17

Desarrollo rápido de software

Objetivos

El objetivo de este capítulo es describir varios enfoques para el desarrollo de software pensados para la entrega rápida del software. Cuando haya leído este capítulo:

- entenderá cómo un enfoque de desarrollo de software iterativo e incremental conduce a una entrega más rápida de un software más útil;
- entenderá las diferencias entre los métodos de desarrollo ágiles y los métodos de desarrollo de software que dependen de la documentación de las especificaciones y diseños;
- conocerá los principios, prácticas y algunas de las limitaciones de la programación extrema;
- entenderá cómo se puede utilizar el prototipado para ayudar a resolver requerimientos y diseñar incertidumbres cuando se tiene que utilizar un enfoque de desarrollo basado en la especificación.

Contenidos

Actualmente, los negocios operan en un entorno global que cambia rápidamente. Tienen que responder a nuevas oportunidades y mercados, condiciones económicas cambiantes y la aparición de productos y servicios competidores. El software es parte de casi todas las operaciones de negocio, por lo que es fundamental que el software nuevo se desarrolle rápidamente para aprovechar nuevas oportunidades y responder a la presión competitiva. Por lo tanto, actualmente el desarrollo y entrega rápidos son a menudo los requerimientos más críticos de los sistemas software. De hecho, muchas compañías están dispuestas a una pérdida en la calidad del software y en el compromiso sobre los requerimientos en favor de una entrega rápida del software.

Debido a que estas compañías operan en un entorno cambiante, a menudo es prácticamente imposible obtener un conjunto completo de requerimientos del software estables. Los requerimientos que se proponen cambian inevitablemente, porque a los clientes les resulta imposible predecir cómo afectará un sistema a la manera de trabajar, cómo interactuará con otros sistemas y qué operaciones de los usuarios se deben automatizar. Es posible que los requerimientos reales sólo queden claros cuando se haya entregado el sistema y los usuarios hayan adquirido experiencia.

Los procesos de desarrollo del software basados en una completa especificación de los requerimientos y posterior diseño, construcción y pruebas del sistema no se ajustan al desarrollo rápido de aplicaciones. Cuando los requerimientos cambian o cuando se descubren problemas con ellos, el diseño o implementación del sistema se tiene que volver a realizar o probar. Como consecuencia, normalmente se prolonga en el tiempo un proceso en cascada convencional o basado en la especificación y el software definitivo se entrega al cliente mucho tiempo después de que fuera inicialmente especificado.

En un entorno de negocios que se mueve con rapidez, esto puede causar verdaderos problemas. Para cuando esté disponible el software, la razón original de su adquisición puede haber cambiado tan radicalmente que el software sea en realidad inútil. Por lo tanto, en particular para los sistemas de negocio, los procesos de desarrollo que se basan en el desarrollo y entrega rápidos de software son esenciales.

Los procesos de desarrollo rápido de software están diseñados para producir software útil de forma rápida. Generalmente, son procesos iterativos en los que se entrelazan la especificación, el diseño, el desarrollo y las pruebas. El software no se desarrolla y utiliza en su totalidad, sino en una serie de incrementos, donde en cada incremento se incluyen nuevas funcionalidades al sistema. Aunque existen muchos enfoques para el desarrollo rápido de software, comparten las mismas características fundamentales:

1. Los procesos de especificación, diseño e implementación son concurrentes. No existe una especificación del sistema detallada, y la documentación del diseño se minimiza o es generada automáticamente por el entorno de programación utilizado para implementar el sistema. El documento de requerimientos del usuario define solamente las características más importantes del sistema.
2. El sistema se desarrolla en una serie de incrementos. Los usuarios finales y otros stakeholders del sistema participan en la especificación y evaluación de cada incremento. Pueden proponer cambios en el software y nuevos requerimientos que se deben implementar en un incremento posterior del sistema.
3. A menudo se desarrollan las interfaces de usuario del sistema utilizando un sistema de desarrollo interactivo que permite que el diseño de la interfaz se cree rápidamente dibujando y colando iconos en la interfaz. El sistema puede generar una interfaz basada en web para un navegador o una interfaz para una plataforma específica como Microsoft Windows.

El desarrollo incremental, introducido en el Capítulo 4, implica producir y entregar el software en incrementos más que en un paquete único. Cada iteración del proceso produce un nuevo incremento del software. Las dos ventajas principales de adoptar un enfoque incremental para el desarrollo del software son:

1. ***Entrega acelerada de los servicios de! cliente.*** En los incrementos iniciales del sistema se pueden entregar las funcionalidades de alta prioridad para que los clientes puedan aprovechar el sistema desde el principio de su desarrollo. Los clientes pueden ver sus requerimientos en la práctica y especificar cambios a incorporar en entregas posteriores del sistema.
2. ***Compromiso del cliente con el sistema.*** Los usuarios del sistema tienen que estar implicados en el proceso de desarrollo incremental debido a que tienen que proporcionar retroal¡mentación sobre los incrementos entregados al equipo de desarrollo. Su participación no sólo significa que es más probable que el sistema cumpla sus requerimientos, sino que también los usuarios finales del sistema tienen que hacer un compromiso con él y conseguir que éste llegue a funcionar.

En la Figura 17.1 se ilustra un modelo de proceso general para el desarrollo incremental. Observe que las etapas iniciales de este proceso se centran en el diseño arquitectónico. Si no se considera la arquitectura al principio del proceso, es probable que la estructura general del sistema sea inestable y se degrade conforme se entreguen nuevos incrementos.

El desarrollo incremental del software es un enfoque mucho mejor para el desarrollo de la mayoría de los sistemas de negocio, comercio electrónico y personales porque refleja el modo fundamental al que todos nosotros tendemos al resolver problemas. Rara vez encontramos una solución completa a un problema por adelantado, pero nos movemos hacia una solución en una serie de pasos, dando marcha atrás cuando nos damos cuenta de que hemos cometido un error.

Sin embargo, puede haber verdaderos problemas con este enfoque, particularmente en las grandes compañías con procedimientos bastante rígidos y en organizaciones donde el desarrollo del software normalmente se subcontrata con un contratista exterior. Los principales problemas con el desarrollo iterativo y la entrega incremental son:

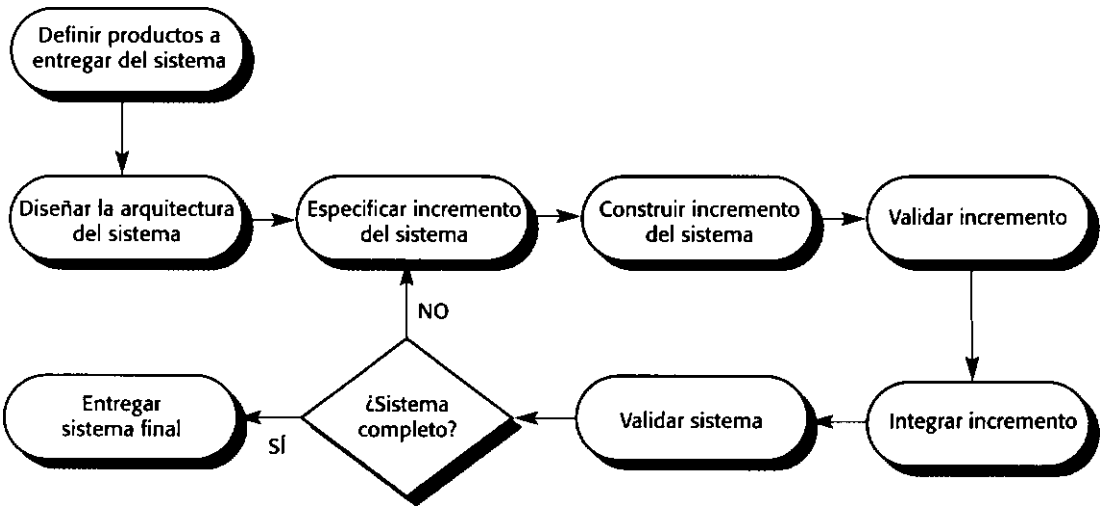

Figura 17.1 Un proceso de desarrollo iterativo.

1. ***Problemas de administración.*** Las estructuras de administración del software para sistemas grandes se diseñan para tratar con modelos de proceso del software que generan entregas periódicas para evaluar el progreso. Los sistemas desarrollados incrementalmente cambian tan rápido que no es rentable producir una gran cantidad de documentación del sistema. Además, el desarrollo incremental muchas veces puede requerir el uso de tecnologías desconocidas para asegurar una entrega más rápida del software. A los administradores puede resultarles difícil utilizar al personal existente en los procesos de desarrollo incremental puesto que carecen de las habilidades requeridas.
2. ***Problemas contractuales.*** El modelo contractual normal entre un cliente y un desarrollador de software se basa en la especificación del sistema. Cuando no existe tal especificación, puede ser difícil diseñar un contrato para el desarrollo del sistema. Los clientes pueden estar descontentos con un contrato que simplemente pague a los desarrolladores por el tiempo invertido en el proyecto, ya que puede conducir a que el sistema se desarrolle lentamente y se sobrepase el presupuesto; los desarrolladores probablemente no aceptarán un contrato con precio fijo debido a que no pueden controlar los cambios requeridos por los usuarios finales.
3. ***Problemas de validación.*** En un proceso basado en la especificación, la verificación y la validación están pensadas para demostrar que el sistema cumple su especificación. Un equipo independiente de V & V puede empezar a trabajar tan pronto como esté disponible la especificación y puede preparar pruebas en paralelo con la implementación del sistema. Los procesos de desarrollo iterativo intentan minimizar la documentación y entrelazan la especificación y el desarrollo. Por lo tanto, la validación independiente de los sistemas desarrollados incrementalmente es difícil.
4. ***Problemas de mantenimiento.*** Los cambios continuos tienden a corromper la estructura del cualquier sistema software. Esto significa que cualquiera, aparte de los desarrolladores originales, puede tener dificultades para entender el software. Una forma de reducir este problema es utilizar refactorización, donde se mejoran continuamente las estructuras del software durante el proceso de desarrollo. Esto se expone en la Sección 17.2, donde se trata la programación extrema. Además, si se utiliza tecnología especializada, como los entornos R A D (descritos en la Sección 17.3), para ayudar al desarrollo rápido del prototipo, la tecnología R A D puede convertirse en obsoleta. Por lo tanto, puede ser difícil encontrar personas que tengan los conocimientos requeridos para dar mantenimiento al sistema.

Por supuesto, existen algunos tipos de sistemas donde el desarrollo y entrega rápidos no son el mejor enfoque. Estos son sistemas muy grandes donde el desarrollo puede implicar equipos que trabajan en diferentes lugares, algunos sistemas embebidos donde el software depende del desarrollo del hardware y algunos sistemas críticos en los que se deben analizar todos los requerimientos para verificar las interacciones que puedan comprometer la seguridad o protección del sistema.

Estos sistemas, por supuesto, sufren los mismos problemas de requerimientos inciertos y cambiantes. Por lo tanto, para abordar estos problemas y conseguir algunos de los beneficios del desarrollo incremental, se puede utilizar un proceso híbrido en el que se desarrolle de forma iterativa un prototipo del sistema y se utilice como una plataforma para experimentar con los requerimientos y diseño del sistema. Con la experiencia adquirida con el prototipo, se puede tener una mayor seguridad de que los requerimientos cumplen las necesidades reales de los stakeholders del sistema.

Se utiliza aquí el término ***prototipado*** para expresar un proceso iterativo para desarrollar un sistema experimental que ***no*** está destinado a la utilización por parte de los clientes. Se desarrolla un prototipo del sistema para ayudar a los desarrolladores de software y a los clientes a comprender qué se debe implementar. Sin embargo, algunas veces se utiliza la expresión ***prototipado evolutivo*** como sinónimo del desarrollo de software incremental. El prototipo no se descarta sino que se desarrolla para cumplir los requerimientos del usuario.

La Figura 17.2 muestra que el desarrollo y el prototipado incremental tienen objetivos diferentes:

1. El objetivo del desarrollo incremental es entregar a los usuarios finales un sistema funcional. Esto significa que normalmente debe comenzar con los requerimientos del usuario que mejor se comprendan y que tengan la prioridad más alta. Los requerimientos inciertos y de prioridad más baja se implementan cuando sean requeridos por los usuarios.
2. El objetivo del prototipado desechable es validar u obtener los requerimientos del sistema. Debe comenzar con aquellos requerimientos que no se comprendan bien ya que requiere saber más de ellos. Puede no necesitar nunca prototipar los requerimientos sencillos.

Otra diferencia importante entre estos enfoques está en la gestión de la calidad de los sistemas. Los prototipos desechables tienen un periodo de vida muy corto. Debe ser posible cambiarlos rápidamente durante el desarrollo, pero no se requiere mantenimiento a largo plazo. En dicho prototipo se puede permitir un rendimiento pobre y una baja fiabilidad siempre y cuando ayude a entender los requerimientos.

Por el contrario, los sistemas desarrollados incrementalmente en que las versiones iniciales evolucionan a un sistema final se deben desarrollar con los mismos estándares de calidad de la organización que cualquier otro software. Deben tener una estructura robusta para que se les pueda dar mantenimiento durante muchos años. Deben ser fiables y eficientes, y deben estar acordes con los estándares organizacionales apropiados.

17.1 Métodos ágiles

En los años 80 y principios de los 90, existía una opinión general de que la mejor forma de obtener un mejor software era a través de una planificación cuidadosa del proyecto, una garantía de calidad formalizada, la utilización de métodos de análisis y diseño soportados por herramientas CASE, y procesos de desarrollo de software controlados y rigurosos. Esta opinión provenía, fundamentalmente, de la comunidad de ingenieros de software implicada en el desarrollo de grandes sistemas software de larga vida que normalmente se componían de un gran número de programas individuales.

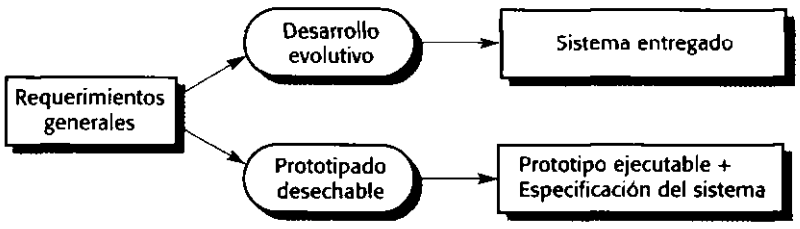

Figura 17.2 Desarrollo y prototipado incremental.

Algunos o la totalidad de estos programas eran a menudo sistemas críticos, como se indicó en el Capítulo 3. Este software era desarrollado por grandes equipos que a veces trabajaban para compañías diferentes. A menudo estaban dispersos geográficamente y trabajaban en el software durante largos periodos de tiempo. Un ejemplo de este tipo de software son los sistemas de control de un avión moderno, en los cuales pueden transcurrir hasta 10 años desde la especificación inicial hasta la utilización. Estos enfoques, algunos de los cuales trato en este libro, implican una importante sobrecarga de trabajo en cuanto a la planificación, diseño y documentación del sistema. Este esfuerzo adicional se justifica cuando se tiene que coordinar el trabajo de múltiples equipos de desarrollo, cuando el sistema es un sistema crítico y cuando muchas personas diferentes estarán involucradas en el mantenimiento del software durante su vida.

Sin embargo, cuando este enfoque «pesado» de desarrollo basado en la planificación fue aplicado a sistemas de negocio pequeños y de tamaño medio, el esfuerzo invertido era tan grande que algunas veces dominaba el proceso de desarrollo del software. Se pasaba más tiempo pensando en cómo se debía desarrollar el sistema que en programar el desarrollo y las pruebas. Cuando cambiaban los requerimientos, se hacía esencial rehacer el trabajo, y al menos en principio, la especificación y el diseño tenían que cambiar con el programa.

El descontento con estos enfoques pesados condujo a varios desarrolladores de software en los años 90 a proponer nuevos métodos ágiles. Éstos permitieron a los equipos de desarrollo centrarse en el software mismo en vez de en su diseño y documentación. Los métodos ágiles umversalmente dependen de un enfoque iterativo para la especificación, desarrollo y entrega del software, y principalmente fueron diseñados para apoyar al desarrollo de aplicaciones de negocio donde los requerimientos del sistema normalmente cambiaban rápidamente durante el proceso de desarrollo. Están pensados para entregar software funcional de forma rápida a los clientes, quienes pueden entonces proponer que se incluyan en iteraciones posteriores del sistema nuevos requerimientos o cambios en los mismos.

Probablemente el método ágil más conocido es la programación extrema (Beck, 1999; Beck, 2000), que se describe posteriormente en este capítulo. Sin embargo, otros enfoques ágiles son Serum (Schwaber y Beedle, 2001), Cristal (Cockburn, 2001), Desarrollo de Software Adaptable (Highsmith, 2000), DSDM (Stapleton, 1997) y Desarrollo Dirigido por Características (Palmer y Felsing, 2002). El éxito de estos métodos ha llevado a una cierta integración con métodos de desarrollo más tradicionales basados en el modelado de sistemas, dando por resultado la noción de modelado ágil (Ambler y Jeffries, 2002) y las instanciaciones ágiles del Proceso Unificado de Rational (Larman, 2002).

Aunque todos estos métodos ágiles se basan en la noción de desarrollo y entrega incremántales, proponen procesos diferentes para alcanzarla. Sin embargo, comparten un conjunto de principios y, por lo tanto, tienen mucho en común. En la Figura 17.3 se muestran estos principios.

Los partidarios de los métodos ágiles han sido evangélicos en la promoción de su uso y han tendido a pasar por alto sus deficiencias. Esto ha provocado una respuesta igualmente radical, la cual, exagera los problemas de este enfoque (Stephens y Rosenberg, 2003). Críticos más razonables como DeMarco y Boehm (DeMarco y Boehm, 2002) resaltan tanto las ventajas como las desventajas de los métodos ágiles. Proponen que un enfoque híbrido en el cual los métodos ágiles incorporen algunas técnicas del desarrollo basado en la planificación puede ser lo mejor a largo plazo.

En la práctica, sin embargo, los principios subyacentes a los métodos ágiles son a veces difíciles de realizar:

Principio	Descripción
Participación del cliente	Los clientes deben estar fuertemente implicados en todo el proceso de desarrollo. Su papel es proporcionar y priorizar nuevos requerimientos del sistema y evaluar las iteraciones del sistema.
Entrega incremental	El software se desarrolla en incrementos, donde el cliente especifica los requerimientos a incluir en cada incremento.
Personas, no procesos	Se deben reconocer y explotar las habilidades del equipo de desarrollo. Se les debe dejar desarrollar sus propias formas de trabajar, sin procesos formales, a los miembros del equipo.
Aceptar el cambio	Se debe contar con que los requerimientos del sistema cambian, por lo que el sistema se diseña para dar cabida a estos cambios.
Mantener la simplicidad	Se deben centrar en la simplicidad tanto en el software a desarrollar como en el proceso de desarrollo. Donde sea posible, se trabaja activamente para eliminar la complejidad del sistema.

Figura 17.3
Los principios de los métodos ágiles.

1. Si bien la idea de la participación del cliente en el proceso de desarrollo es atractiva, su éxito depende de tener un cliente que esté dispuesto y pueda pasar tiempo con el equipo de desarrollo y que pueda representar a todos los stakeholders del sistema. Frecuentemente, los representantes de los clientes están sometidos a otras presiones y no pueden participar plenamente en el desarrollo del software.
2. Los miembros individuales del equipo pueden no tener la personalidad apropiada para la participación intensa que es típica de los métodos ágiles. Por lo tanto, es posible que no se relacionen adecuadamente con los otros miembros del equipo.
3. Priorizar los cambios puede ser extremadamente difícil, especialmente en sistemas en lo que existen muchos stakeholders. Por lo general, cada stakeholders proporciona prioridades distintas a diferentes cambios.
4. Mantener la simplicidad requiere un trabajo extra. Bajo presión por las agendas de entregas, los miembros del equipo pueden no tener tiempo de llevar a cabo las simplificaciones deseables del sistema.

Otro problema no técnico, el cual es un problema general con el desarrollo y entrega increméntales, ocurre cuando los clientes del sistema utilizan una organización extema para el desarrollo del sistema. Como se explicó en el Capítulo 6, normalmente el documento de requerimientos del software es parte del contrato entre el cliente y el proveedor. Puesto que la especificación incremental es inherente a los métodos ágiles, redactar contratos para este tipo de desarrollo puede ser difícil.

Por consiguiente, los métodos ágiles tienen que depender de contratos donde el cliente paga por el tiempo necesario para el desarrollo del sistema en vez de por el desarrollo de un conjunto de requerimientos específico. Siempre y cuando vaya todo bien, esto beneficia tanto al cliente como al desarrollados Sin embargo, si surgen problemas puede haber disputas sobre quién es el culpable y quién debe pagar por el tiempo extra y recursos necesarios para resolver los problemas.

Todos los métodos tienen límites, y los métodos ágiles son sólo apropiados para algunos tipos de desarrollo de sistemas. Son los más idóneos para el desarrollo de sistemas de negocio pequeños y de tamaño medio y para el desarrollo de productos para ordenadores perso-

nales. No son adecuados para el desarrollo de sistemas a gran escala con equipos de desarrollo ubicados en diferentes lugares y donde puedan haber complejas interacciones con otros sistemas hardware o software. No se deben utilizar los métodos ágiles para el desarrollo de sistemas críticos en los que es necesario un análisis detallado de todos los requerimientos del sistema para comprender sus implicaciones de seguridad o protección.

17.2 Programación extrema

La Programación Extrema (XP) es posiblemente el método ágil más conocido y ampliamente utilizado. El nombre fue acuñado por Beck (Beck, 2000) debido a que el enfoque fue desarrollado utilizando buenas prácticas reconocidas, como el desarrollo iterativo, y con la participación del cliente en niveles «extremos».

En la programación extrema, todos los requerimientos se expresan como escenarios (llamados historias de usuario), los cuales se implementan directamente como una serie de tareas. Los programadores trabajan en parejas y desarrollan pruebas para cada tarea antes de escribir el código. Todas las pruebas se deben ejecutar satisfactoriamente cuando el código nuevo se integre al sistema. Existe un pequeño espacio de tiempo entre las entregas del sistema. La Figura 17.4 ilustra el proceso de la XP para producir un incremento del sistema que se está desarrollando.

La programación extrema implica varias prácticas, resumidas en la Figura 17.5, que se ajustan a los principios de los métodos ágiles:

1. El desarrollo incremental se lleva a cabo través de entregas del sistema pequeñas y frecuentes y por medio de un enfoque para la descripción de requerimientos basado en las historias de cliente o escenarios que pueden ser la base para el proceso de planificación.
2. La participación del cliente se lleva a cabo a través del compromiso a tiempo completo del cliente en el equipo de desarrollo. Los representantes de los clientes participan en el desarrollo y son los responsables de definir las pruebas de aceptación del sistema.
3. El interés en las personas, en vez de en los procesos, se lleva a cabo a través de la programación en parejas, la propiedad colectiva del código del sistema, y un proceso de desarrollo sostenible que no implique excesivas jornadas de trabajo.
4. El cambio se lleva a cabo a través de las entregas regulares del sistema, un desarrollo previamente probado y la integración continua.
5. El mantenimiento de la simplicidad se lleva a cabo a través de la refactorización constante para mejorar la calidad del código y la utilización de diseños sencillos que no prevén cambios futuros en el sistema.

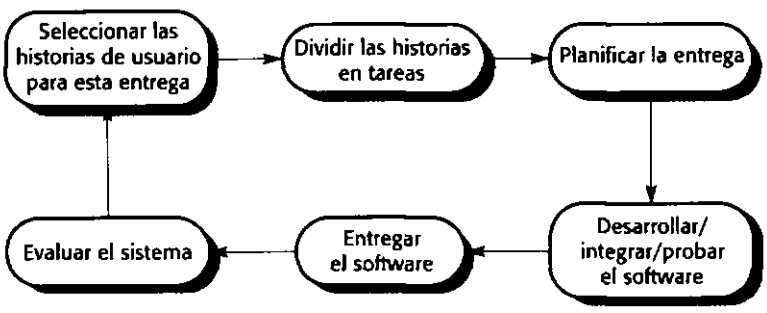

Figura 17.4 El ciclo de entrega en la programación extrema.

Principio o práctica	Descripción
Planificación incremental	Los requerimientos se registran en tarjetas de historias y las historias a incluir en una entrega se determinan según el tiempo disponible y su prioridad relativa. Los desarrolladores dividen estas Historias en «Tareas» de desarrollo. Véanse las Figuras 17.6 y 17.7.
Entregas pequeñas	El mínimo conjunto útil de funcionalidad que proporcione valor de negocio se desarrolla primero. Las entregas del sistema son frecuentes e incrementalmente añaden funcionalidad a la primera entrega.
Diseño sencillo	Sólo se lleva a cabo el diseño necesario para cumplir los requerimientos actuales.
Desarrollo previamente probado	Se utiliza un sistema de pruebas de unidad automatizado para escribir pruebas para nuevas funcionalidades antes de que éstas se implementen.
Refactorización	Se espera que todos los desarrolladores refactoricen el código continuamente tan pronto como encuentren posibles mejoras en el código. Esto conserva el código sencillo y mantenible.
Programación en parejas	Los desarrolladores trabajan en parejas, verificando cada uno el trabajo del otro y proporcionando la ayuda necesaria para hacer siempre un buen trabajo.
Propiedad colectiva	Las parejas de desarrolladores trabajan en todas las áreas del sistema, de modo que no desarrollen islas de conocimientos y todos los desarrolladores posean todo el código. Cualquiera puede cambiar cualquier cosa.
Integración continua	En cuanto acaba el trabajo en una tarea, se integra en el sistema entero. Después de la integración, se deben pasar al sistema todas las pruebas de unidad.
Ritmo sostenible	No se consideran aceptables grandes cantidades de horas extras, ya que a menudo el efecto que tienen es que se reduce la calidad del código y la productividad a medio plazo.
Cliente presente	Debe estar disponible al equipo de la XP un representante de los usuarios finales del sistema (el cliente) a tiempo completo. En un proceso de la programación extrema, el cliente es miembro del equipo de desarrollo y es responsable de formular al equipo los requerimientos del sistema para su implementación.

Figura 17.5 Prácticas de la programación extrema.

En un proceso de la XP, los clientes están fuertemente implicados en la especificación y establecimiento de prioridades de los requerimientos del sistema. Los requerimientos no se especifican como una lista de funciones requeridas del sistema. Más bien, los clientes del sistema son parte del equipo de desarrollo y discuten escenarios con otros miembros del equipo. Desarrollan conjuntamente una «tarjeta de historias» ***(story cara)*** que recoge las necesidades del cliente. El equipo de desarrollo intentará entonces implementar ese escenario en una entrega futura del software. En la Figura 17.6 se ilustra un ejemplo de una tarjeta de historia para el sistema LIBSYS, basada en un escenario del Capítulo 7.

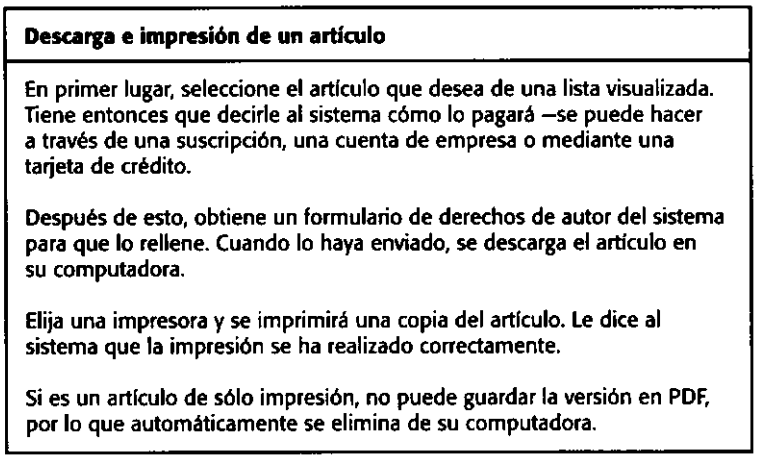

Figura 17.6 Tarjeta de historia para la descarga de documentos.

Una vez que se han desarrollado las tarjetas de historias, el equipo de desarrollo las divide en tareas y estima el esfuerzo y recursos requeridos para su implementación. El cliente establece entonces la prioridad de las historias a implementar, eligiendo aquellas historias que pueden ser utilizadas inmediatamente para entregar un apoyo útil al negocio. Por supuesto, cuando los requerimientos cambian, las historias sin implementar también cambian o se pueden descartar. Si se requieren cambios en un sistema que ya se ha entregado, se desarrollan nuevas tarjetas de historias y, de nuevo, el cliente decide si estos cambios tienen prioridad sobre nuevas funcionalidades.

La programación extrema adopta un enfoque «extremo» para el desarrollo iterativo. Se pueden construir varias veces al día nuevas versiones del software y los incrementos se entregan al cliente cada dos meses aproximadamente. Cuando un programador construye el sistema para crear una versión nueva, debe ejecutar todas las pruebas automatizadas existentes además de las pruebas para las funcionalidades nuevas. El nuevo software generado solamente se acepta si se ejecutan satisfactoriamente todas las pruebas.

Un precepto fundamental de la ingeniería del software tradicional es que se debe diseñar para el cambio. Esto es, hay que prever los cambios futuros en el software y diseñar éste de forma que tales cambios se puedan implementar fácilmente. Sin embargo, la programación extrema ha descartado este principio partiendo del hecho de que diseñar para el cambio es a menudo un esfuerzo inútil. Con frecuencia los cambios previstos nunca se materializan y realmente se efectúan peticiones de cambios completamente diferentes.

El problema con la implementación de cambios imprevistos es que tienden a degradar la estructura del software, por lo que los cambios se hacen cada vez más difíciles de implementar. La programación extrema aborda este problema sugiriendo que se debe refactorizar constantemente el software. Esto significa que el equipo de programación busca posibles mejoras del software y las implementa inmediatamente. Por lo tanto, el software siempre debe ser fácil de entender y cambiar cuando se implementen nuevas historias.

17.2.1 Pruebas en XP

Como se ha indicado en la introducción de este capítulo, una de las diferencias importantes entre el desarrollo iterativo y el desarrollo basado en la planificación es la forma de probar el sistema. Con el desarrollo iterativo, no existe una especificación del sistema que pueda ser utilizada por un equipo de pruebas externo para desarrollar las pruebas del sistema. Por consi-

guíente, algunos enfoques para el desarrollo iterativo tienen un proceso de pruebas muy informal.

Para evitar algunos de los problemas de las pruebas y de la validación del sistema, XP pone más énfasis en el proceso de pruebas que otros métodos ágiles. Las pruebas del sistema son fundamentales en XP, en la que se ha desarrollado un enfoque que reduce la probabilidad de producir nuevos incrementos del sistema que introduzcan errores en el software existente.

Las características clave de las pruebas en XP son:

1. Desarrollo previamente probado.
2. Desarrollo de pruebas incremental a partir de los escenarios.
3. Participación del usuario en el desarrollo de las pruebas y en la validación,
4. El uso de bancos de pruebas automatizados.

El desarrollo previamente probado es una de las innovaciones más importantes en XP. Al escribir primero las pruebas se define implícitamente tanto una interfaz como una especificación del funcionamiento para la funcionalidad a desarrollar. Se reducen los problemas en los requerimientos y las confusiones de la interfaz. Se puede adoptar este enfoque en cualquier proceso en el que exista una relación clara entre un requerimiento del sistema y el código que implementa ese requerimiento. En XP, siempre puede verse este vínculo debido a que las tarjetas de historias que representan los requerimientos se dividen en tareas, y las tareas son las unidades principales de implementación.

Como se ha señalado, los requerimientos de usuario en XP se expresan como escenarios o historias y el usuario establece las prioridades de éstos para su desarrollo. El equipo de desarrollo evalúa cada escenario y lo divide en tareas. Cada tarea representa una característica distinta del sistema y se puede diseñar entonces una prueba de unidad para esa tarea. Por ejemplo, en la Figura 17.7 se muestran algunas de las tarjetas de tareas ***(task cards)*** desarrolladas a partir de la tarjeta de historia para la descarga de documentos (Figura 17.6).

Cada tarea genera una o más pruebas de unidad que verifican la implementación descrita en esa tarea. Por ejemplo, la Figura 17.8 es una descripción abreviada de un caso de pruebas que se ha desarrollado para comprobar que se ha implementado un número de tarjeta de crédito válido.

El papel del cliente en el proceso de pruebas es ayudar a desarrollar las pruebas de aceptación para las historias que se tienen que implementar en la siguiente entrega del sistema. Como se explica en el Capítulo 23, las pruebas de aceptación son el proceso en el que se prueba el sistema utilizando datos del cliente para verificar que cumple sus necesidades reales.

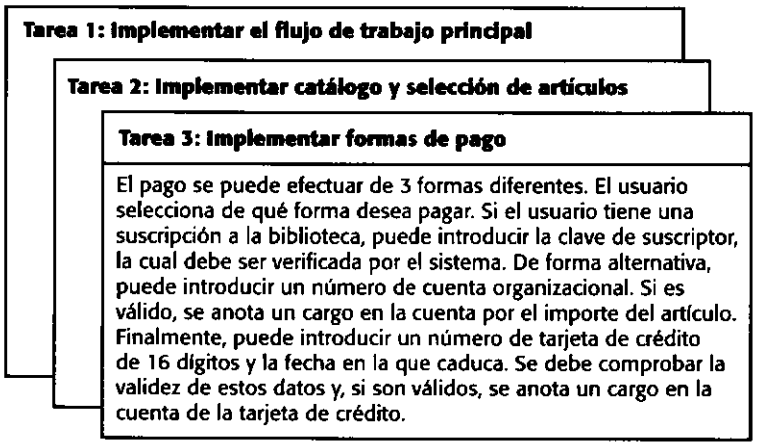

Figura 17.7 Tarjetas de tareas para la descarga de documentos.

Prueba 4: Prueba de la validez de la tarjeta de crédito
Entrada: Una cadena que representa el número de tarjeta de crédito y dos enteros que representan el mes y el año de la caducidad de la tarjeta. **Pruebas:** Comprobar que todos los bytes de la cadena son dígitos. Comprobar que el mes se encuentra entre 1 y 12 y que el año es mayor o igual que el año actual. Utilizando los 4 primeros dígitos del número de tarjeta de crédito, comprobar que el emisor de la tarjeta es válido consultando la tabla de emisores de tarjetas. Comprobar la validez de la tarjeta de crédito enviando el número de tarjeta y la fecha en la que caduca el emisor de la tarjeta. **Salida:** OK o un mensaje de error indicando que la tarjeta no es válida.

Figura 17.8
Descripción del caso de prueba para comprobar la validez de la tarjeta de crédito.

En XP, las pruebas de aceptación, como el desarrollo, son increméntales. Para esta historia en particular, la prueba de aceptación implicaría seleccionar varios documentos, pagarlos de diferentes formas e imprimirlos en impresoras distintas. En la práctica, probablemente se desarrollaría una serie de pruebas de aceptación en vez de una única prueba.

El desarrollo previamente probado y el uso de bancos de pruebas automatizados son las principales virtudes del enfoque de la XP. En vez de escribir el código del programa y luego las pruebas de ese código, el desarrollo previamente probado significa que las pruebas se escriben antes que el código. Fundamentalmente, las pruebas se escriben como un componente ejecutable antes de que se implemente la tarea. Una vez que se ha implementado el software, se pueden ejecutar las pruebas inmediatamente. Este componente de pruebas debe ser una aplicación independiente, debe simular el envío de la entrada a probar y debe verificar que el resultado cumple la especificación de salida. El banco de pruebas automatizado es un sistema que envía a ejecución estas pruebas automatizadas.

Con el desarrollo previamente probado, siempre hay un conjunto de pruebas que se pueden ejecutar fácilmente y de forma rápida. Esto significa que siempre que se añada cualquier funcionalidad al sistema, se pueden ejecutar las pruebas y detectar inmediatamente los problemas que el código nuevo haya introducido.

En el desarrollo previamente probado, las personas encargadas de ¡mplementar las tareas tienen que comprender perfectamente la especificación, de modo que puedan escribir las pruebas del sistema. Esto significa que se tienen que clarificar las ambigüedades y omisiones en la especificación antes de que empiece la implementación. Además, esto también evita el problema del «retraso de las pruebas», en el que, debido a que los desarroliadores del sistema trabajan a un ritmo más elevado que los probadores, cada vez se adelanta más la implementación sobre las pruebas y hay una tendencia a saltarse las pruebas para poder mantener la agenda establecida.

Sin embargo, el desarrollo previamente probado no siempre funciona según lo previsto. Los programadores prefieren la programación a las pruebas y algunas veces escriben pruebas incompletas que no comprueban las situaciones excepcionales. Además, puede ser difícil escribir algunas pruebas. Por ejemplo, en una interfaz de usuario compleja, con frecuencia es difícil escribir las pruebas de unidad para el código que implementa la «vista lógica» y el flujo de trabajo entre las pantallas. Por último, es difícil juzgar la completitud de un conjunto de pruebas. Aunque se pueda tener una gran cantidad de pruebas del sistema, ese conjunto de pruebas puede no proporcionar una cobertura completa. Es posible que no se ejecuten partes cruciales del sistema y, por lo tanto, se queden sin probar.

Contar con el cliente para el apoyo al desarrollo de las pruebas de aceptación es a veces un problema serio en el proceso de pruebas de la XP. Las personas que adoptan el papel de cliente tienen muy poco tiempo disponible y es posible que no puedan trabajar a tiempo completo con el equipo de desarrollo. El cliente puede pensar que proporcionar los requerimientos es contribución suficiente y puede ser reacio a participar en el proceso de pruebas.

17.2.2 Programación en parejas

Otra práctica innovadora que se ha introducido es que los programadores trabajan en parejas para desarrollar el software. De hecho, se sientan juntos en la misma estación de trabajo para desarrollar el software. El desarrollo no siempre implica la misma pareja de personas trabajando juntas. Más bien, la idea es que las parejas se creen de forma dinámica para que todos los miembros del equipo puedan trabajar con los otros miembros en una pareja de programación durante el proceso de desarrollo.

El uso de la programación en parejas tiene varias ventajas:

1. Apoya la idea de la propiedad y responsabilidad comunes del sistema. Esto refleja la idea de Weinberg de la programación sin ego (Weinberg, 1971), en la que el equipo como un todo es dueño del software y las personas individuales no tienen la culpa de los problemas con el código. En cambio, el equipo tiene una responsabilidad colectiva para resolver estos problemas.
2. Actúa como un proceso de revisión informal ya que cada línea de código es vista por al menos dos personas. Las inspecciones y revisiones del código (tratadas en el Capítulo 22) consiguen descubrir un alto porcentaje de errores del software. Sin embargo, requiere mucho tiempo organizarías y, por lo general, generan retrasos en el proceso de desarrollo. A pesar de que la programación en parejas es un proceso menos formal que probablemente no encuentre tantos errores, es un proceso de inspección mucho más económico que las inspecciones formales de programas.
3. Ayuda en la refactorización, la cual es un proceso de mejora del software. Un principio de la XP es que se debe refactorizar constantemente el software. Es decir, se deben escribir nuevamente partes del código para mejorar su claridad y estructura. El problema para implementar esto en un entorno de desarrollo normal es que es un esfuerzo que se gasta para obtener un beneficio a largo plazo, y se puede juzgar a una persona que practique la refactorización como menos eficiente que otra que simplemente realice el desarrollo del código. Cuando se utiliza la programación en parejas y la propiedad colectiva, otros se benefician inmediatamente con la refactorización, por lo que probablemente apoyarán el proceso.

Podría pensarse que la programación en parejas es menos eficiente que la programación individual y que una pareja de desarroliadores produciría, como mucho, la mitad del código que dos personas trabajando individualmente. Sin embargo, diversos estudios sobre desarrollos que utilizan XP no confirman esto. La productividad del desarrollo con programación en parejas parece ser comparable con la de dos personas trabajando de forma independiente (Williams ***el al.,*** 2000). Las razones para esto son que las parejas discuten sobre el software antes de empezar el desarrollo, por lo probablemente tengan menos comienzos en falso y tengan que rehacer menos trabajo, y que el número de errores evitados debido a la inspección informal es tal que se pasa menos tiempo arreglando errores descubiertos durante el proceso de pruebas.

17.3 Desarrollo rápido de aplicaciones

Aunque los métodos ágiles como un enfoque para el desarrollo iterativo han recibido una gran atención en los últimos años, los sistemas de negocio se han desarrollado de forma iterativa durante muchos años utilizando técnicas de desarrollo rápido de aplicaciones. Las técnicas de desarrollo rápido de aplicaciones (RAD) evolucionaron de los llamados lenguajes de cuarta generación en los años 80 y se utilizan para desarrollar aplicaciones con un uso intensivo de datos. Por consiguiente, normalmente están organizadas como un conjunto de herramientas que permiten crear datos, buscarlos, visualizarlos y presentarlos en informes. La Figura 17.9 ilustra una organización típica de un sistema RAD.

Las herramientas que se incluyen en un entorno RAD son:

1. ***Un lenguaje de programación de bases de dalos,*** que contiene conocimiento de la estructura de la base de datos y que incluye las operaciones básicas de manipulación de bases de datos. El lenguaje estándar de programación de base de datos es SQL (Groff ***et ai,*** 2002). Los comandos SQL se pueden introducir directamente o generar de forma automática a partir de formularios rellenados por los usuarios finales.
2. ***Un generador de interfaces,*** que se utiliza para crear formularios de introducción y visual i zación de datos.
3. ***Enlaces a aplicaciones de oficina,*** como una hoja de cálculo para el análisis y manipulación de información numérica o un procesador de textos para la creación de plantillas de informes.
4. ***Un generador de informes,*** que se utiliza para definir y crear informes a partir de la información de la base de datos.

Los sistemas RAD tienen éxito debido a que, como se explicó en el Capítulo 13, las aplicaciones de negocio tienen muchas cosas en común. Esencialmente, a menudo estas aplicaciones comprenden la actualización de una base de datos y la producción de informes a partir de la información existente en ella. Se utilizan formularios estándar para las entradas y salidas. Los sistemas RAD están dirigidos a la producción de aplicaciones interactivas que se apoyan la abstracción de la información en una base de datos organizacional, presentándola a los usuarios finales en su terminal o estación de trabajo, y actualizando la base de datos con los cambios hechos por los usuarios.

Muchas de las aplicaciones de negocio se apoyan en formularios estructurados para las entradas y salidas, por lo que los entornos RAD proporcionan recursos potentes para la definición de pantallas y generación de informes. A menudo, las pantallas se definen como una se-

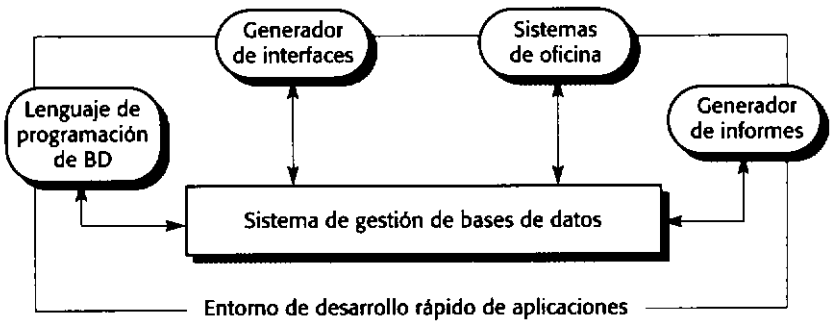

Figura 17.9
Un entorno de desarrollo rápido de aplicaciones.

rie de formularios vinculados (en una aplicación que estudiamos, hubo 137 definiciones de formularios), por lo que el sistema de generación de pantallas debe proporcionar:

1. ***Definición de formularios interactivos*** que permitan al desarrollador definir los campos a visualizar y la manera en que éstos deben organizarse.
2. ***Vinculación de los formularios*** que permitan al desarrollador especificar que ciertas entradas provocan la visualización de formularios adicionales.
3. ***Verificación de campos*** que permitan al desarrollador definir los rangos permitidos para los valores de entrada en los campos de los formularios.

Hoy en día, muchos entornos R A D permiten el desarrollo de interfaces de bases de datos basadas en navegadores web. Dichos entornos permiten el acceso a la base de datos desde cualquier lugar a través de una conexión válida de Internet. Esto reduce los costes de formación y de software, y permite a los usuarios externos tener acceso a una base de datos. Sin embargo, las limitaciones inherentes de los navegadores web y los protocolos de Internet implican que este enfoque no sea adecuado para sistemas en los que se requieran respuestas interactivas muy rápidas.

Actualmente, la mayoría de los sistemas R A D incluyen herramientas de programación visual que permiten desarrollar sistemas de forma interactiva. En vez de escribir un programa secuencial, el desarrollador del sistema manipula iconos gráficos que representan funciones, datos o componentes de interfaces de usuario, y asocia el procesamiento de secuencias de comandos con estos iconos. Se genera automáticamente un programa ejecutable a partir de la representación visual del sistema.

Los sistemas de desarrollo visual, como Visual Basic, permiten este enfoque basado en la reutilización para el desarrollo de aplicaciones. Los programadores de éstas construyen el sistema de forma interactiva definiendo la interfaz en términos de pantallas, campos, botones y menús. A éstos, se les asigna un nombre y se asocia el procesamiento de secuencias de comandos con partes individuales de la interfaz (por ejemplo, un botón denominado Simular). Estas secuencias de comandos pueden hacer llamadas a componentes reutilizables, a código de propósito especial o a una mezcla de ambos.

Este enfoque se ilustra en la Figura 17.10, la cual muestra una pantalla de aplicación que incluye menús en la parte superior, campos de entrada (los campos blancos a la izquierda de la pantalla), campos de salida (el campo gris a la izquierda de la pantalla) y botones (los rectángulos redondeados a la derecha de la pantalla). Cuando el sistema de programación visual ubica en la pantalla estas entidades, el desarrollador define qué componentes reutilizables se deben asociar con ellos o escribe un fragmento de programa para llevar a cabo el procesamiento requerido. También se muestran en la Figura 17.10 los componentes asociados con algunos de los elementos de la pantalla.

Visual Basic es un ejemplo muy sofisticado de un lenguaje de creación de secuencias de comandos (Ousterhout, 1998). Estos son lenguajes de alto nivel sin tipos diseñados que ayudan a integrar componentes para crear sistemas. Uno de los primeros lenguajes de este tipo fue el shell de Unix (Bourne, 1978; Gordon y Bieman, 1995); desde su desarrollo, se han creado varios lenguajes de creación de secuencias de comandos más potentes (Ousterhout, 1994; Lutz, 1996; Wall ***et al.,*** 1996). Estos lenguajes incluyen estructuras de control y juegos de herramientas gráficas que, como ilustra Ousterhout (Ousterhout, 1998), pueden reducir radicalmente el tiempo requerido para el desarrollo del sistema.

Este enfoque para el desarrollo de sistemas permite el desarrollo rápido de aplicaciones relativamente sencillas que pueden ser construidas por un equipo pequeño de personas. Es más difícil de organizar para sistemas más grandes que deben ser desarrollados por equipos con

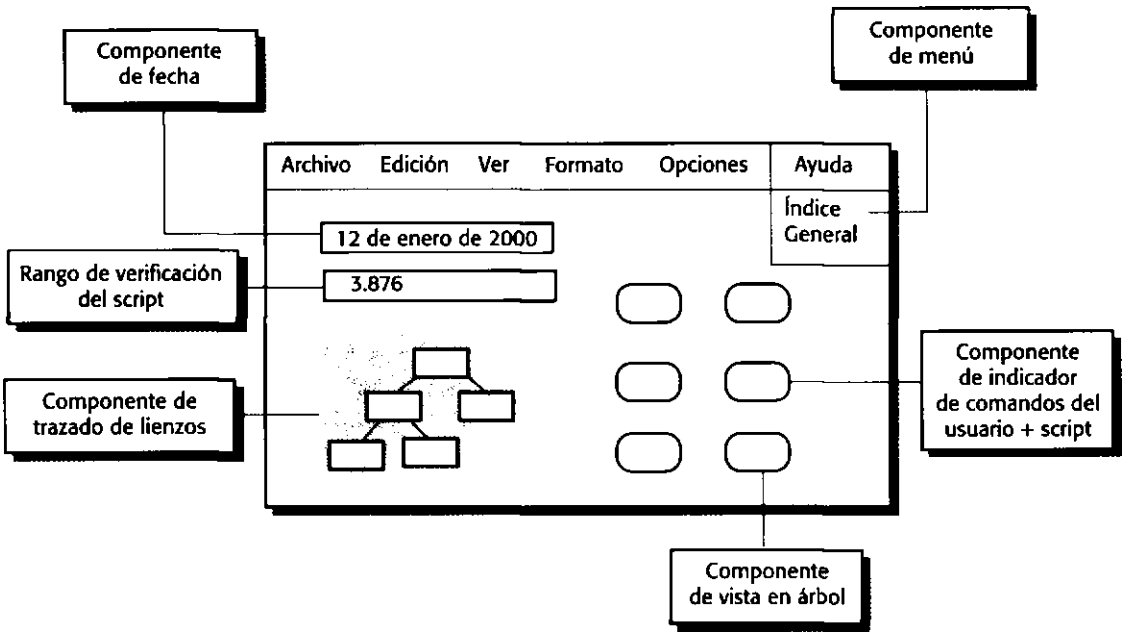

Figura 17.10 Programación visual con reutilización.

un mayor número de personas. No existe una arquitectura explícita del sistema y a menudo existen dependencias complejas entre las partes del sistema, lo que puede causar problemas cuando se requieran cambios. Además, debido a que los lenguajes de creación de secuencias de comandos se limitan a un conjunto específico de objetos en interacción, puede ser difícil implementar interfaces de usuario no estándares.

El ***desarrollo visual*** es un enfoque a RAD que utiliza la integración de componentes software reulilizables de grano fino. Un enfoque alternativo basado en la reutilización reutiliza «componentes» que son sistemas de aplicaciones completos. Este se denomina a veces desarrollo basado en COTS, donde COTS (Commercial Off-the-Shelf) significa que las aplicaciones ya están disponibles. Por ejemplo, si un sistema requiere las funcionalidades de un procesador de textos, podría utilizar un procesador de textos estándar como Microsoft Word. En el Capítulo 18 se estudia el desarrollo basado en COTS desde la perspectiva de la reutilización.

Para ilustrar el tipo de aplicación que se podría desarrollar utilizando un enfoque basado en COTS, consideremos el proceso de gestión de requerimientos descrito en el Capítulo 7. Un sistema de apoyo a dicha gestión necesita una forma de capturar y almacenar los requerimientos, producir los informes, descubrir las relaciones entre los requerimientos y gestionar estas relaciones como tablas de rastreo. En un enfoque basado en COTS, se podría construir un prototipo vinculando una base de datos (para almacenar los requerimientos), un procesador de textos (para capturar los requerimientos y los formatos de los informes), una hoja de cálculo (para gestionar las tablas de rastreo) y código escrito específicamente para encontrar las relaciones entre los requerimientos.

El desarrollo basado en COTS da acceso al desarrollador a toda la funcionalidad de una aplicación. Si ésta también proporciona recursos de creación de secuencias de comandos o de ajuste (por ejemplo, macros de Excel), éstos se pueden utilizar para desarrollar cierto tipo de funcionalidad de la aplicación. Para comprender este enfoque de desarrollo de aplicaciones

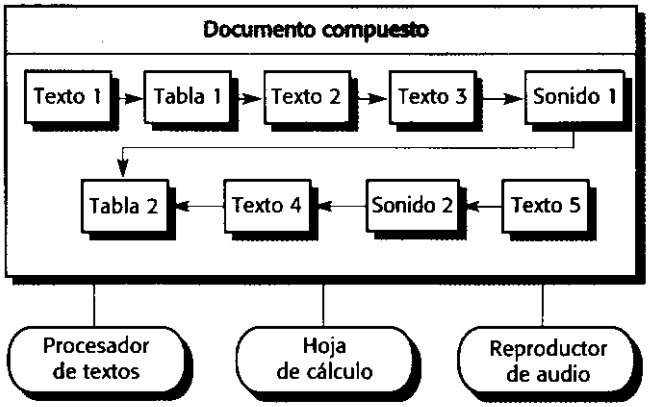

Figura 17.11 Vinculación de aplicaciones.

es de utilidad una metáfora de composición de documentos. Los datos procesados por el sistema se pueden organizar en un documento compuesto que actúa como un contenedor de varios objetos. Estos objetos contienen diferentes tipos de datos (como una tabla, un diagrama, un formulario) que pueden ser procesados por diferentes aplicaciones. Los objetos se enlazan y se clasifican para que al acceder a un objeto se inicie la aplicación asociada.

La Figura 17.11 ilustra un sistema de aplicaciones formado por un documento compuesto que incluye elementos de texto, de hojas de cálculo y archivos de sonido. Los elementos de texto son procesados por el procesador de textos; las tablas, por la aplicación de hojas de cálculo, y los archivos de sonido, por el reproductor de audio. Cuando un usuario del sistema accede a un objeto de un tipo particular, se llama a la aplicación asociada para proporcionar la funcionalidad al usuario. Por ejemplo, cuando se accede a objetos de tipo sonido, se llama al reproductor de audio para procesarlos.

La principal ventaja de este enfoque es que mucha de la funcionalidad de la aplicación se puede implementar rápidamente a un coste muy bajo. Los usuarios que ya estén familiarizados con las aplicaciones que componen el sistema no tendrán que aprender cómo utilizar las nuevas características. Sin embargo, si no saben cómo utilizar las aplicaciones, el aprendizaje puede ser difícil, especialmente si están confundidos con la funcionalidad innecesaria de la aplicación. Puede haber también problemas de rendimiento en la aplicación debido a la necesidad de cambiar de una aplicación del sistema a otra. Este esfuerzo adicional para realizar el cambio entre aplicaciones depende de la ayuda que proporcione el sistema operativo.

17.4 Prototipado del software

Como se ha indicado en la introducción de este capítulo, existen algunas circunstancias en las que. por razones prácticas o contractuales, no se puede utilizar un proceso de entrega del software incrémental. En esas situaciones, se completa una declaración de los requerimientos del sistema y es utilizada por el equipo de desarrollo como la base para el software del sistema. Como se ha explicado, se pueden conseguir algunos de los beneficios de un proceso de desarrollo incrementai creando un prototipo del software. Este enfoque se denomina a veces prototipado desechable debido a que el prototipo no es entregado al cliente o mantenido por el désarroi lador.

Un prototipo es una versión inicial de un sistema software que se utiliza para demostrar conceptos, probar opciones de diseño y, en general, informarse más del problema y sus posi-

bles soluciones. El desarrollo rápido e iterativo del prototipo es esencial, de modo que los costes sean controlados y los stakeholders del sistema puedan experimentar con el prototipo en las primeras etapas del proceso del software.

Un prototipo del software se puede utilizar de varias maneras en un proceso de desarrollo de software:

1. En el proceso de ingeniería de requerimientos, un prototipo puede ayudar en la obtención y validación de los requerimientos del sistema.
2. En el proceso de diseño del sistema, se puede utilizar un prototipo para explorar soluciones software particulares y para apoyar al diseño de las interfaces de usuario.
3. En el proceso de pruebas, se puede utilizar un prototipo para ejecutar pruebas back-to-back con el sistema que se entregarán al cliente.

Los prototipos del sistema permiten a los usuarios ver cómo éste apoya su trabajo. Pueden adquirir nuevas ideas para los requerimientos y encontrar áreas fuertes y débiles en el software. Entonces pueden proponer nuevos requerimientos del sistema. Además, a medida que se desarrolla el prototipo, puede revelar errores y omisiones en los requerimientos propuestos. Una función descrita en una especificación podría parecer útil y bien definida. Sin embargo, cuando la función se combina con otras, a menudo los usuarios comprueban que su visión inicial fue incorrecta o incompleta. La especificación del sistema podría modificarse para reflejar el cambio en la comprensión de los requerimientos.

Se puede utilizar un prototipo del sistema mientras se esté diseñando el sistema para llevar a cabo experimentos de diseño con el fin de verificar la viabilidad de un diseño propuesto. Por ejemplo, un diseño de una base de datos puede ser prototipado y probado para verificar que las consultas más comunes de los usuarios tienen el acceso a los datos más eficiente. El prototipado es también una parte fundamental del proceso de diseño de las interfaces de usuario. Debido a la naturaleza dinámica de las interfaces de usuario, las descripciones textuales y los diagramas no son suficientes para expresar los requerimientos de éstas. Por lo tanto, el prototipado rápido con la participación del usuario final es la única forma razonable de desarrollar interfaces gráficas de usuario para sistemas software.

Un problema importante en las pruebas del sistema es la validación de las pruebas, donde tiene que comprobarse si los resultados de una prueba son lo que se esperaba. Cuando está disponible un prototipo del sistema, se puede reducir el esfuerzo realizado en la comprobación de los resultados ejecutando pruebas back-to-back (Figura 17.12). Se envían los mismos casos de prueba tanto al prototipo como al sistema en prueba. Si ambos dan el mismo resultado, probablemente el caso de prueba no haya detectado ningún defecto. Si los resultados difieren, puede significar que hay un defecto en el sistema y se deben investigar las razones de la diferencia.

Por último, además de apoyar las actividades del proceso del software, se pueden utilizar prototipos a fin de reducir el tiempo requerido para desarrollar la documentación del usuario y para formarlos. Un sistema funcional, aunque limitado, está disponible de forma rápida para demostrar la viabilidad y utilidad de la aplicación a la dirección.

En un estudio de 39 proyectos de prototipado, Gordon y Bieman (Gordon y Bieman, 1995) observaron que los beneficios de utilizar el prototipado fueron:

1. Mejora en la usabilidad del sistema
2. Una mejor concordancia entre el sistema y las necesidades del usuario
3. Mejora en la calidad del diseño
4. Mejora en el mantenimiento
5. Reducción en el esfuerzo de desarrollo

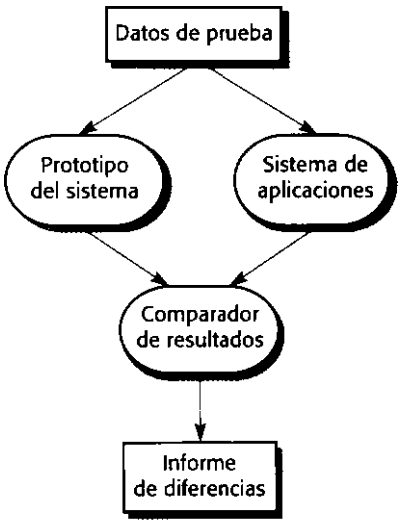

Figura 17.12
Pruebas hark-tn-harlí

Su estudio sugiere que las mejoras en la usabilidad y en la definición de requerimientos del usuario debido a la utilización de un prototipo no significan necesariamente un incremento general en los costes de desarrollo. Por lo general, la construcción de prototipos incrementa los costes en las etapas iniciales del proceso del software pero reduce los costes posteriores en el proceso de desarrollo. La razón principal de esto es que se evita rehacer el trabajo durante el desarrollo debido a que los clientes solicitan menos cambios en el sistema. Sin embargo, Gordon y Bieman observaron que el rendimiento general del sistema algunas veces se degrada si se reutiliza código ineficiente proveniente del prototipo.

En la Figura 17.13 se muestra un modelo del proceso para el desarrollo de prototipos. Los objetivos de la construcción de éstos deben ser explícitos desde el inicio del proceso. Estos pueden ser desarrollar un sistema para construir un prototipo de la interfaz de usuario, desarrollar un sistema para validar los requerimientos funcionales del sistema o para demostrar la viabilidad de la aplicación a la dirección. El mismo prototipo no puede cumplir todos los objetivos. Si éstos no se especifican, la dirección o los usuarios finales pueden mal interpretar la función del prototipo. En consecuencia, es posible que no obtengan los beneficios que esperan del desarrollo de éste.

La siguiente etapa en el proceso es decidir qué incluir y, quizás lo más importante, qué excluir del sistema prototipo. Para reducir los costes de la construcción del prototipo y acelerar la agenda de entregas, se puede excluir de éste cierta funcionalidad. Se puede decidir relajar los requerimientos no funcionales, como el tiempo de respuesta y la utilización de la memoria. La gestión y manejo de errores se puede pasar por alto o hacerse de forma rudimentaria,

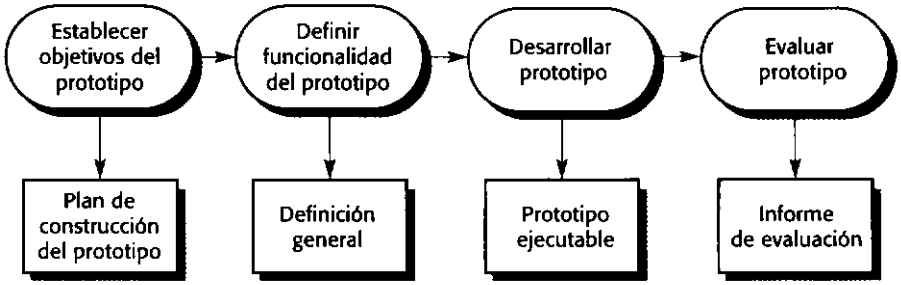

Figura 17.13
El proceso de desarrollo de prototipos.

a menos que el objetivo del prototipo sea establecer una interfaz de usuario. Se pueden reducir los estándares de fiabilidad y calidad de la programación.

La etapa final del proceso es la evaluación del prototipo. En ésta se debe prever la formación del usuario y se deben utilizar los objetivos del prototipo para obtener un plan de evaluación. Los usuarios requieren tiempo para acostumbrarse a un nuevo sistema y utilizarlo de forma normal. Una vez que lo utilizan, descubren los errores y omisiones en los requerimientos.

Un problema general con el desarrollo de un prototipo desechable ejecutable es que el modo de utilizarlo puede que no se corresponda con el modo en que se utiliza el sistema final entregado. El probador del prototipo puede no ser el típico usuario de éste. El tiempo de formación durante la evaluación de! prototipo puede ser insuficiente. Si el prototipo es lento, los evaluadores pueden modificar su forma de trabajar y evitar aquellas características que tengan tiempos de respuesta lentos. Si el sistema final tiene mejor tiempo de respuesta, pueden utilizarlo de forma diferente.

Algunas veces, los gerentes presionan a los desarrolladores para que entreguen los prototipos desechables, especialmente cuando existen retrasos en la entrega de la versión final de! software. En vez de hacer frente a los retrasos en el proyecto, los gerentes pueden creer que entregar un sistema incompleto o de baja calidad es mejor que nada. Sin embargo, normalmente esto no es aconsejable por las siguientes razones:

1. Puede ser imposible ajustar el prototipo para que cumpla con los requerimientos no funcionales que fueron dejados de lado durante su desarrollo, como los de rendimiento, protección, robustez y fiabilidad.
2. El cambio rápido durante el desarrollo significa, inevitablemente, que no se documenta el prototipo. La única especificación del diseño es el código del prototipo. Esto no es suficiente para el mantenimiento a largo plazo.
3. Los cambios hechos durante el desarrollo del prototipo probablemente degradan la estructura del sistema. Éste será difícil y caro de mantener.
4. Los estándares de calidad organizacionales normalmente se relajan para el desarrollo del prototipo.

Los prototipos desechables no tienen que ser ejecutables para ser de utilidad en el proceso de ingeniería de requerimientos. Como se explica en el Capítulo 16, las maquetas en papel de la interfaz de usuario (Rettig. 1994) pueden ser efectivas para ayudar a los usuarios a perfeccionar un diseño de la interfaz y a trabajar a través de escenarios de utilización. Éstos son muy baratos de desarrollar y se pueden construir en pocos días. Una extensión de esta técnica es un prototipo Mago de Oz en el que sólo se desarrolla la interfaz de usuario. Los usuarios interactúan con esta interfaz, pero sus peticiones se pasan a una persona que los interpreta y muestra la respuesta apropiada.

PUNTOS CLAVE

- **Al crecer la presión por una entrega rápida del software, se utiliza cada vez más un enfoque iterativo para et desarrollo del software como una técnica de desarrollo estándar para sistemas pequeños y de tamaño medio, especialmente en et dominio de los negocios.**

- Los métodos ágiles son métodos de desarrollo iterativo que se centran en la especificación, diseño e Implementación del sistema de forma incremental. Implican directamente a tos usuarios en el proceso de desarrollo. Reducir la sobrecarga en cuanto al esfuerzo de desarrollo puede hacer posible un desarrollo del software más rápido.
- La programación extrema es un método ágil conocido que integra una variedad de buenas prácticas de programación, como tas pruebas sistemáticas, la continua mejora del software y ta participación del cliente en el equipo de desarrollo.
- Un punto fuerte particular de la programación extrema es el desarrollo de pruebas automatizadas antes de que se cree una funcionalidad en un programa. Se deben ejecutar de forma satisfactoria todas las pruebas cuando se integra un incremento en un sistema.
- Et desarrollo rápido de aplicaciones implica la utilización de entornos de desarrollo que incluyan herramientas potentes para apoyar la producción del sistema. Éstas comprenden lenguajes de programación de bases de datos, generadores de formularios e informes, y enlaces a aplicaciones de oficina.
- El prototipado desechabte es un proceso de desarrollo iterativo en el que se utiliza un sistema prototipo para explorar los requerimientos y las opciones de diseño. Este prototipo no está destinado para su utilización por parte de los clientes del sistema.
- Cuando se implementa un prototipo desechabte, primero se tienen que desarrollar las partes del sistema que menos se comprenden; por el contrario, en un enfoque de desarrollo incremental, se empieza desarrollando tas partes del sistema que mejor se comprenden.

LECTURAS ADICIONALES

Extreme Programming Explained. Éste fue el primer libro sobre XP y es todavía, quizás, el que más merece la pena leer. Explica el enfoque desde la perspectiva de uno de sus inventores, y llega muy claramente su entusiasmo. (Kent Beck, **2000,** Addison-Wesley.)

«Get ready for agile methods, with care». Una seria crítica de los métodos ágiles que analiza sus puntos fuertes y débiles, redactada por ingenieros de software tremendamente experimentados. (B. Boehm, ***IEEE Computer,*** enero de **2002.)**

«Scripting: Higher-level programming for the **2ist** century». Una visión general de tos lenguajes de creación de secuencias de comandos por el inventor de Tcl/Tk, quien describe las ventajas de este enfoque para el desarrollo rápido de aplicaciones. O- K. Ousterhout, ***IEEE Computer,*** marzo de **1998.)**

DSDM: Dynamic Systems Development Meted. Una descripción de un enfoque para el desarrollo rápido de aplicaciones que algunas personas consideran que es un ejemplo inicial de un método ágil. O- Stapleton, **1997.** Addison-Wesley.)

EJERCICIOS

17.1 Explique por qué la entrega y desarrollo rápidos de sistemas nuevos es a menudo más importante para las empresas que una funcionalidad detallada de estos sistemas.

17.2 Explique cómo los principios subyacentes a los métodos ágiles conducen a un desarrollo y utilización del software acelerados.

17.3 ¿Cuándo recomendaría *contra* el uso de un método ágil para desarrollar un sistema software?

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

17.5 Explique por qué el desarrollo previamente probado ayuda al programador a desarrollar una mejor comprensión de los requerimientos del sistema. ¿Cuáles son los problemas potenciales del desarrollo previamente probado?

17.6 Sugiera cuatro razones por las que el índice de productividad de programadores trabajando en parejas sea aproximadamente el mismo que dos programadores trabajando de forma individual.

17.7 Se le ha pedido que investigue la viabilidad de la construcción de prototipos en el proceso de desarrollo de software de su organización. Redacte un informe para su jefe que explique las clases de proyectos donde se deba utilizar el prototipado, y precise los costes y beneficios esperados de éste.

17.8 Un administrador de software trabaja en el desarrollo de un proyecto de diseño de un sistema de ayuda que traduce los requerimientos del software a una especificación software formal. Comente las ventajas y desventajas de las siguientes estrategias de desarrollo:

a) Desarrollo de un prototipo desechable, su evaluación y posterior revisión de los requerimientos del sistema. Desarrollo del sistema final utilizando C.

b) Desarrollo del sistema a partir de los requerimientos existentes utilizando Java, posterior modificación para que se adapte a cualquier cambio en los requerimientos del usuario.

c) Desarrollo del sistema utilizando un desarrollo incremental con la participación del cliente en el equipo de desarrollo.

17.9 Una organización benéfica te ha solicitado construir un prototipo de un sistema que dé seguimiento al historial de todas las donaciones que ha recibido. Este sistema tiene que almacenar los nombres y direcciones de los donantes, sus intereses particulares, la cantidad donada y cuándo fue donada. Si la donación está por encima de cierta cantidad, el donante puede poner condiciones para hacerla (por ejemplo, debe gastarse en un proyecto en particular), y el sistema debe mantener el historial de éstas y cómo fueron gastadas. Comente cómo construiría el prototipo de este sistema, teniendo presente que la organización tiene una mezcla de trabajadores asalariados y voluntarios. Muchos de los voluntarios son jubilados que tienen poca o nula experiencia en informática.

17.10 Usted ha desarrollado un prototipo desechable de un sistema para un cliente que está muy contento con él. Sin embargo, sugiere que no existe necesidad de desarrollar otro sistema, sino que debe entregarle el prototipo y le ofrece un precio excelente por él. Usted sabe que habrá problemas futuros en el mantenimiento del sistema. Explique cómo debería responder a este cliente.