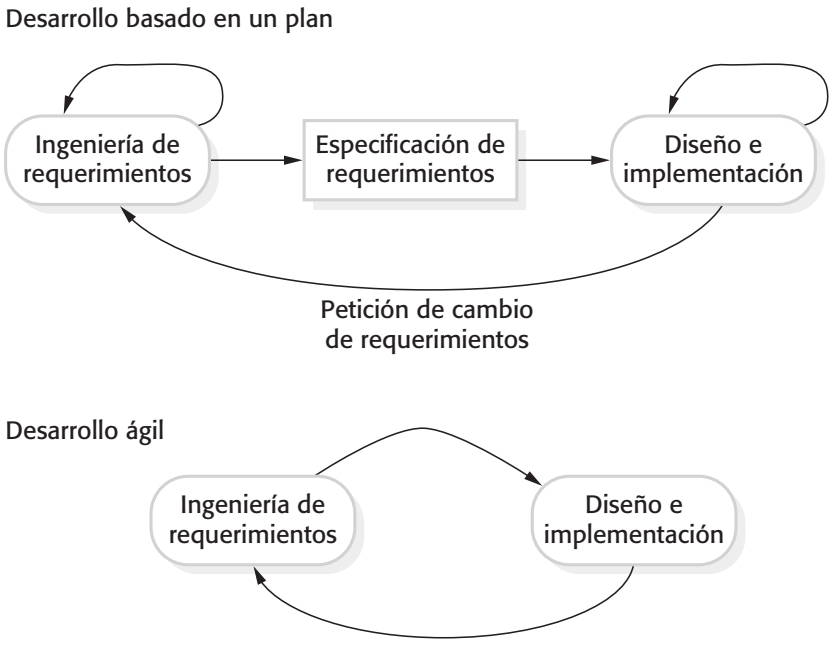


Figura 3.2 Especificación ágil y dirigida por un plan

mente enfocado al código y puede producir cierta documentación de diseño. Como se expone en la siguiente sección, el equipo de desarrollo ágil puede incluir un "pico" de documentación donde, en vez de producir una nueva versión de un sistema, el equipo generará documentación del sistema.

De hecho, la mayoría de los proyectos de software incluyen prácticas de los enfoques ágil y basado en un plan. Para decidir sobre el equilibrio entre un enfoque basado en un plan y uno ágil, se deben responder algunas preguntas técnicas, humanas y organizacionales:

1. ¿Es importante tener una especificación y un diseño muy detallados antes de dirigirse a la implementación? Siendo así, probablemente usted tenga que usar un enfoque basado en un plan.
2. ¿Es práctica una estrategia de entrega incremental, donde se dé el software a los clientes y se obtenga así una rápida retroalimentación de ellos? De ser el caso, considere el uso de métodos ágiles.
3. ¿Qué tan grande es el sistema que se desarrollará? Los métodos ágiles son más efectivos cuando el sistema logra diseñarse con un pequeño equipo asignado que se comunique de manera informal. Esto sería imposible para los grandes sistemas que precisan equipos de desarrollo más amplios, de manera que tal vez se utilice un enfoque basado en un plan.
4. ¿Qué tipo de sistema se desarrollará? Los sistemas que demandan mucho análisis antes de la implementación (por ejemplo, sistema en tiempo real con requerimientos de temporización compleja), por lo general, necesitan un diseño bastante detallado para realizar este análisis. En tales circunstancias, quizá sea mejor un enfoque basado en un plan.
5. ¿Cuál es el tiempo de vida que se espera del sistema? Los sistemas con lapsos de vida prolongados podrían requerir más documentación de diseño, para comunicar al equipo de apoyo los propósitos originales de los desarrolladores del sistema. Sin embargo,

los defensores de los métodos ágiles argumentan acertadamente que con frecuencia la documentación no se conserva actualizada, ni se usa mucho para el mantenimiento del sistema a largo plazo.

6. ¿Qué tecnologías se hallan disponibles para apoyar el desarrollo del sistema? Los métodos ágiles se auxilian a menudo de buenas herramientas para seguir la pista de un diseño en evolución. Si se desarrolla un sistema con un IDE sin contar con buenas herramientas para visualización y análisis de programas, entonces posiblemente se requiera más documentación de diseño.
7. ¿Cómo está organizado el equipo de desarrollo? Si el equipo de desarrollo está distribuido, o si parte del desarrollo se subcontrata, entonces tal vez se requiera elaborar documentos de diseño para comunicarse a través de los equipos de desarrollo. Quizá se necesite planear por adelantado cuáles son.
8. ¿Existen problemas culturales que afecten el desarrollo del sistema? Las organizaciones de ingeniería tradicionales presentan una cultura de desarrollo basada en un plan, pues es una norma en ingeniería. Esto requiere comúnmente una amplia documentación de diseño, en vez del conocimiento informal que se utiliza en los procesos ágiles.
9. ¿Qué tan buenos son los diseñadores y programadores en el equipo de desarrollo? Se argumenta en ocasiones que los métodos ágiles requieren niveles de habilidad superiores a los enfoques basados en un plan, en que los programadores simplemente traducen un diseño detallado en un código. Si usted tiene un equipo con niveles de habilidad relativamente bajos, es probable que necesite del mejor personal para desarrollar el diseño, siendo otros los responsables de la programación.
10. ¿El sistema está sujeto a regulación externa? Si un regulador externo tiene que aprobar el sistema (por ejemplo, la Agencia de Aviación Federal [FAA] estadounidense aprueba el software que es crítico para la operación de una aeronave), entonces, tal vez se le requerirá documentación detallada como parte del sistema de seguridad.

En realidad, es irrelevante el conflicto sobre si un proyecto puede considerarse dirigido por un plan o ágil. A final de cuentas, la principal inquietud de los compradores de un sistema de software es si cuentan o no con un sistema de software ejecutable, que cubra sus necesidades y realice funciones útiles para el usuario de manera individual o dentro de una organización. En la práctica, muchas compañías que afirman haber usado métodos ágiles adoptaron algunas habilidades ágiles y las integraron con sus procesos dirigidos por un plan.

3.3 Programación extrema

La programación extrema (XP) es quizás el método ágil mejor conocido y más ampliamente usado. El nombre lo acuñó Beck (2000) debido a que el enfoque se desarrolló llevando a niveles "extremos" las prácticas reconocidas, como el desarrollo iterativo. Por ejemplo, en la XP muchas versiones actuales de un sistema pueden desarrollarse mediante diferentes programadores, integrarse y ponerse a prueba en un solo día.

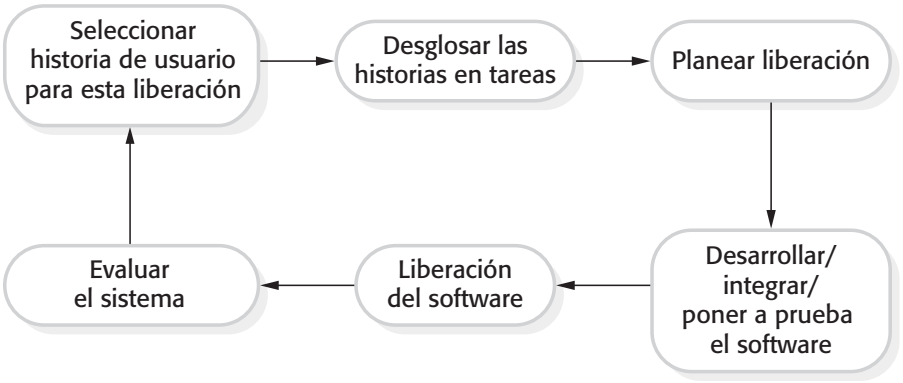


Figura 3.3 El ciclo de liberación de la programación extrema

En la programación extrema, los requerimientos se expresan como escenarios (llamados historias de usuario), que se implementan directamente como una serie de tareas. Los programadores trabajan en pares y antes de escribir el código desarrollan pruebas para cada tarea. Todas las pruebas deben ejecutarse con éxito una vez que el nuevo código se integre en el sistema. Entre las liberaciones del sistema existe un breve lapso. La figura 3.3 ilustra el proceso XP para producir un incremento del sistema por desarrollar.

La programación extrema incluye algunas prácticas, resumidas en la figura 3.4, las cuales reflejan los principios de los métodos ágiles:

1. El desarrollo incremental se apoya en pequeñas y frecuentes liberaciones del sistema. Los requerimientos se fundamentan en simples historias del cliente, o bien, en escenarios usados como base para decidir qué funcionalidad debe incluirse en un incremento del sistema.
2. La inclusión del cliente se apoya a través de un enlace continuo con el cliente en el equipo de desarrollo. El representante del cliente participa en el desarrollo y es responsable de definir las pruebas de aceptación para el sistema.
3. Las personas, no los procesos, se basan en la programación en pares, en la propiedad colectiva del código del sistema y en un proceso de desarrollo sustentable que no incluya jornadas de trabajo excesivamente largas.
4. El cambio se acepta mediante liberaciones regulares del sistema a los clientes, desarrollo de primera prueba, refactorización para evitar degeneración del código e integración continua de nueva funcionalidad.
5. Mantener la simplicidad se logra mediante la refactorización constante, que mejora la calidad del código, y con el uso de diseños simples que no anticipan innecesariamente futuros cambios al sistema.

En un proceso XP, los clientes intervienen estrechamente en la especificación y priorización de los requerimientos del sistema. Estos últimos no se especifican como listas de actividades requeridas del sistema. En cambio, el cliente del sistema forma parte del equipo de desarrollo y discute los escenarios con otros miembros del equipo. En conjunto, desarrollan una "tarjeta de historia" que encapsula las necesidades del cliente. Entonces, el equipo de desarrollo implementa dicho escenario en una liberación futura del software. En la figura 3.5 se muestra el ejemplo de una tarjeta de historia para el

Principio o práctica	Descripción
Planeación incremental	Los requerimientos se registran en tarjetas de historia (*story cards*) y las historias que se van a incluir en una liberación se determinan por el tiempo disponible y la prioridad relativa. Los desarrolladores desglosan dichas historias en "tareas" de desarrollo. Vea las figuras 3.5 y 3.6.
Liberaciones pequeñas	Al principio se desarrolla el conjunto mínimo de funcionalidad útil, que ofrece valor para el negocio. Las liberaciones del sistema son frecuentes y agregan incrementalmente funcionalidad a la primera liberación.
Diseño simple	Se realiza un diseño suficiente para cubrir sólo aquellos requerimientos actuales.
Desarrollo de la primera prueba	Se usa un marco de referencia de prueba de unidad automatizada al escribir las pruebas para una nueva pieza de funcionalidad, antes de que esta última se implemente.
Refactorización	Se espera que todos los desarrolladores refactoricen de manera continua el código y, tan pronto como sea posible, se encuentren mejoras de éste. Lo anterior conserva el código simple y mantenible.
Programación en pares	Los desarrolladores trabajan en pares, y cada uno comprueba el trabajo del otro; además, ofrecen apoyo para que se realice siempre un buen trabajo.
Propiedad colectiva	Los desarrolladores en pares laboran en todas las áreas del sistema, de manera que no se desarrollan islas de experiencia, ya que todos los desarrolladores se responsabilizan por todo el código. Cualquiera puede cambiar cualquier función.
Integración continua	Tan pronto como esté completa una tarea, se integra en todo el sistema. Después de tal integración, deben aprobarse todas las pruebas de unidad en el sistema.
Ritmo sustentable	Grandes cantidades de tiempo extra no se consideran aceptables, pues el efecto neto de este tiempo libre con frecuencia es reducir la calidad del código y la productividad de término medio.
Cliente en sitio	Un representante del usuario final del sistema (el cliente) tiene que disponer de tiempo completo para formar parte del equipo XP. En un proceso de programación extrema, el cliente es miembro del equipo de desarrollo y responsable de llevar los requerimientos del sistema al grupo para su implementación.

Figura 3.4 Prácticas de programación extrema

sistema de administración de pacientes en atención a la salud mental. Ésta es una breve descripción de un escenario para prescribir medicamentos a un paciente.

Las tarjetas de historia son las entradas principales al proceso de planeación XP o el "juego de planeación". Una vez diseñadas las tarjetas de historia, el equipo de desarrollo las descompone en tareas (figura 3.6) y estima el esfuerzo y los recursos requeridos para implementar cada tarea. Esto involucra por lo general discusiones con el cliente para refinar los requerimientos. Entonces, para su implementación, el cliente prioriza las historias y elige aquellas que pueden usarse inmediatamente para entregar apoyo empresarial útil. La intención es identificar funcionalidad útil que pueda implementarse en aproximadamente dos semanas, cuando la siguiente liberación del sistema esté disponible para el cliente.

Desde luego, conforme cambian los requerimientos, las historias no implementadas cambian o se desechan. Si se demandan cambios para un sistema que ya se entregó, se desarrollan nuevas tarjetas de historia y, otra vez, el cliente decide si dichos cambios tienen prioridad sobre la nueva función.

Prescripción de medicamentos

Kate es una médica que quiere prescribir fármacos a un paciente que se atiende en una clínica. El archivo del paciente ya se desplegó en su computadora, de manera que da clic en el campo del medicamento y luego puede seleccionar "medicamento actual", "medicamento nuevo" o "formulario".

Si selecciona "medicamento actual", el sistema le pide comprobar la dosis. Si quiere cambiar la dosis, ingresa la dosis y luego confirma la prescripción.

Si elige "medicamento nuevo", el sistema supone que Kate sabe cuál medicamento prescribir. Ella teclea las primeras letras del nombre del medicamento. El sistema muestra una lista de medicamentos posibles cuyo nombre inicia con dichas letras. Posteriormente elige el fármaco requerido y el sistema responde solicitándole que verifique que el medicamento seleccionado sea el correcto. Ella ingresa la dosis y luego confirma la prescripción.

Si Kate elige "formulario", el sistema muestra un recuadro de búsqueda para el formulario aprobado. Entonces busca el medicamento requerido. Ella selecciona un medicamento y el sistema le pide comprobar que éste sea el correcto. Luego ingresa la dosis y confirma la prescripción.

El sistema siempre verifica que la dosis esté dentro del rango aprobado. Si no es así, le pide a Kate que la modifique.

Después de que ella confirma la prescripción, se desplegará para su verificación. Kate hace clic o en "OK" o en "Cambiar". Si hace clic en "OK", la prescripción se registra en la base de datos de auditoría. Si hace clic en "Cambiar", reingresa al proceso de "prescripción de medicamento".

Figura 3.5 Una historia de la "prescripción de medicamento"

A veces, durante la planeación del juego, salen a la luz preguntas que no pueden responderse fácilmente y se requiere trabajo adicional para explorar posibles soluciones. El equipo puede elaborar algún prototipo o tratar de desarrollarlo para entender el problema y la solución. En términos XP, éste es un "pico" (*spike*), es decir, un incremento donde no se realiza programación. También suele haber "picos" para diseñar la arquitectura del sistema o desarrollar la documentación del sistema.

La programación extrema toma un enfoque "extremo" para el desarrollo incremental. Nuevas versiones del software se construyen varias veces al día y las versiones se entregan a los clientes aproximadamente cada dos semanas. Nunca se descuidan las fechas límite de las liberaciones; si hay problemas de desarrollo, se consulta al cliente y la funcionalidad se elimina de la liberación planeada.

Cuando un programador diseña el sistema para crear una nueva versión, debe correr todas las pruebas automatizadas existentes, así como las pruebas para la nueva funcionalidad. La nueva construcción del software se acepta siempre que todas las pruebas se ejecuten con éxito. Entonces esto se convierte en la base para la siguiente iteración del sistema.

Un precepto fundamental de la ingeniería de software tradicional es que se tiene que diseñar para cambiar. Esto es, deben anticiparse cambios futuros al software y diseñarlo de manera que dichos cambios se implementen con facilidad. Sin embargo, la programación extrema descartó este principio basada en el hecho de que al diseñar para el cambio con frecuencia se desperdicia esfuerzo. No vale la pena gastar tiempo en adicionar generalidad a un programa para enfrentar el cambio. Los cambios anticipados casi nunca se materializan y en realidad pueden hacerse peticiones de cambio diametralmente opuestas. Por lo tanto, el enfoque XP acepta que los cambios sucederán y cuando éstos ocurran realmente se reorganizará el software.

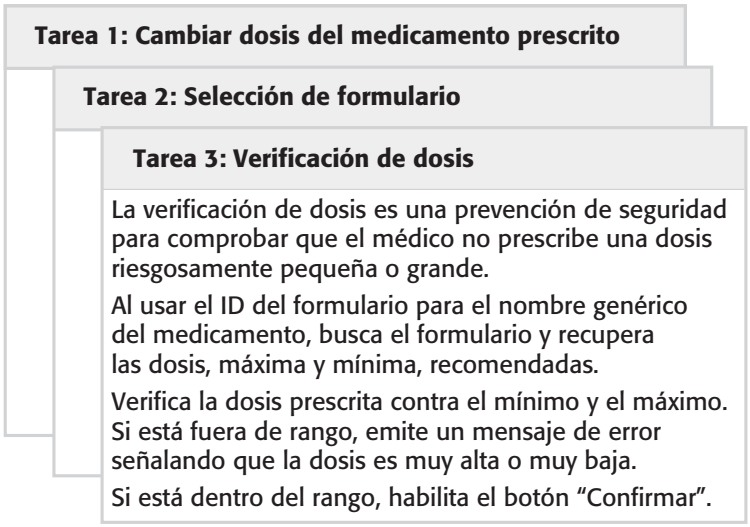


Figura 3.6 Ejemplos de tarjetas de tarea para prescripción de medicamentos.

Un problema general con el desarrollo incremental es que tiende a degradar la estructura del software, de modo que los cambios al software se vuelven cada vez más difíciles de implementar. En esencia, el desarrollo avanza al encontrar soluciones alternativas a los problemas, con el resultado de que el código se duplica con frecuencia, partes del software se reutilizan de forma inadecuada y la estructura global se degrada conforme el código se agrega al sistema.

La programación extrema aborda este problema al sugerir que el software debe refactorizarse continuamente. Esto significa que el equipo de programación busca posibles mejoras al software y las implementa de inmediato. Cuando un miembro del equipo observa un código que puede optimarse, realiza dichas mejoras, aun en situaciones donde no hay necesidad apremiante de ellas. Los ejemplos de refactorización incluyen la reorganización de una jerarquía de clases para remover un código duplicado, el ordenamiento y el cambio de nombre de atributos y métodos, y la sustitución de código con llamadas a métodos definidos en la librería de un programa. Los entornos de desarrollo del programa, como Eclipse (Carlson, 2005), incluyen herramientas para refactorizar, lo cual simplifica el proceso de encontrar dependencias entre secciones de código y realizar modificaciones globales al código.

Entonces, en principio, el software siempre debe ser de fácil comprensión y cambiar a medida que se implementen nuevas historias. En la práctica, no siempre es el caso. En ocasiones la presión del desarrollo significa que la refactorización se demora, porque se dedica el tiempo a la implementación de una nueva funcionalidad. Algunas características y cambios nuevos no pueden ajustarse con facilidad al refactorizar el nivel del código y al requerir modificar la arquitectura del sistema.

En la práctica, muchas compañías que adoptaron XP no usan todas las prácticas de programación extrema que se mencionan en la figura 3.4. Seleccionan según sus formas específicas de trabajar. Por ejemplo, algunas compañías encuentran útil la programación en pares; otras prefieren usar la programación y las revisiones individuales. Para acomodar diferentes niveles de habilidad, algunos programadores no hacen refactorización en partes del sistema que ellos no desarrollan, y pueden usarse requerimientos convencionales en vez de historias de usuario. Sin embargo, la mayoría de las compañías que adoptan una variante XP usan liberaciones pequeñas, desarrollo de primera prueba e integración continua.

3.3.1 Pruebas en XP

Como se indicó en la introducción de este capítulo, una de las diferencias importantes entre desarrollo incremental y desarrollo dirigido por un plan está en la forma en que el sistema se pone a prueba. Con el desarrollo incremental, no hay especificación de sistema que pueda usar un equipo de prueba externo para desarrollar pruebas del sistema. En consecuencia, algunos enfoques del desarrollo incremental tienen un proceso de pruebas muy informal, comparado con las pruebas dirigidas por un plan.

Para evitar varios de los problemas de prueba y validación del sistema, XP enfatiza la importancia de la prueba de programa. La XP incluye un enfoque para probar que reduce las posibilidades de introducir errores no detectados en la versión actual del sistema.

Las características clave de poner a prueba en XP son:

1. Desarrollo de primera prueba,
2. desarrollo de pruebas incrementales a partir de escenarios,
3. involucramiento del usuario en el desarrollo y la validación de pruebas, y
4. el uso de marcos de pruebas automatizadas.

El desarrollo de la primera prueba es una de las innovaciones más importantes en XP. En lugar de escribir algún código y luego las pruebas para dicho código, las pruebas se elaboran antes de escribir el código. Esto significa que la prueba puede correrse conforme se escribe el código y descubrir problemas durante el desarrollo.

Escribir pruebas implícitamente define tanto una interfaz como una especificación del comportamiento para la funcionalidad a desarrollar. Se reducen los problemas de la mala interpretación de los requerimientos y la interfaz. Este enfoque puede adoptarse en cualquier proceso donde haya una relación clara entre un requerimiento de sistema y el código que implementa dicho requerimiento. En la XP, siempre se observa este vínculo porque las tarjetas de historia que representan los requerimientos se descomponen en tareas, y éstas son la principal unidad de implementación. La adopción del desarrollo de primera prueba en XP condujo a enfoques de desarrollo basados en pruebas más generales (Astels, 2003). Éstas se estudian en el capítulo 8.

En el desarrollo de primera prueba, los implementadores de tarea deben comprender ampliamente la especificación, de modo que sean capaces de escribir pruebas para el sistema. Esto significa que las ambigüedades y omisiones en la especificación deben clarificarse antes de comenzar la implementación. Más aún, también evita el problema del "retraso en la prueba". Esto puede ocurrir cuando el desarrollador del sistema trabaja a un ritmo más rápido que el examinador. La implementación está cada vez más adelante de las pruebas y hay una tendencia a omitirlas, de modo que se mantenga la fecha de desarrollo.

Los requerimientos de usuario en XP se expresan como escenarios o historias, y el usuario los prioriza para su desarrollo. El equipo de desarrollo valora cada escenario y lo descompone en tareas. Por ejemplo, en la figura 3.6 se muestran algunas de las tarjetas de tarea desarrolladas a partir de la tarjeta de historia para la prescripción de medicamentos (figura 3.5). Cada tarea genera una o más pruebas de unidad, que verifican la implementación descrita en dicha tarea. La figura 3.7 es una descripción breve de un caso de prueba que se desarrolló para comprobar que la dosis prescrita de un medicamento no se halle fuera de los límites de seguridad conocidos.

Prueba 4: Comprobación de dosis
Entrada: 1. Un número en mg que represente una sola dosis del medicamento. 2. Un número que signifique el número de dosis individuales por día. **Pruebas:** 1. Probar las entradas donde la dosis individual sea correcta, pero la frecuencia muy elevada. 2. Probar las entradas donde la dosis individual sea muy alta y muy baja. 3. Probar las entradas donde la dosis individual × frecuencia sea muy alta y muy baja. 4. Probar las entradas donde la dosis individual × frecuencia esté en el rango permitido. **Salida:** OK o mensaje de error que indique que la dosis está fuera del rango de seguridad.

Figura 3.7
Descripción de caso de prueba para comprobar dosis

El papel del cliente en el proceso de pruebas es ayudar a desarrollar pruebas de aceptación para las historias, que deban implementarse en la siguiente liberación del sistema. Como se estudiará en el capítulo 8, las pruebas de aceptación son el proceso donde el sistema se pone a prueba usando datos del cliente para verificar que se cubren las necesidades reales de éste.

En XP, la prueba de aceptación, como el desarrollo, es incremental. El cliente que forma parte del equipo escribe pruebas conforme avanza el desarrollo. Por lo tanto, todo código nuevo se valida para garantizar que eso sea lo que necesita el cliente. Para la historia en la figura 3.5, la prueba de aceptación implicaría escenarios donde *a*) se cambió la dosis de un medicamento, *b*) se seleccionó un nuevo medicamento y *c*) se usó el formulario para encontrar un medicamento. En la práctica, se requiere por lo general una serie de pruebas de aceptación en vez de una sola prueba.

Contar con el cliente para apoyar el desarrollo de pruebas de aceptación en ocasiones es una gran dificultad en el proceso de pruebas XP. Quienes adoptan el rol del cliente tienen disponibilidad muy limitada, por lo que es probable que no trabajen a tiempo completo con el equipo de desarrollo. El cliente podría creer que brindar los requerimientos fue suficiente contribución y, por lo tanto, se mostrarían renuentes a intervenir en el proceso de pruebas.

La automatización de las pruebas es esencial para el desarrollo de la primera prueba. Las pruebas se escriben como componentes ejecutables antes de implementar la tarea. Dichos componentes de pruebas deben ser independientes, simular el envío de la entrada a probar y verificar que el resultado cumple con la especificación de salida. Un marco de pruebas automatizadas es un sistema que facilita la escritura de pruebas realizables y envía una serie de pruebas para su ejecución. Junit (Massol y Husted, 2003) es un ejemplo usado ampliamente de un marco de pruebas automatizadas.

Conforme se automatizan las pruebas, siempre hay una serie de pruebas que se ejecutan rápida y fácilmente. Cada vez que se agregue cualquier funcionalidad al sistema, pueden correrse las pruebas y conocerse de inmediato los problemas que introduce el nuevo código.

El desarrollo de la primera prueba y las pruebas automatizadas por lo general dan por resultado un gran número de pruebas que se escriben y ejecutan. Sin embargo, este enfoque no conduce necesariamente a pruebas minuciosas del programa. Existen tres razones para ello:

1. Los programadores prefieren programar que probar y, en ocasiones, toman atajos cuando escriben pruebas. Por ejemplo, escriben pruebas incompletas que no comprueban todas las posibles excepciones que quizás ocurran.

2. Algunas pruebas llegan a ser muy difíciles de escribir de manera incremental. Por ejemplo, en una interfaz de usuario compleja, suele ser complicado escribir pruebas de unidad para el código que implementa la "lógica de despliegue" y el flujo de trabajo entre pantallas.

3. Es difícil juzgar la totalidad de un conjunto de pruebas. Aunque tenga muchas pruebas de sistema, su conjunto de pruebas no ofrece cobertura completa. Partes críticas del sistema pueden no ejecutarse y, por ende, permanecerían sin probarse.

En consecuencia, aunque un gran conjunto de pruebas ejecutadas regularmente da la impresión de que el sistema está completo y es correcto, esto tal vez no sea el caso. Si las pruebas no se revisan y se escriben más pruebas después del desarrollo, entonces pueden entregarse *bugs* (problemas, errores en el programa) en la liberación del sistema.

3.3.2 Programación en pares

Otra práctica innovadora que se introdujo en XP es que los programadores trabajan en pares para desarrollar el software. En realidad, trabajan juntos en la misma estación de trabajo para desarrollar el software. Sin embargo, los mismos pares no siempre programan juntos. En vez de ello, los pares se crean dinámicamente, de manera que todos los miembros del equipo trabajen entre sí durante el proceso de desarrollo.

El uso de la programación en pares tiene algunas ventajas:

1. Apoya la idea de la propiedad y responsabilidad colectivas para el sistema. Esto refleja la idea de Weinberg (1971) sobre la programación sin ego, donde el software es propiedad del equipo como un todo y los individuos no son responsables por los problemas con el código. En cambio, el equipo tiene responsabilidad colectiva para resolver dichos problemas.

2. Actúa como un proceso de revisión informal, porque al menos dos personas observan cada línea de código. Las inspecciones y revisiones de código (que se explican en el capítulo 24) son muy eficientes para detectar un alto porcentaje de errores de software. Sin embargo, consumen tiempo en su organización y, usualmente, presentan demoras en el proceso de desarrollo. Aunque la programación en pares es un proceso menos formal que quizá no identifica tantos errores como las inspecciones de código, es un proceso de inspección mucho más económico que las inspecciones formales del programa.

3. Ayuda a la refactorización, que es un proceso de mejoramiento del software. La dificultad de implementarlo en un entorno de desarrollo normal es que el esfuerzo en la refactorización se utiliza para beneficio a largo plazo. Un individuo que practica la refactorización podría calificarse como menos eficiente que uno que simplemente realiza desarrollo del código. Donde se usan la programación en pares y la propiedad colectiva, otros se benefician inmediatamente de la refactorización, de modo que es probable que apoyen el proceso.

Al respecto, tal vez se pensaría que la programación en pares es menos eficiente que la programación individual. En un tiempo dado, un par de desarrolladores elaboraría

la mitad del código que dos individuos que trabajen solos. Hay varios estudios de la productividad de los programadores en pares con resultados mixtos. Al usar estudiantes voluntarios, Williams y sus colaboradores (Cockburn y Williams, 2001; Williams *et al*., 2000) descubrieron que la productividad con la programación en pares es comparable con la de dos individuos que trabajan de manera independiente. Las razones sugeridas son que los pares discuten el software antes de desarrollarlo, de modo que probablemente tengan menos salidas en falso y menos rediseño. Más aún, el número de errores que se evitan por la inspección informal es tal que se emplea menos tiempo en reparar los *bugs* descubiertos durante el proceso de pruebas.

Sin embargo, los estudios con programadores más experimentados (Arisholm *et al*., 2007; Parrish *et al*., 2004) no replican dichos resultados. Hallaron que había una pérdida de productividad significativa comparada con dos programadores que trabajan individualmente. Hubo algunos beneficios de calidad, pero no compensaron por completo los costos de la programación en pares. No obstante, el intercambio de conocimiento que ocurre durante la programación en pares es muy importante, pues reduce los riesgos globales de un proyecto cuando salen miembros del equipo. En sí mismo, esto hace que la programación de este tipo valga la pena.

3.4 Administración de un proyecto ágil

La responsabilidad principal de los administradores del proyecto de software es dirigir el proyecto, de modo que el software se entregue a tiempo y con el presupuesto planeado para ello. Supervisan el trabajo de los ingenieros de software y monitorizan el avance en el desarrollo del software.

El enfoque estándar de la administración de proyectos es el basado en un plan. Como se estudia en el capítulo 23, los administradores se apoyan en un plan para el proyecto que muestra lo que se debe entregar y cuándo, así como quién trabajará en el desarrollo de los entregables del proyecto. Un enfoque basado en un plan requiere en realidad que un administrador tenga una visión equilibrada de todo lo que debe diseñarse y de los procesos de desarrollo. Sin embargo, no funciona bien con los métodos ágiles, donde los requerimientos se desarrollan incrementalmente, donde el software se entrega en rápidos incrementos cortos, y donde los cambios a los requerimientos y el software son la norma.

Como cualquier otro proceso de diseño de software profesional, el desarrollo ágil tiene que administrarse de tal modo que se busque el mejor uso del tiempo y de los recursos disponibles para el equipo. Esto requiere un enfoque diferente a la administración del proyecto, que se adapte al desarrollo incremental y a las fortalezas particulares de los métodos ágiles.

Aunque el enfoque de Scrum (Schwaber, 2004; Schwaber y Beedle, 2001) es un método ágil general, su enfoque está en la administración iterativa del desarrollo, y no en enfoques técnicos específicos para la ingeniería de software ágil. La figura 3.8 representa un diagrama del proceso de administración de Scrum. Este proceso no prescribe el uso de prácticas de programación, como la programación en pares y el desarrollo de primera prueba. Por lo tanto, puede usarse con enfoques ágiles más técnicos, como XP, para ofrecer al proyecto un marco administrativo.

Existen tres fases con Scrum. La primera es la planeación del bosquejo, donde se establecen los objetivos generales del proyecto y el diseño de la arquitectura de software.

Figura 3.8 El proceso de Scrum

A esto le sigue una serie de ciclos *sprint*, donde cada ciclo desarrolla un incremento del sistema. Finalmente, la fase de cierre del proyecto concluye el proyecto, completa la documentación requerida, como los marcos de ayuda del sistema y los manuales del usuario, y valora las lecciones aprendidas en el proyecto.

La característica innovadora de Scrum es su fase central, a saber, los ciclos *sprint*. Un *sprint* de Scrum es una unidad de planeación en la que se valora el trabajo que se va a realizar, se seleccionan las particularidades por desarrollar y se implementa el software. Al final de un *sprint*, la funcionalidad completa se entrega a los participantes. Las características clave de este proceso son las siguientes:

1. Los *sprints* tienen longitud fija, por lo general de dos a cuatro semanas. Corresponden al desarrollo de una liberación del sistema en XP.
2. El punto de partida para la planeación es la cartera del producto, que es la lista de trabajo por realizar en el proyecto. Durante la fase de valoración del *sprint*, esto se revisa, y se asignan prioridades y riesgos. El cliente interviene estrechamente en este proceso y al comienzo de cada *sprint* puede introducir nuevos requerimientos o tareas.
3. La fase de selección incluye a todo el equipo del proyecto que trabaja con el cliente, con la finalidad de seleccionar las características y la funcionalidad a desarrollar durante el *sprint*.
4. Una vez acordado, el equipo se organiza para desarrollar el software. Con el objetivo de revisar el progreso y, si es necesario, volver a asignar prioridades al trabajo, se realizan reuniones diarias breves con todos los miembros del equipo. Durante esta etapa, el equipo se aísla del cliente y la organización, y todas las comunicaciones se canalizan a través del llamado "maestro de Scrum". El papel de este último es proteger al equipo de desarrollo de distracciones externas. La forma en que el trabajo se realiza depende del problema y del equipo. A diferencia de XP, Scrum no hace sugerencias específicas sobre cómo escribir requerimientos, desarrollar la primera prueba, etcétera. Sin embargo, dichas prácticas XP se usan cuando el equipo las considera adecuadas.
5. Al final del *sprint*, el trabajo hecho se revisa y se presenta a los participantes. Luego comienza el siguiente ciclo de *sprint*.

La idea detrás de Scrum es que debe autorizarse a todo el equipo para tomar decisiones, de modo que se evita deliberadamente el término "administrador del proyecto". En

lugar de ello, el "maestro de Scrum" es el facilitador que ordena las reuniones diarias, rastrea el atraso del trabajo a realizar, registra las decisiones, mide el progreso del atraso, y se comunica con los clientes y administradores fuera del equipo.

Todo el equipo asiste a las reuniones diarias, que en ocasiones son reuniones en las que los participantes no se sientan, para hacerlas breves y enfocadas. Durante la reunión, todos los miembros del equipo comparten información, describen sus avances desde la última reunión, los problemas que han surgido y los planes del día siguiente. Ello significa que todos en el equipo conocen lo que acontece y, si surgen problemas, replantean el trabajo en el corto plazo para enfrentarlo. Todos participan en esta planeación; no hay dirección descendente desde el maestro de Scrum.

En la Web existen muchos reportes anecdóticos del uso exitoso del Scrum. Rising y Janoff (2000) discuten su uso exitoso en un entorno de desarrollo de software para telecomunicaciones y mencionan sus ventajas del modo siguiente:

1. El producto se desglosa en un conjunto de piezas manejables y comprensibles.
2. Los requerimientos inestables no retrasan el progreso.
3. Todo el equipo tiene conocimiento de todo y, en consecuencia, se mejora la comunicación entre el equipo.
4. Los clientes observan la entrega a tiempo de los incrementos y obtienen retroalimentación sobre cómo funciona el producto.
5. Se establece la confianza entre clientes y desarrolladores, a la vez que se crea una cultura positiva donde todos esperan el triunfo del proyecto.

Scrum, como originalmente se designó, tenía la intención de usarse con equipos coasignados, donde todos los miembros del equipo pudieran congregarse a diario en reuniones breves. Sin embargo, mucho del desarrollo del software implica ahora equipos distribuidos con miembros del equipo ubicados en diferentes lugares alrededor del mundo. En consecuencia, hay varios experimentos en marcha con la finalidad de desarrollar el Scrum para entornos de desarrollo distribuidos (Smits y Pshigoda, 2007; Sutherland *et al*., 2007).

3.5 Escalamiento de métodos ágiles

Los métodos ágiles se desarrollaron para usarse en pequeños equipos de programación, que podían trabajar juntos en la misma habitación y comunicarse de manera informal. Por lo tanto, los métodos ágiles se emplean principalmente para el diseño de sistemas pequeños y medianos. Desde luego, la necesidad de entrega más rápida del software, que es más adecuada para las necesidades del cliente, se aplica también a sistemas más grandes. Por consiguiente, hay un enorme interés en escalar los métodos ágiles para enfrentar los sistemas de mayor dimensión, desarrollados por grandes organizaciones.

Denning y sus colaboradores (2008) argumentan que la única forma de evitar los problemas comunes de la ingeniería de software, como los sistemas que no cubren las necesidades del cliente y exceden el presupuesto, es encontrar maneras de hacer que los métodos ágiles funcionen para grandes sistemas. Leffingwell (2007) discute cuáles prácticas ágiles se escalan al desarrollo de grandes sistemas. Moore y Spens (2008) reportan su experiencia al usar un enfoque ágil para desarrollar un gran sistema médico, con 300 desarrolladores que trabajaban en equipos distribuidos geográficamente.

El desarrollo de grandes sistemas de software difiere en algunas formas del desarrollo de sistemas pequeños:

1. Los grandes sistemas son, por lo general, colecciones de sistemas separados en comunicación, donde equipos separados desarrollan cada sistema. Dichos equipos trabajan con frecuencia en diferentes lugares, en ocasiones en otras zonas horarias. Es prácticamente imposible que cada equipo tenga una visión de todo el sistema. En consecuencia, sus prioridades son generalmente completar la parte del sistema sin considerar asuntos de los sistemas más amplios.
2. Los grandes sistemas son "sistemas abandonados" (Hopkins y Jenkins, 2008); esto es, incluyen e interactúan con algunos sistemas existentes. Muchos de los requerimientos del sistema se interesan por su interacción y, por lo tanto, en realidad no se prestan a la flexibilidad y al desarrollo incremental. Aquí también podrían ser relevantes los conflictos políticos y a menudo la solución más sencilla a un problema es cambiar un sistema existente. Sin embargo, esto requiere negociar con los administradores de dicho sistema para convencerlos de que los cambios pueden implementarse sin riesgo para la operación del sistema.
3. Donde muchos sistemas se integran para crear un solo sistema, una fracción significativa del desarrollo se ocupa en la configuración del sistema, y no en el desarrollo del código original. Esto no necesariamente es compatible con el desarrollo incremental y la integración frecuente del sistema.
4. Los grandes sistemas y sus procesos de desarrollo por lo común están restringidos por reglas y regulaciones externas, que limitan la forma en que pueden desarrollarse, lo cual requiere de ciertos tipos de documentación del sistema que se va a producir, etcétera.
5. Los grandes sistemas tienen un tiempo prolongado de adquisición y desarrollo. Es difícil mantener equipos coherentes que conozcan el sistema durante dicho periodo, pues resulta inevitable que algunas personas se cambien a otros empleos y proyectos.
6. Los grandes sistemas tienen por lo general un conjunto variado de participantes. Por ejemplo, cuando enfermeras y administradores son los usuarios finales de un sistema médico, el personal médico ejecutivo, los administradores del hospital, etcétera, también son participantes en el sistema. En realidad es imposible involucrar a todos estos participantes en el proceso de desarrollo.

Existen dos perspectivas en el escalamiento de los métodos ágiles:

1. Una perspectiva de "expansión" (*scaling up*), que se interesa por el uso de dichos métodos para el desarrollo de grandes sistemas de software que no logran desarrollarse con equipos pequeños.

2. Una perspectiva de "ampliación" (*scaling out*), que se interesa por que los métodos ágiles se introduzcan en una organización grande con muchos años de experiencia en el desarrollo de software.

Los métodos ágiles tienen que adaptarse para enfrentar la ingeniería de los sistemas grandes. Leffingwell (2007) explica que es esencial mantener los fundamentos de los métodos ágiles: planeación flexible, liberación frecuente del sistema, integración continua, desarrollo dirigido por pruebas y buena comunicación del equipo. El autor considera que las siguientes adaptaciones son críticas y deben introducirse:

1. Para el desarrollo de grandes sistemas no es posible enfocarse sólo en el código del sistema. Es necesario hacer más diseño frontal y documentación del sistema. Debe diseñarse la arquitectura de software y producirse documentación para describir los aspectos críticos del sistema, como esquemas de bases de datos, división del trabajo entre los equipos, etcétera.
2. Tienen que diseñarse y usarse mecanismos de comunicación entre equipos. Esto debe incluir llamadas telefónicas regulares, videoconferencias entre los miembros del equipo y frecuentes reuniones electrónicas breves, para que los equipos se actualicen mutuamente del avance. Hay que ofrecer varios canales de comunicación (como correo electrónico, mensajería instantánea, wikis y sistemas de redes sociales) para facilitar las comunicaciones.
3. La integración continua, donde todo el sistema se construya cada vez que un desarrollador verifica un cambio, es prácticamente imposible cuando muchos programas separados deben integrarse para crear el sistema. Sin embargo, resulta esencial mantener construcciones del sistema frecuentes y liberaciones del sistema regulares. Esto podría significar la introducción de nuevas herramientas de gestión de configuración que soporten el desarrollo de software por parte de múltiples equipos.

Las compañías de software pequeñas que desarrollan productos de software están entre quienes adoptan con más entusiasmo los métodos ágiles. Dichas compañías no están restringidas por burocracias organizacionales o estándares de procesos, y son capaces de cambiar rápidamente para acoger nuevas ideas. Desde luego, las compañías más grandes también experimentan en proyectos específicos con los métodos ágiles; sin embargo, para ellas es mucho más difícil "ampliar" dichos métodos en toda la organización. Lindvall y sus colaboradores (2004) analizan algunos de los problemas al escalar los métodos ágiles en cuatro grandes compañías tecnológicas.

Es difícil introducir los métodos ágiles en las grandes compañías por algunas razones:

1. Los gerentes del proyecto carecen de experiencia con los métodos ágiles; pueden ser reticentes para aceptar el riesgo de un nuevo enfoque, pues no saben cómo afectará sus proyectos particulares.
2. Las grandes organizaciones tienen a menudo procedimientos y estándares de calidad que se espera sigan todos los proyectos y, dada su naturaleza burocrática, es probable que sean incompatibles con los métodos ágiles. En ocasiones, reciben apoyo de herramientas de software (por ejemplo, herramientas de gestión de requerimientos), y el uso de dichas herramientas es obligatorio para todos los proyectos.

3. Los métodos ágiles parecen funcionar mejor cuando los miembros del equipo tienen un nivel de habilidad relativamente elevado. Sin embargo, dentro de grandes organizaciones, probablemente haya una amplia gama de habilidades y destrezas, y los individuos con niveles de habilidad inferiores quizá no sean miembros de equipos efectivos en los procesos ágiles.
4. Quizás haya resistencia cultural contra los métodos ágiles, en especial en aquellas organizaciones con una larga historia de uso de procesos convencionales de ingeniería de sistemas.

Los procedimientos de gestión de cambio y de pruebas son ejemplos de procedimientos de la compañía que podrían no ser compatibles con los métodos ágiles. La administración del cambio es el proceso que controla los cambios a un sistema, de modo que el efecto de los cambios sea predecible y se controlen los costos. Antes de realizarse, todos los cambios deben aprobarse y esto entra en conflicto con la noción de refactorización. En XP, cualquier desarrollador puede mejorar cualquier código sin conseguir aprobación externa. Para sistemas grandes, también existen estándares de pruebas, donde una construcción del sistema se envía a un equipo de pruebas externo. Esto entraría en conflicto con los enfoques de primera prueba y prueba frecuente utilizados en XP.

Introducir y sostener el uso de los métodos ágiles a lo largo de una organización grande es un proceso de cambio cultural. El cambio cultural tarda mucho tiempo en implementarse y a menudo requiere un cambio de administración antes de llevarse a cabo. Las compañías que deseen usar métodos ágiles necesitan promotores para alentar el cambio. Tienen que dedicar recursos significativos para el proceso del cambio. Al momento de escribir este texto, unas cuantas compañías clasificadas como grandes han realizado una transición exitosa al desarrollo ágil a lo largo de la organización.

PUNTOS CLAVE

- Los métodos ágiles son métodos de desarrollo incremental que se enfocan en el diseño rápido, liberaciones frecuentes del software, reducción de gastos en el proceso y producción de código de alta calidad. Hacen que el cliente intervenga directamente en el proceso de desarrollo.
- La decisión acerca de si se usa un enfoque de desarrollo ágil o uno basado en un plan depende del tipo de software que se va a elaborar, las capacidades del equipo de desarrollo y la cultura de la compañía que diseña el sistema.
- La programación extrema es un método ágil bien conocido que integra un rango de buenas prácticas de programación, como las liberaciones frecuentes del software, el mejoramiento continuo del software y la participación del cliente en el equipo de desarrollo.
- Una fortaleza particular de la programación extrema, antes de crear una característica del programa, es el desarrollo de pruebas automatizadas. Todas las pruebas deben ejecutarse con éxito cuando un incremento se integra en un sistema.

- El método de Scrum es un método ágil que ofrece un marco de referencia para la administración del proyecto. Se centra alrededor de un conjunto de *sprints*, que son periodos fijos cuando se desarrolla un incremento de sistema. La planeación se basa en priorizar un atraso de trabajo y seleccionar las tareas de importancia más alta para un *sprint*.
- Resulta difícil el escalamiento de los métodos ágiles para sistemas grandes, ya que éstos necesitan diseño frontal y cierta documentación. La integración continua es prácticamente imposible cuando existen muchos equipos de desarrollo separados que trabajan en un proyecto.

LECTURAS SUGERIDAS

Extreme Programming Explained. Éste fue el primer libro sobre XP y todavía es, quizá, el más legible. Explica el enfoque desde la perspectiva de uno de sus inventores y el entusiasmo se evidencia claramente en el libro. (Kent Beck, Addison-Wesley, 2000.)

"Get Ready for Agile Methods, With Care". Una crítica detallada de los métodos ágiles, que examina sus fortalezas y debilidades; está escrito por un ingeniero de software con vasta experiencia. (B. Boehm, *IEEE Computer*, enero de 2002.) http://doi.ieeecomputersociety.org/10.1109/2.976920.

Scaling Software Agility: Best Practices for Large Enterprises. Aunque se enfoca en los conflictos del escalamiento de los métodos ágiles, este libro también incluye un resumen de los principales métodos ágiles, como XP, Scrum y Crystal. (D. Leffingwell, Addison-Wesley, 2007.)

Running an Agile Software Development Project. La mayoría de los libros acerca de los métodos ágiles se enfocan en un método específico, pero este texto toma un enfoque diferente y analiza cómo poner en práctica XP en un proyecto. Buen consejo práctico. (M. Holcombe, John Wiley and Sons, 2008.)

EJERCICIOS

3.1. Explique por qué la entrega e implementación rápidas de nuevos sistemas es con frecuencia más importante para las empresas que la funcionalidad detallada de dichos sistemas.

3.2. Señale cómo los principios subyacentes a los métodos ágiles conducen al acelerado desarrollo e implementación del software.

3.3. ¿Cuándo desaconsejaría el uso de un método ágil para desarrollar un sistema de software?

3.4. La programación extrema expresa los requerimientos del usuario como historias, y cada historia se escribe en una tarjeta. Analice las ventajas y desventajas de este enfoque para la descripción de requerimientos.

3.5. Explique por qué el desarrollo de la primera prueba ayuda al programador a diseñar una mejor comprensión de los requerimientos del sistema. ¿Cuáles son las dificultades potenciales con el desarrollo de la primera prueba?

3.6. Sugiera cuatro razones por las que la tasa de productividad de los programadores que trabajan en pares llega a ser más de la mitad que la de dos programadores que trabajan individualmente.

3.7. Compare y contraste el enfoque de Scrum para la administración de proyectos con enfoques convencionales basados en un plan, estudiados en el capítulo 23. Las comparaciones deben basarse en la efectividad de cada enfoque para planear la asignación de personal a los proyectos, estimar el costo de los mismos, mantener la cohesión del equipo y administrar los cambios en la conformación del equipo del proyecto.

3.8. Usted es el administrador de software en una compañía que desarrolla software de control crítico para una aeronave. Es el responsable de la elaboración de un sistema de apoyo al diseño de software, que ayude a la traducción de los requerimientos de software a una especificación formal del software (que se estudia en el capítulo 13). Comente acerca de las ventajas y las desventajas de las siguientes estrategias de desarrollo:

a) Recopile los requerimientos para tal sistema con los ingenieros de software y los participantes externos (como la autoridad de certificación reguladora), y desarrolle el sistema usando un enfoque basado en un plan.

b) Diseñe un prototipo usando un lenguaje de script, como Ruby o Python, evalúe este prototipo con los ingenieros de software y otros participantes; luego, revise los requerimientos del sistema. Vuelva a desarrollar el sistema final con Java.

c) Desarrolle el sistema en Java usando un enfoque ágil, con un usuario involucrado en el equipo de diseño.

3.9. Se ha sugerido que uno de los problemas de tener un usuario estrechamente involucrado con un equipo de desarrollo de software es que "se vuelve nativo"; esto es, adopta el punto de vista del equipo de desarrollo y pierde la visión de las necesidades de sus colegas usuarios. Sugiera tres formas en que se podría evitar este problema y discuta las ventajas y desventajas de cada enfoque.

3.10. Con la finalidad de reducir costos y el impacto ambiental del cambio, su compañía decide cerrar algunas oficinas y ofrecer apoyo al personal para trabajar desde casa. Sin embargo, el gerente que introdujo la política no está consciente de que el software se desarrolla usando métodos ágiles, que se apoya en el trabajo cercano del equipo y de la programación en pares. Analice las dificultades que causaría esta nueva política y cómo podría solventar estos problemas.

REFERENCIAS

Ambler, S. W. y Jeffries, R. (2002). *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. New York: John Wiley & Sons.

Arisholm, E., Gallis, H., Dyba, T. y Sjoberg, D. I. K. (2007). "Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise". *IEEE Trans. on Software Eng.,* **33** (2), 65–86.

Astels, D. (2003). *Test Driven Development: A Practical Guide.* Upper Saddle River, NJ: Prentice Hall.

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

Beck, K. (2000). *Extreme Programming explained*. Reading, Mass.: Addison-Wesley.

Carlson, D. (2005). *Eclipse Distilled*. Boston: Addison-Wesley.

Cockburn, A. (2001). *Agile Software Development*. Reading, Mass.: Addison-Wesley.

Cockburn, A. (2004). *Crystal Clear: A Human-Powered Methodology for Small Teams*. Boston: Addison-Wesley.

Cockburn, A. y Williams, L. (2001). "The costs and benefits of pair programming". *In Extreme programming examined. (ed.)*. Boston: Addison-Wesley.

Cohn, M. (2009). *Succeeding with Agile: Software Development Using Scrum*. Boston: Addison-Wesley.

Demarco, T. y Boehm, B. (2002). "The Agile Methods Fray". *IEEE Computer,* **35** (6), 90–2.

Denning, P. J., Gunderson, C. y Hayes-Roth, R. (2008). "Evolutionary System Development". *Comm. ACM,* **51** (12), 29–31.

Drobna, J., Noftz, D. y Raghu, R. (2004). "Piloting XP on Four Mission-Critical Projects". *IEEE Software,* **21** (6), 70–5.

Highsmith, J. A. (2000). *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. New York: Dorset House.

Hopkins, R. y Jenkins, K. (2008). *Eating the IT Elephant: Moving from Greenfield Development to Brownfield. Boston,* Mass.: IBM Press.

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

Leffingwell, D. (2007). *Scaling Software Agility: Best Practices for Large Enterprises*. Boston: Addison-Wesley.

Lindvall, M., Muthig, D., Dagnino, A., Wallin, C., Stupperich, M., Kiefer, D., May, J. y Kahkonen, T. (2004). "Agile Software Development in Large Organizations". *IEEE Computer,* **37** (12), 26–34.

Martin, J. (1981). *Application Development Without Programmers*. Englewood Cliffs, NJ: Prentice-Hall.

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

Mills, H. D., O'Neill, D., Linger, R. C., Dyer, M. y Quinnan, R. E. (1980). "The Management of Software Engineering". *IBM Systems*. J., **19** (4), 414–77.

Moore, E. y Spens, J. (2008). "Scaling Agile: Finding your Agile Tribe". *Proc. Agile 2008 Conference,* Toronto: IEEE Computer Society. 121–124.

Palmer, S. R. y Felsing, J. M. (2002). *A Practical Guide to Feature-Driven Development*. Englewood Cliffs, NJ: Prentice Hall.

Parrish, A., Smith, R., Hale, D. y Hale, J. (2004). "A Field Study of Developer Pairs: Productivity Impacts and Implications". *IEEE Software,* **21** (5), 76–9.

Poole, C. y Huisman, J. W. (2001). "Using Extreme Programming in a Maintenance Environment". *IEEE Software,* **18** (6), 42–50.

Rising, L. y Janoff, N. S. (2000). "The Scrum Software Development Process for Small Teams". *IEEE Software,* **17** (4), 26–32.

Schwaber, K. (2004). *Agile Project Management with Scrum*. Seattle: Microsoft Press.

Schwaber, K. y Beedle, M. (2001). *Agile Software Development with Scrum*. Englewood Cliffs, NJ: Prentice Hall.

Smits, H. y Pshigoda, G. (2007). "Implementing Scrum in a Distributed Software Development Organization". Agile 2007, Washington, DC: IEEE Computer Society.

Stapleton, J. (1997). *DSDM Dynamic Systems Development Method*. Harlow, UK: Addison-Wesley.

Stapleton, J. (2003). *DSDM: Business Focused Development, 2nd ed*. Harlow, UK: Pearson Education.

Stephens, M. y Rosenberg, D. (2003). *Extreme Programming Refactored*. Berkley, Calif.: Apress.

Sutherland, J., Viktorov, A., Blount, J. y Puntikov, N. (2007). "Distributed Scrum: Agile Project Management with Outsourced Development Teams". 40th Hawaii Int. Conf. on System Sciences, Hawaii: IEEE Computer Society.

Weinberg, G. (1971). *The Psychology of Computer Programming*. New York: Van Nostrand.

Williams, L., Kessler, R. R., Cunningham, W. y Jeffries, R. (2000). "Strengthening the Case for Pair Programming". *IEEE Software,* **17** (4), 19–25.