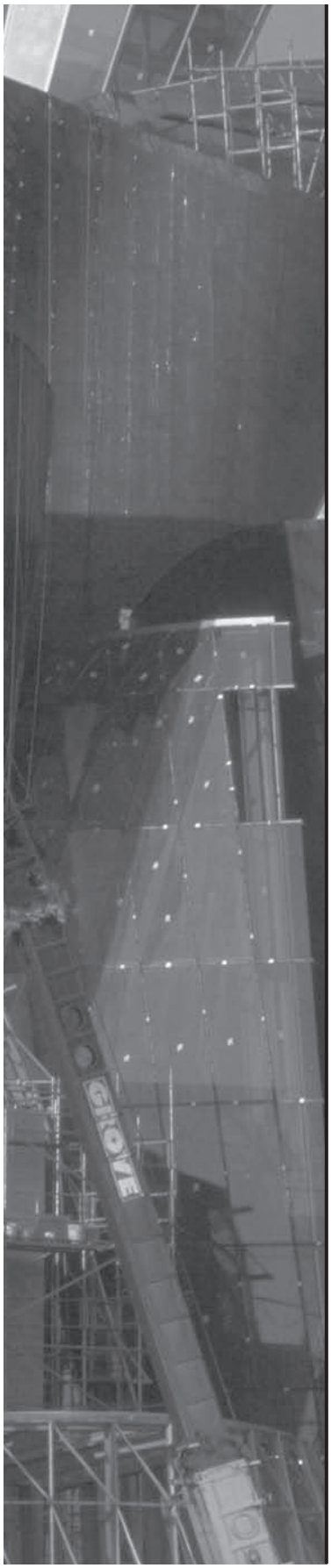

3

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-

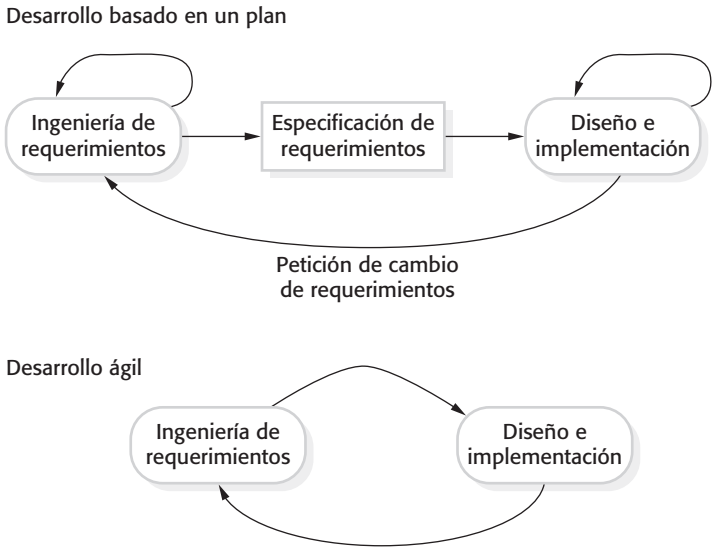

Figura 3.2
Especificación ágil y dirigida por un plan

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.

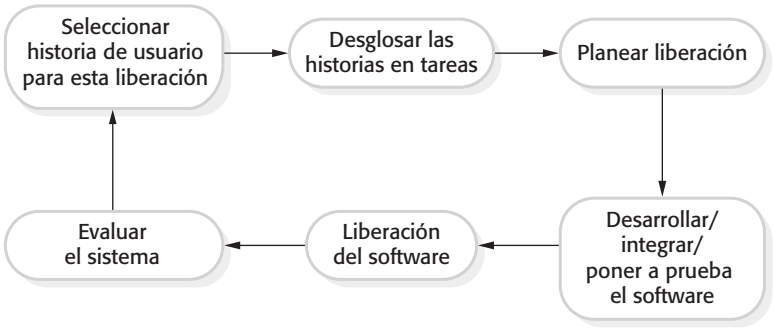

Figura 3.3 El ciclo de liberación de la programación extrema

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.

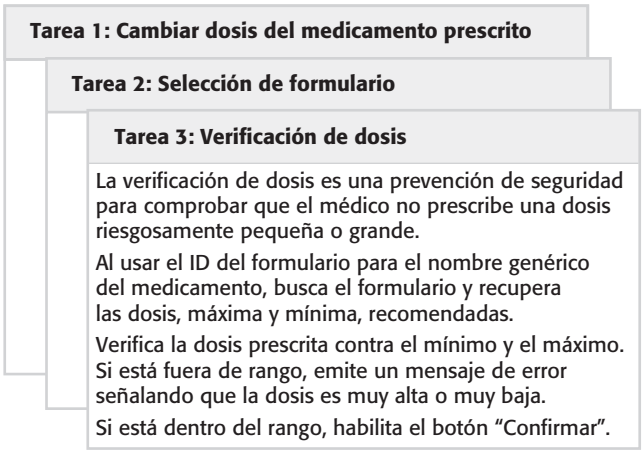

Figura 3.6 Ejemplos de tarjetas de tarea para prescripción de medicamentos.

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.