

DESARROLLO ÁGIL

3

CONSEPTOS CLAVE 56
agilidad 56
Grisol 72

Desarrollo adaptativo 68
Desarrollo estable 68
Desarrollo software 73

DIC 72
historias 62
MDSD 71

proceso ágil 58
Proceso unificado ágil 75

proceso XP 62
programación extrema 61

programación por proyectos 64
rediseño 63

Scrum 69
velocidad del proyecto 63

XP industrial 65

En 2001, Kent Beck y otros 16 notables desarrolladores de software, escritores y consultores (Becol) al (grupo conocido como la 'Alianza Ágil') firmaron el "Manifiesto por el desarrollo ágil de software". En él se establecía lo siguiente:

Estamos descubriendo formas mejores de desarrollar software, por medio de hacerlo y de dar ayuda a otros para que lo hagan. Ese trabajo nos ha hecho valorar:

Los individuos y sus interacciones, sobre los procesos y las herramientas de software que funcionan, más que la documentación exhaustiva

La colaboración con el cliente, y no tanto la negociación del contrato

Responder al cambio, mejor que apegarse a un plan

Es decir, saben bien son valiosos los conceptos que aparecen en segundo lugar, valoramos más los que aparecen en primer sitio.

Un manifiesto normalmente se asocia con un movimiento político emergente: ataca a la vieja guardia y sugiere un cambio revolucionario (se espera que para mejorar). En cierta forma, de eso es de lo que trata el desarrollo ágil.

Aunque las ideas subyacentes que lo guían han estado durante muchos años entre nosotros, ha sido en menos de dos décadas que cristalizaron en un 'movimiento'. Los métodos ágiles se desarrollaron como un esfuerzo por superar las debilidades reales y perceptibles de la ingeniería de software convencional. El desarrollo ágil proporciona beneficios importantes, pero no es

UNA MIRADA RÁPIDA

¿Qué es? La ingeniería de software ágil combina una filosofía con un conjunto de lineamientos de desarrollo. La filosofía pone el énfasis en: la satisfacción del cliente y en la entrega rápida de software incremental, los equipos pequeños y muy motivados para efectuar el proyecto, los métodos informales, los productos del trabajo con mínima ingeniería de software y la sencillez general en el desarrollo. Los lineamientos de desarrollo enfatizan la retroalimentación y el diseño (aunque estos actividades no se desdoblaron) y la comunicación activa y continua entre desarrolladores y clientes.

¿Qué hace? Los ingenieros de software y otros participantes en el proyecto (gerentes, clientes, usuarios finales, etc.) trabajan juntos en un proyecto ágil, formando un equipo con organización propia y que controla su propio destino. Un equipo ágil facilita la comunicación y colaboración entre aquellos a quienes sirve.

¿Por qué es importante? El ambiente moderno de negocios que genera sistemas basados en computadora y productos de software evoluciona rápida y constantemente. La ingeniería de software ágil representa una alternativa

razonable a la ingeniería de software convencional para ciertas clases de software y en algunos tipos de proyectos. Asimismo, se ha demostrado que conduce con rapidez sistemas exitosos.

¿Cuáles son los pasos? Un nombre más apropiado para el desarrollo ágil sería "ingeniería de software ligero". Permanecen las actividades estructurales fundamentales: comunicación, planeación, modelado, construcción y despliegue. Pero se transforman en un conjunto mínimo de tareas que lleva al equipo del proyecto hacia la construcción y entrega (algunas personas dirían que esto se hace a costa del análisis del problema y del diseño de la solución).

¿Cuál es el producto final? Tanto el cliente como el ingeniero de software tienen la misma perspectiva: el único producto del trabajo realmente importante es un "incremento de software" operativo que se entrega al cliente exactamente en la fecha acordada.

¿Cómo me aseguro de que lo hice bien? El trabajo estará bien hecho si el equipo ágil concuerda en que el proceso funciona y en que produce incrementos de software utilizables que satisfagan al cliente.

aplicable a todos los proyectos, productos, personas y situaciones. No es la antítesis de la práctica de la ingeniería de software sólida y puede aplicarse como filosofía general para todo el trabajo de software.

Es frecuente que en la economía moderna sea difícil o imposible predecir la forma en la que evolucionará un sistema basado en computadora (por ejemplo, una aplicación con base en web). Las condiciones del mercado cambian con rapidez, las necesidades de los usuarios finales se transforman y emergen nuevas amenazas sin previo aviso. En muchas situaciones no se será posible definir los requerimientos por completo antes de que el proyecto comience. Se debe ser suficientemente ágil para responder a lo fluido que se presenta el ambiente de negocios.

La fluidez implica cambio, y el cambio es caro, en particular si es descontrolado o si se admistra mal. Una de las características más atractivas del enfoque ágil es su capacidad de reducir los costos del cambio durante el proceso del software.

¿Significa esto que el reconocimiento de los retos planteados por las realidades modernas hace que sean descartables los valiosos principios, conceptos, métodos y herramientas de la ingeniería del software? No, en absoluto... ¡Sí! que todas las disciplinas de la ingeniería, la del software evolucionan en forma continua. Puede adaptarse con facilidad para que satisfaga los deseos que surgen de la demanda de agilidad.

En un libro que suscita la reflexión sobre el desarrollo de software ágil, Alistair Cockburn [Coco2] argumenta que los modelos de proceso prescriptivo, introducidos en el capítulo 2, tienen una falla grande: *olvidan las flaquezas de las personas cuando construyen software*. Los ingenieros de software no son robots. Sus estilos de trabajo varían mucho, tienen diferencias significativas en habilidad, creatividad, orden, consistencia y espontaneidad. Algunos se comunican bien por escrito, pero otros no. Cockburn afirma que los modelos de proceso pueden "manejar las carencias de disciplina o tolerancia de las personas comunes" y que los modelos de proceso más prescriptivo eligen la disciplina. Dice: "Como la consistencia de las acciones es una debilidad humana, las metodologías que requieren mucha disciplina son frágiles."

Para funcionar, los modelos de proceso deben proveer un mecanismo realista que estimule la disciplina necesaria, o deben caracterizarse por la 'tolerancia' con las personas que hacen el trabajo de ingeniería de software. Invariamente, las prácticas tolerantes son más fáciles de adoptar y sostener por parte de la comunidad del software, pero son menos productivas (como admite Cockburn). Debe considerarse la negociación entre ellas, como en todas las cosas de la vida.

3.1 ¿QUÉ ES LA AGILIDAD?

Pero, ¿qué es la agilidad en el contexto del trabajo de la ingeniería de software? Ivar Jacobson [Jacob2] hace un análisis útil:

La *agilidad* se ha convertido en la palabra mágica de hoy para describir un proceso del software moderno. Todos son ágiles. Un equipo ágil es diestro y capaz de responder de manera apropiada a los cambios. El cambio es de lo que trata el software en gran medida. Hay cambios en el software que se construye, en los miembros del equipo, debido a las nuevas tecnologías, de todas clases y que tienen un efecto en el producto que se elabora o en el proyecto que lo crea. Deben introducirse apoyos para el cambio en todo lo que se haga en el software; en ocasiones se hace porque es el alma y corazón de éste. Un equipo ágil reconoce que el software es desarrollado por individuos que trabajan en equipo, y que su capacidad, su habilidad para colaborar, es el fundamento para el éxito del proyecto.

Desde el punto de vista de Jacobson, la ubicuidad del cambio es el motor principal de la agilidad. Los ingenieros de software deben ir rápido si han de adaptarse a los cambios veloces que describen Jacobson.

¹ En ocasiones se conoce a los métodos ágiles como métodos ligeros o métodos esbeltas.



No cometas el error de suponer que la agilidad le da permiso para improvisar soluciones. Se requiere de un proceso, y la disciplina es esencial.

Però la agilidad es algo más que una respuesta efectiva al cambio. También incluye la filosofía expuesta en el manifiesto citado al principio de este capítulo. Esta recomienda las estructuras de equipo y las actitudes que hacen más fácil la comunicación entre los miembros del equipo, tecnológicos y gente de negocios, entre los ingenieros de software y sus gerentes, etc.; pone el énfasis en la entrega rápida de software funcional y resta importancia a los productos intermedios del trabajo (lo que no siempre es bueno); adopta al cliente como parte del equipo de desarrollo y trabaja para eliminar la actitud de “nosotros y ellos” que todavía invade muchos proyectos de software; reconoce que la planeación en un mundo incierto tiene sus límites y que un plan de proyecto debe ser flexible.

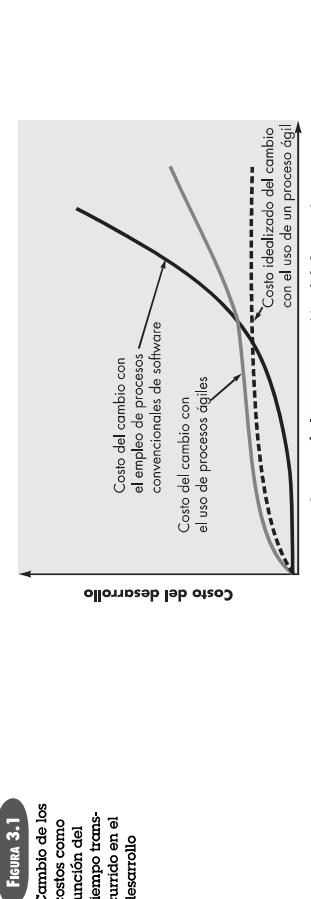
La agilidad puede aplicarse a cualquier proceso del software. Sin embargo, para lograrlo es esencial que éste se diseñe en forma que permita al equipo del proyecto adaptar las tareas y hacerlas directas, ejecutar la planeación de manera que entienda la *fluidez* de un enfoque ágil del desarrollo, eliminar todos los productos del trabajo excepto los más esenciales y mantenerlos esbeltas, y poner el énfasis en una estrategia de entrega incremental que haga trabajar al software tan rápido como sea posible para el cliente, según el tipo de producto y el ambiente de operación.

3.2 LA AGILIDAD Y EL COSTO DEL CAMBIO

La sabiduría convencional del desarrollo de software (apoyada por décadas de experiencia) señala que el costo se incrementa en forma no lineal a medida que el proyecto avanza (véase la figura 3.1, curva continua negra). Es relativamente fácil efectuar un cambio cuando el equipo de software reúne los requerimientos (al principio de un proyecto). El escenario de uso tal vez tenga que modificarse, la lista de funciones puede aumentar, o editarse una especificación escrita. Los costos de hacer que esto funcione son mínimos, y el tiempo requerido no perjudicará el resultado del proyecto. Pero, ¿qué pasa una vez transcurridos algunos meses? El equipo está a la mitad de las pruebas de validación (algo que ocurre cuando el proyecto está relativamente avanzado) y un participante de importancia solicita que se haga un cambio funcional grande. El cambio requiere modificar el diseño de la arquitectura del software, el diseño y construcción de tres componentes nuevos, hacer cambios en otros cinco componentes, diseñar nuevas pruebas, etc. Los costos aumentan con rapidez, y no son pocos el tiempo y el dinero requeridos para asegurar que se haga el cambio sin efectos colaterales no intencionados.

Los defensores de la agilidad (por ejemplo [Bac00]) y [Amh04]) afirman que un proceso ágil bien diseñado “aplana” el costo de la curva de cambio (véase la figura 3.1, curva continua y

Cita:
“La agilidad es dinámica, espe-
cífica en el contexto, acepta
con entusiasmo el cambio y se
orienta al crecimiento.”
Steven Goldmann et al.



CLAVE
Un proceso ágil reduce el costo del cambio porque el software se entrega en incrementos y esto forma el cambio se controla mejor.

sombreada), lo que permite que el equipo de software haga cambios en una fase tardía de un proyecto de software sin que haya un efecto notable en el costo y en el tiempo. El lector ya sabe que el proceso ágil incluye la entrega incremental. Cuando ésta se acopla con otras prácticas ágiles, como las pruebas unitarias continuas y la programación por parejas (que se estudia más adelante, en este capítulo), el costo de hacer un cambio disminuye. Aunque hay debate sobre el grado en el que se aplana la curva de costo, existen evidencias [Coco01] al que sugieren que es posible lograr una reducción significativa del costo.

3.3 ¿QUÉ ES UN PROCESO ÁGIL?

Qualquier proceso del software ágil se caracteriza por la forma en la que aborda cierto número de suposiciones clave [Fow02] acerca de la mayoría de proyectos de software:

1. Es difícil predecir qué requerimientos de software persistirán y cuáles cambiarán. También es difícil pronosticar cómo cambiarán las prioridades del cliente a medida que avanza el proyecto.
 2. Para muchos tipos de software, el diseño y la construcción están imbricados. Es decir, ambas actividades deben ejecutarse en forma simultánea, de modo que los modelos de diseño se prueban a medida que se crean. Es difícil predecir cuánto diseño se necesita antes de que se use la construcción para probar el diseño.
 3. El análisis, el diseño, la construcción y las pruebas no son tan predecibles como nos gustaría (desde un punto de vista de planeación).
- Dadas estas tres suposiciones, surge una pregunta importante: ¿cómo crear un proceso que pueda manejar lo *impredecible*? La respuesta, como ya se dijo, está en la adaptabilidad del proceso al cambio rápido del proyecto y a las condiciones técnicas. Por tanto, un proceso ágil debe ser *adaptable*.
- Però la adaptación continua logra muy poco si no hay avance. Entonces, un proceso de software ágil debe adaptarse *incrementalmente*. Para lograr la adaptación incremental, un equipo ágil requiere retroalimentación con el cliente (de modo que sea posible hacer las adaptaciones apropiadas). Un catalizador ético para la retroalimentación con el cliente es un prototipo operativo o una porción de un sistema operativo. Así, debe instituirse una *estrategia de desarrollo incremental*. Deben entregarse *incrementos de software* (prototipos ejecutables o porciones de un sistema operativo) en períodos cortos de tiempo, de modo que la adaptación vaya a ritmo con el cambio (impredecible). Este enfoque iterativo permite que el cliente evalúe en forma regular las adaptaciones del proceso que se realicen para aprovechar la retroalimentación.

3.3.1 Principios de agilidad

La Alianza Ágil (véase [Ag03]), [Fow01]) define 12 principios de agilidad para aquellos que la quieran alcanzar:

1. La prioridad más alta es satisfacer al cliente a través de la entrega pronta y continua de software valioso.
2. Son bienvenidos los requerimientos cambiantes, aun en una etapa avanzada del desarrollo. Los procesos ágiles dominan el cambio para provecho de la ventaja competitiva del cliente.
3. Entregar con frecuencia software que funcione, de dos semanas a un par de meses, de preferencia lo más pronto que se pueda.

4. Las personas de negocios y los desarrolladores deben trabajar juntos, a diario y durante todo el proyecto.



El software que funciona es importante, pero no olvida que también debe poseer unos atributos de calidad, como ser confiable, utilizable y susceptible de recibir mantenimiento.

5. Hay que desarrollar los proyectos con individuos motivados. Debe darse a éstos el ambiente y el apoyo que necesiten, y confiar en que harán el trabajo.

6. El método más eficiente y ético para transmitir información a los integrantes de un equipo de desarrollo, y entre éstos, es la conversación cara a cara.

7. La medida principal de avance es el software que funciona.

8. Los procesos ágiles promueven el desarrollo sostenible. Los patrocinadores, desarrolladores y usuarios deben poder mantener un ritmo constante en forma indefinida.

9. La atención continua a la excelencia técnica y el buen diseño mejora la agilidad.

10. Es esencial la simplicidad: el arte de maximizar la cantidad de trabajo no realizado.

11. Las mejores arquitecturas, requerimientos y diseños surgen de los equipos con organización propia.

12. El equipo reflexiona a intervalos regulares sobre cómo ser más eficaz, para después afinar y ajustar su comportamiento en consecuencia.

No todo modelo de proceso ágil aplica estos 12 principios con igual intensidad y algunos eligen ignorar (o al menos sostener) la importancia de uno o más de ellos. Sin embargo, los principios definen un *espíritu ágil* que se mantiene en cada uno de los modelos de proceso que se presentan en este capítulo.

3.3.2 La política del desarrollo ágil

Hay mucho debate (a veces estridente) sobre los beneficios y aplicabilidad del desarrollo de software ágil como oposición a los procesos más convencionales. Jim Highsmith [Hig02a] señala (en tono de burla) los extremos cuando caracteriza la posición del campo a favor de la agilidad (“agilistas”). “Los metodólogos tradicionales están atrapados en un pantano y producirán una documentación sin defectos en vez de un sistema funcional que satisfaga las necesidades del negocio.” Como contrapunto, plantea de nuevo como burla la posición del campo de la ingeniería de software tradicional: “Los metodólogos ligeros, perdón, ‘ágiles’; son un grupo de mendigos, famosos que se van a llevar una sorpresa cuando intenten convertir sus juguetes en software a la medida de la empresa.”

Como todos los argumentos sobre la tecnología de software, este debate sobre la metodología corre el riesgo de degenerar en una guerra religiosa. Si estalla, desparece el pensamiento racional y lo que guía la toma de decisiones son las creencias y no los hechos.

Nadie está contra la agilidad. La pregunta real es: ¿cuál es la mejor forma de lograrla? De igual importancia: ¿cómo construir software que satisfaga en el momento las necesidades de los clientes y que tenga características de calidad que permitan ampliarlo y escalarlo para que también las saltega en el largo plazo?

No hay respuestas absolutas a ninguna de estas preguntas. Aun dentro de la escuela ágil hay muchos modelos de proceso propuestos (véase la sección 3.4), cada uno con un enfoque algo diferente para el problema de la agilidad. Dentro de cada modelo hay un conjunto de “ídeas” (los agilistas las llaman “tareas del trabajo”) que representan un alejamiento significativo de la ingeniería de software tradicional. No obstante, muchos conceptos ágiles solo son adaptaciones de algunos que provienen de la buena ingeniería de software. En resumen: hay mucho por ganar si se considera lo mejor de ambas escuelas, y virtualmente no se gana nada si se denega cualquiera de los enfoques.

Si el lector está interesado, consulte [Hig01], [Hig02a] y [Dem02] para ver un resumen ameno de otros aspectos técnicos y políticos importantes.

3.3.3 Factores humanos

Los defensores del desarrollo de software ágil se tornan muchas molestias para enfatizar la importancia de los “factores personales”. Como dicen Cockburn y Highsmith [Coco1] al “El desarrollo ágil se centra en los talentos y habilidades de los individuos, y adapta el proceso a personas y equipos específicos.” El punto clave de esta afirmación es que *el proceso se adapta a las necesidades de las personas y del equipo, no al revés*.²

Si los miembros del equipo de software son los que van a generar las características del proceso que van a aplicarse a la elaboración de software, ente ellos debe existir cierto número de características clave, mismas que debe compartir el equipo ágil como tal:

Cita:
“Los métodos ágiles obtienen gran parte de su agilidad por basarse en el compromiso íntimo de incorporarlo en el equipo, más que en escribir el ‘conocimiento en plenos.’”

Barry Boehm

Competencia. En un contexto de desarrollo ágil (así como en la ingeniería de software), la “competencia” incluye el talento innato, las habilidades específicas relacionadas con el software y el conocimiento general del proceso que el equipo haya elegido aplicar. La habilidad y el conocimiento del proceso pueden y deben considerarse para todas las personas que sean miembros ágiles del equipo.

Enfoque común. Aunque los miembros del equipo ágil realicen diferentes tareas y aparten habilidades distintas al proyecto, todos deben centrarse en una meta: entregar al cliente en la fecha prometida un incremento de software que funcione. Para lograrlo, el equipo también se centrará en adaptaciones continuas (pequeñas y grandes) que hagan que el proceso se ajuste a las necesidades del equipo.

Colaboración. La ingeniería de software (sin importar el proceso) trata de evaluar, analizar y usar la información que se comunica al equipo de software; crear información que ayudará a todos los participantes a entender el trabajo del equipo; y generar información (software de cómputo y bases de datos relevantes) que aporten al cliente valor del negocio. Para efectuar estas tareas, los miembros del equipo deben colaborar, entre sí y con todos los participantes.

Habilidades para tomar decisiones. Cualquier equipo bueno de software (incluso los equipos ágiles) debe tener libertad para controlar su destino. Esto implica que se dé autonomía al equipo: autoridad para tomar decisiones sobre asuntos tanto técnicos como del proyecto.

Capacidad para resolver problemas difusos. Los gerentes de software deben reconocer que el equipo ágil tendrá que tratar en forma continua con la ambigüedad y que será sacudido de manera permanente por el cambio. En ciertos casos, el equipo debe aceptar el hecho de que el problema que resuelven ahora tal vez no sea el que se necesita resolver mañana. Sin embargo, las lecciones aprendidas de cualquier actividad relacionada con la solución de problemas (incluso aquellas que resuelven el problema equivocado) serán benéficas para el equipo en una etapa posterior del proyecto.

Confianza y respeto mutuos. El equipo ágil debe convertirse en lo que DeMarco y Lister [Dem98] llaman “pegado” (véase el capítulo 24). Un equipo pegado tiene la confianza y respeto que son necesarios para hacer “su tejido tan fuerte que el todo es más que la suma de sus partes” [Dem98].

Organización propia. En el contexto del desarrollo ágil, la organización propia implica tres cosas: 1) el equipo ágil se organiza a sí mismo para hacer el trabajo, 2) el equipo organiza el proceso que se adapte mejor a su ambiente local, 3) el equipo organiza la programación del trabajo a fin de que se logre del mejor modo posible la entrega del incremento que elijan.

² Las organizaciones exitosas de ingeniería de software reconocen esta realidad sin importar el modelo de proceso que elijan.

de software. La organización propia tiene cierto número de beneficios técnicos, pero, lo que es más importante, sirve para mejorar la colaboración y elevar la moral del equipo. En esencia, el equipo sirve como su propio gerente. Ken Schwaber [Sch02] aborda estos aspectos cuando escribe: "El equipo selecciona cuánto trabajo cree que puede realizar en cada iteración, y se compromete con la labor. Nada desmotiva tanto a un equipo como que alguien establezca compromisos por él. Nada motiva más a un equipo como aceptar la responsabilidad de cumplir los compromisos que haya hecho él mismo."

3.4 PROGRAMACIÓN EXTREMA (XP)

A fin de ilustrar un proceso ágil con más detalle, daremos un panorama de la *programación extrema* (XP), el enfoque más utilizado del desarrollo de software ágil. Aunque las primeras actividades con las ideas y los métodos asociados a XP ocurrieron al final de la década de 1980, el trabajo fundamental sobre la materia habría sido escrito por Kent Beck [Bec04a]. Una variante de XP llamada *XP industrial* [XPi] se propuso en una época más reciente [Ker05]. XP mejora la XP y tiene como objetivo el proceso ágil para ser usado específicamente en organizaciones grandes.

3.4.1 Valores XP

Beck [Bec04a] define un conjunto de cinco *valores* que establecen el fundamento para todo trabajo realizado como parte de XP: comunicación, simplicidad, retroalimentación, valentía y respeto. Cada uno de estos valores se usa como un motor para actividades, acciones y tareas específicas de XP.

A fin de lograr la *comunicación eficaz* entre los ingenieros de software y otros participantes (por ejemplo, para establecer las características y funciones requeridas para el software), XP pone el énfasis en la colaboración estrecha pero informal (verbal) entre los clientes y los desarrolladores, en el establecimiento de *metáforas*³ para comunicar conceptos importantes, en la retroalimentación continua y en evitar la documentación voluminosa como medio de comunicación.

Para alcanzar la *simplificación*, XP restringiendo a los desarrolladores para que diseñen sólo las necesidades inmediatas, en lugar de considerar las del futuro. El objetivo es: crear un diseño sencillo que se implemente con facilidad en forma de código. Si hay que mejorar el diseño, se rediseñará en un momento posterior.

La *retroalimentación* se obtiene de tres fuentes: el software implementado, el cliente y otros miembros del equipo de software. Al diseñar e implementar una estrategia de pruebas eficaz (capítulos 17 a 20), el software (por medio de los resultados de las pruebas) da retroalimentación al equipo ágil. XP usa la *prueba unitaria* como su táctica principal de pruebas. A medida que se desarrolla cada clase, el equipo implementa una prueba unitaria para ejecutar cada operación de acuerdo con su funcionalidad especificada. Cuando se entrega un incremento a un cliente, las *historias del usuario o casos de uso* (véase el capítulo 5) que se implementan con el incremento se utilizan como base para las pruebas de aceptación. El grado en el que el software implementa la salida, función y comportamiento del caso de uso es una forma de retroalimentación. Por último, conforme se obtienen nuevos requerimientos como parte de la planeación iterativa, el equipo da al cliente una retroalimentación rápida con miras al costo y al efecto en la programación de actividades.



Cita:
"XP es la respuesta a la pregunta: '¿Qué pequeño podemos hacer un gran software?'"
Anónimo

3.4.2 El proceso XP

La programación extrema usa un enfoque orientado a objetos (véase el apéndice 2) como paradigma preferido de desarrollo y engloba un conjunto de reglas y prácticas que ocurren en el contexto de cuatro actividades estructurales: planeación, diseño, codificación y pruebas. La figura 3.2 ilustra el proceso XP y resalta algunas de las ideas y tareas clave que se asocian con cada actividad estructural. En los párrafos que siguen se resumen las actividades de XP clave.

Planeación. La actividad de planeación (también llamada *juego de planeación*) comienza escurriendo —actividad para recabar requerimientos que permite que los miembros técnicos del equipo XP entiendan el contexto del negocio para el software y adquieran la sensibilidad de la salida y características principales que se requieren—. Escuchar lleva a la creación de algunas "historias" (también llamadas *historias del usuario*), que describen la salida necesaria, características y funcionalidad del software que se va a elaborar. Cada *historia* (similar a los casos de uso descritos en el capítulo 5) es escrita por el cliente y colocada en una tarjeta indexada. El cliente asigna un *valor* (es decir, una prioridad) a la historia con base en el valor general de la característica o función para el negocio.⁴ Después, los miembros del equipo XP

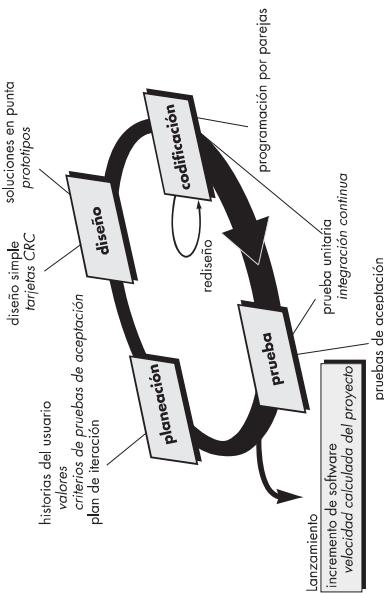


? **¿Qué es una "historia" XP?**



CONSEJO
Manejalo sensible, porque el diseño continuo consume mucho tiempo y recursos.

FIGURA 3.2
El proceso de la programación extrema



³ En el contexto de XP, una *metáfora* es "una historia que cada quien—clientes, programadores y gerentes—narrá, acerca de cómo funciona el sistema" [Bec04a].

⁴ El rediseño permite que un ingeniero mejore la estructura interna de un diseño (o código fuente) sin cambiar su funcionalidad o comportamiento externos. En esencia, el rediseño puede utilizarse para mejorar la eficiencia, disponibilidad o rendimiento de un diseño o del código que lo implementa.

⁵ El valor de una historia también puede depender de la presencia de otra historia.

evalúan cada historia y le asignan un costo, medido en semanas de desarrollo. Si se estima que la historia requiere más de tres semanas de desarrollo, se pide al cliente que la descomponga en historias más chicas y de nuevo se asigna un valor y costo. Es importante observar que en cualquier momento es posible escribir nuevas historias.

Los clientes y desarrolladores trabajan juntos para decidir como agrupar las historias en la siguiente entrega (el siguiente incremento de software) que desarrollará el equipo XP. Una vez que se llega a un compromiso sobre la entrega (acuerdo sobre las historias por incluir, la fecha de entrega y otros aspectos del proyecto), el equipo XP ordena las historias que serán desarrolladas en una de tres formas: 1) todas las historias se implementarán de inmediato (en pocas semanas), 2) las historias con más valor entrarán a la programación de actividades y se implementarán en primer lugar o 3) las historias más riesgosas formarán parte de la programación de actividades y se implementarán primero.

Después de la primera entrega del proyecto (también llamada incremento de software), el equipo XP calcula la velocidad de éste. En pocas palabras, la *velocidad del proyecto* es el número de historias de los clientes implementadas durante la primera entrega. La velocidad del proyecto se usa para: 1) ayudar a estimar las fechas de entrega y programar las actividades para las entregas posteriores, y 2) determinar si se ha hecho un gran compromiso para todas las historias durante todo el desarrollo del proyecto. Si esto ocurre, se modifica el contenido de las entregas o se cambian las fechas de entrega final.

A medida que avanza el trabajo, el cliente puede agregar historias, cambiar el valor de una ya existente, descomponerlas o eliminarlas. Entonces, el equipo XP reconsidera todas las entregas faltantes y modifica sus planes en consecuencia.

Diseño. El diseño XP sigue rigurosamente el principio MS (mantengo sencillo). Un diseño sencillo siempre se prefiere sobre una representación más compleja. Además, el diseño guía la implementación de una historia conforme se escribe: nada más y nada menos. Se desalienta el diseño de funcionalidad adicional porque el desarrollador supone que se requerirá después.⁶

XP estimula el uso de las tarjetas CRC (véase el capítulo 7) como un mecanismo eficaz para pensar en el software en un contexto orientado a objetos. Las tarjetas CRC (clase-responsabilidad-colaborador) identifican y organizan las clases orientadas a objetos⁷ que son relevantes para el incremento actual de software. El equipo XP dirige el ejercicio de diseño con el uso de un proceso similar al que se describe en el capítulo 8. Las tarjetas CRC son el único producto del trabajo de diseño que se genera como parte del proceso XP.

Si en el diseño de una historia se encuentra un problema de diseño difícil, XP recomienda la creación inmediata de un prototipo operativo de esa porción del diseño. Entonces, se implementa y evalúa el prototipo del diseño, llamado *solución en punta*. El objetivo es disminuir el riesgo cuando comience la implementación verdadera y validar las estimaciones originales para la historia que contiene el problema de diseño.

En la sección anterior se dijo que XP estimula el *rediseño*, técnica de construcción que también es un método para la optimización del diseño. Fowler [Fow00] describe el rediseño del modo siguiente:

Rediseño es el proceso mediante el cual se cambia un sistema de software en forma tal que no altere el comportamiento externo del código, pero si mejore la estructura interna. Es una manera disciplinada de limpiar el código (y modificar o simplificar el diseño interno) que minimiza la probabilidad de introducir errores. En esencia, cuando se rediseña, se mejora el diseño del código después de haber sido escrito.

Como el diseño XP virtualmente no utiliza notación y genera pocos, si alguno, productos del trabajo que no sean tarjetas CRC y soluciones en punta, el diseño es visto como un artefacto en transición que puede y debe modificarse continuamente a medida que avanza la construcción. El objetivo del rediseño es controlar dichas modificaciones, sugiriendo pequeños cambios en el diseño que "son capaces de mejorarlo en forma radical" [Fow00]. Sin embargo, debe notarse que el esfuerzo que requiere el rediseño aumenta en forma notable con el tamaño de la aplicación.

Un concepto central en XP es que el diseño ocurre tanto antes como *después* de que comienza la codificación. Rediseñar significa que el diseño se hace de manera continua conforme se construye el sistema. En realidad, la actividad de construcción en sí misma dará al equipo XP una guía para mejorar el diseño.

Codificación. Después de que las historias han sido desarrolladas y de que se ha hecho el trabajo de diseño preliminar, el equipo *no* inicia la codificación, sino que desarrolla una serie de pruebas unitarias a cada una de las historias que se van a incluir en la entrega en curso (incremento de software).⁸ Una vez creada la "prueba unitaria", el desarrollador está mejor capacitado para centrarse en lo que debe implementarse para pasar la prueba. No se agrega nada extraño (MS). Una vez que el código está terminado, se le aplica de inmediato una prueba unitaria, con lo que se obtiene retroalimentación instantánea para los desarrolladores.

Un concepto clave durante la actividad de codificación (y uno de los aspectos del que más se habla en la XP) es la *programación por parejas*. XP recomienda que dos personas trabajen juntas en una estación de trabajo con el objeto de crear código para una historia. Esto da un mecenazgo para la solución de problemas en tiempo real (es frecuente que dos cabezas piensen más que una) y para el aseguramiento de la calidad también en tiempo real (el código se revisa conforme se crea). También mantiene a los desarrolladores centrados en el problema de que se trate. En la práctica, cada persona adopta un papel un poco diferente. Por ejemplo, una de ellas tal vez piense en los detalles del código de una porción particular del diseño, mientras la otra se asegura de que se siguen los estándares de codificación (parte necesaria de XP) o de que el código para la historia satisfará la prueba unitaria desarrollada a fin de validar el código concerniendo con la historia.

A medida que las parejas de programadores terminan su trabajo, el código que desarrollan se integra con el trabajo de los demás. En ciertos casos, esto lo lleva a cabo a diario un equipo de integración. En otros, las parejas de programadores tienen la responsabilidad de la integración. Esta estrategia de "integración continua" ayuda a evitar los problemas de compatibilidad e interfaz y brinda un ambiente de "prueba de humo" (véase el capítulo 17) que ayuda a descubrir a tiempo los errores.

Pruebas. Ya se dijo que la creación de pruebas unitarias antes de que comience la codificación es un elemento clave del enfoque de XP. Las pruebas unitarias que se crean deben implementarse con el uso de una estructura que permita automatizarlas (de modo que puedan ejecutarse en repetidas veces y con facilidad). Esto estimula una estrategia de pruebas de regresión (véase el capítulo 17) siempre que se modifique el código (lo que ocurre con frecuencia, dada la filosofía del rediseño en XP).

A medida que se organizan las pruebas unitarias individuales en un "grupo de prueba universitario" [Weis99], las pruebas de la integración y validación del sistema pueden efectuarse a diario. Esto da al equipo XP una indicación continua del avance y también lanza señales de alerta si las

⁸ Este enfoque es equivalente a saber las preguntas del examen antes de comenzar a estudiar. Vuelve mucho más fácil el estudio porque centra la atención sólo en las preguntas que se van a responder.

⁹ La prueba unitaria, que se estudia en detalle en el capítulo 17, se centra en un componente de software individual sobre interfaz, estructuras de datos y funcionalidad del componente, en un esfuerzo por descubrir errores locales del componente.



CLAVE
El diseño mejor la estructura interna de un diseño (o código fuente) sin cambiar su funcionalidad o comportamiento externo.



WebRef
En la dirección 2.com/tg/ se halla un **wikiPlanningGame** que incluye un "juego de planeación" XP provechoso.

Hay información útil sobre de XP en el sitio www.xprogramming.com.

¿Qué es la programación por parejas?



CONSEJO

Muchas parejas de software están llenas de individualistas. Si la programación por parejas no funciona con eficacia, tendrás que trabajar para cambiar esa cultura.



¿Cómo se usan las pruebas unitarias en XP?



CONSEJO

XP destaca la importancia del diseño, opinión con la que no todos están de acuerdo. En realidad, hay veces en las que debe hacerse énfasis en el diseño.



WebRef

En la dirección www.electrónica.com se encuentran artículos y herramientas de diseño.

Rediseño es el proceso mediante el cual se cambia un sistema de software en forma tal que no altere el comportamiento externo del código, pero si mejore la estructura interna. Es una manera disciplinada de limpiar el código (y modificar o simplificar el diseño interno) que minimiza la probabilidad de introducir errores. En esencia, cuando se rediseña, se mejora el diseño del código después de haber sido escrito.

⁶ Estos lineamientos de diseño deben seguirse en todo método de ingeniería de software, aunque hay ocasiones en los que la notación y terminología sofisticadas del diseño son un camino hacia la simplicidad.

⁷ Las clases orientadas a objetos se estudian en el apéndice 2, en el capítulo 8 y en toda la parte 2 de este libro.

cosas marchan mal. Wells [Wel99] dice: "Corregir pequeños problemas cada cierto número de horas toma menos tiempo que resolver problemas enormes justo antes del plazo final."

Las pruebas de aceptación XP. también llamadas *pruebas del cliente*, son especificadas por el cliente y se centran en las características y funcionalidad generales del sistema que son visibles y revisables por parte del cliente. Las pruebas de aceptación se derivan de las historias de los usuarios que se han implementado como parte de la liberación del software.

3.4.3 XP industrial

Joshua Kerievsky [Ker05] describe la *programación extrema industrial* [XP] por sus siglas en inglés] en la forma siguiente: "XP es la evolución orgánica de XP. Está imbuida del espíritu minimalista, centrado en el cliente y orientado a las pruebas que tiene XP. XP difiere sobre todo de la XP original en su mayor inclusión de la gerencia, el papel más amplio de los clientes y en sus prácticas técnicas actualizadas". XP incorpora seis prácticas nuevas diseñadas para ayudar a garantizar que un proyecto XP funciona con éxito para proyectos significativos dentro de una organización grande.

Evaluación de la factibilidad. Antes de iniciar un proyecto XP, la organización debe efectuar una evaluación de la *factibilidad*. Esta deja en claro si: 1) existe un ambiente apropiado de desarrollo que acepte XP, 2) el equipo estará constituido por los participantes adecuados, 3) la organización tiene un programa de calidad distintivo y apoya la mejora continua, 4) la cultura organizacional apoyará los nuevos valores de un equipo ágil, y 5) la comunidad extendida del proyecto estará constituida de modo apropiado.

Comunidad del proyecto. La XP clásica sugiere que se utilice personal apropiado para formar el equipo ágil a fin de asegurar el éxito. La implicación es que las personas en el equipo deben estar bien capacitadas, ser adaptables y hábiles, y tener el temperamento apropiado para contribuir al equipo con organización propia. Cuando se aplica XP a un proyecto significativo en una organización grande, el concepto de "equipo" debe adoptar la forma de *comunidad*. Una comunidad puede tener un tecnólogo y clientes que son fundamentales para el éxito del proyecto, así como muchos otros participantes (equipo jurídico; auditores de calidad, de tipos de manufactura o de ventas, etc.) que "con frecuencia se encuentran en la periferia en un proyecto XP, pero que desempeñan en éste papeles importantes" [Ker05]. En XP, los miembros de la comunidad y sus papeles deben definirse de modo explícito, así como establecer los mecanismos para la comunicación y coordinación entre los integrantes de la comunidad.

Calificación del proyecto. El equipo de XP evalúa el proyecto para determinar si existe una justificación apropiada de negocios y si el proyecto cumplirá las metas y objetivos generales de la organización. La calificación también analiza el contexto del proyecto a fin de determinar cómo complementa, extiende o reemplaza sistemas o procesos existentes.

Administración orientada a pruebas. Un proyecto XP requiere criterios medibles para evaluar el estado del proyecto y el avance realizado. La administración orientada a pruebas establece una serie de "destinos" medibles [Ker05] y luego define los mecanismos para determinar si se han alcanzado o no éstos.

Retrospectivas. Después de entregar un incremento de software, el equipo XP realiza una revisión técnica especializada que se llama *retrospectiva* y que examina "los temas, eventos y lecciones aprendidas" [Ker05] a lo largo del incremento de software y/o de la liberación de todo el software. El objetivo es mejorar el proceso XP.

Aprendizaje continuo. Como el aprendizaje es una parte vital del proceso de mejora continua, los miembros del equipo XP son invitados (y tal vez incentivados) a aprender nuevos métodos y técnicas que conduzcan a una calidad más alta del producto.



Los puntos de orientación se derivan de las historias de los usuarios.

¿Qué nuevas prácticas se agregan a XP para crear XP?

Cita:

"Habilidad es **tú** que eres capaz de hacer. La motivación determina **lo que haces**. La **actitud** determina **cómo bien lo haces.**"
Lou Holtz

Además de las seis nuevas prácticas analizadas, XP modifica algunas de las existentes en XP. El *desarrollo impulsado por la historia* (DID) insiste en que las historias de las pruebas de aceptación se escriban antes de generar una sola línea de código. El *diseño impulsado por el dominio* (DID) es una mejora sobre el concepto de la "metáfora del sistema" usado en XP. El DID [Eva03] sugiere la creación evolutiva de un modelo de dominio que "represente con exactitud cómo pienzan los expertos del dominio en su maternidad" [Ker05]. La *formación de parejas* amplía el concepto de programación en pareja para que incluya a los gerentes y a otros participantes. El objetivo es mejorar la manera de compartir conocimientos entre los integrantes del equipo XP que no estén directamente involucrados en el desarrollo técnico. La *usabilidad iterativa* desalienta el diseño de una interfaz cargada al frente y estimula un diseño que evoluciona a medida que se liberan los incrementos de software y que se estudia la interacción de los usuarios con el software. La XP hace modificaciones más pequeñas a otras prácticas XP y redefine ciertos roles y responsabilidades para hacerlos más asequibles a proyectos significativos de las organizaciones grandes. Para mayores detalles de XP visite el sitio <http://industrixa.org>.

3.4.4 El debate XP

Los nuevos modelos y métodos de proceso han motivado análisis provechosos y en ciertas instancias debates acalorados. La programación extrema desencadena ambos. En un libro interesante que examina la eficacia de XP, Stephens y Rosenberg [Ste03] afirman que muchas prácticas de XP son beneficiosas, pero que otras están sobreestimadas y unas más son problemáticas. Los autores sugieren que la naturaleza codependiente de las prácticas de XP constituye tanto su fortaleza como su debilidad. Debido a que muchas organizaciones adoptan sólo un subconjunto de prácticas XP debilitan la eficacia de todo el proceso. Los defensores contradicen esto al afirmar que la XP está en evolución continua y que muchas de las críticas que se le hacen han llevado a correcciones conforme maduran sus prácticas. Entre los aspectos que destacan algunos críticos de la XP están los siguientes:¹⁰

- **Volatilidad de los requerimientos.** Como el cliente es un miembro activo del equipo XP, los cambios a los requerimientos se solicitan de manera informal. En consecuencia, el alcance del proyecto cambia y el trabajo inicial tiene que modificarse para dar acomodo a las nuevas necesidades. Los defensores afirman que esto pasa sin importar el proceso que se aplique y que la XP proporciona mecanismos para controlar los vaivenes del alcance.
- **Necesidades conflictivas del cliente.** Muchos proyectos tienen clientes múltiples, cada uno con sus propias necesidades. En XP, el equipo mismo tiene la tarea de asimilar las necesidades de distintos clientes, trabajo que tal vez está más allá del alcance de su autoridad.
- **Los requerimientos se expresan informalmente.** Las historias de usuario y las pruebas de aceptación son la única manifestación explícita de los requerimientos en XP. Los críticos afirman que es frecuente que se necesite un modelo o especificación más formal para garantizar que se descubran las omisiones, inconsistencias y errores antes de que se construya el sistema. Los defensores contratacan diciendo que la naturaleza cambiante de los requerimientos vuelve obsoletos esos modelos y especificaciones casi tan pronto como se desarrollan.
- **Falta de un diseño formal.** XP desalienta la necesidad del diseño de la arquitectura y, en muchas instancias, sugiere que el diseño de todas las clases debe ser relativamente informal. Los críticos argumentan que cuando se construyen sistemas complejos, debe ponerse el énfasis en el diseño con el objeto de garantizar que la estructura general del software tendrá calidad y que será susceptible de recibir mantenimiento. Los defensores

10 Para un estudio más detallado de ciertas críticas profundas hechas a XP visite www.softwareareality.com/ExtremeProgramming.jsp.

de XP sugieren que la naturaleza incremental del proceso XP limita la complejidad (la sencillez es un valor fundamental), lo que reduce la necesidad de un diseño extenso.

El lector debe observar que todo proceso del software tiene sus desventajas, y que muchas organizaciones de software, han utilizado con éxito la XP. La clave es identificar donde tiene sus debilidades un proceso y adaptarlo a las necesidades de la organización.

3.5 OTROS MODELOS ÁGILES DE PROCESO



Cita:

"Nuestra profesión pasa por las metodologías como un chico de 14 años pisa por la roja."

Stephen Hawrysh y Jim Ruprecht

La historia de la ingeniería de software está salpicada de descripciones y metodologías de proceso, métodos de modelado y notaciones, herramientas y tecnología, todos ellos obsoletos. Cada uno tuvo notoriedad y luego fue eclipsado por algo nuevo y (supuestamente) mejor. Con la introducción de una amplia variedad de modelos ágiles del proceso –cada uno en lucha por la aceptación de la comunidad de desarrollo de software— el movimiento ágil está siguiendo la misma ruta histórica.¹¹

siguiendo la misma ruta histórica.¹¹

CASASEGURA



Consideración del desarrollo ágil de software

Doug (afirmando con la cabeza): Ellos son uno de los partidarios, ¿o no?

Jamie: Sí.... Pedirán cambios cada cinco minutos.

Vinod: No necesariamente. Mi amigo dijo que hay formas de "adoptar" los cambios durante un proyecto de XP.

Doug: Entonces, chicos, ¿piensan que debemos usar XP?

Jamie: Definitivamente, sería bueno considerarlo.

Doug: Estoy de acuerdo. E incluido si elegimos un modelo incremental como nuestro enfoque, no hay razón para no incorporar mucho de lo que XP tiene que ofrecer.

Vinod: Doug, díselo hace un rato "cosas buenas y malas". ¿Cuáles son las malas?

Doug: Lo que no me gusta es la forma en la que XP desprecia el análisis y al diseño... digo así como decir que la escritura del código está donde hay acción...

(los miembros del equipo se miran entre sí y sonríen.)

Doug (habla por ambos): Escribir códigos es lo que hacemos, jefes!

Doug (réplica): Es cierto, pero me gustaría ver que dediquen un poco menos de tiempo a escribir código y luego a reescribirlo, y que pasen algo más de tiempo en el análisis de lo que debe hacerse para diseñar una solución que funcione.

Vinod: Tal vez tengamos las dos cosas, agilidad con un poco de disciplina.

Doug: Creo que podemos, Vinod. En realidad, estoy seguro de que se puede.

Jamie: ¿Qué? ¿Quieren decir que mercadotecnía trabajará con nosotros en el proyecto?

Como se dijo en la sección anterior, el más usado de todos los modelos ágiles de proceso es la programación extrema (XP). Pero se han propuesto muchos otros y están en uso en toda la industria. Entre ellos se encuentran los siguientes:

- Desarrollo adaptativo de software (DAS)
- Scrum
- Método de desarrollo de sistemas dinámicos (MDSD)
- Cristal
- Desarrollo impulsado por las características (DIC)
- Desarrollo esbelto de software (DES)
- Modelado ágil (MA)
- Proceso unificado ágil (PUA)

En las secciones que siguen se presenta un panorama muy breve de cada uno de estos modelos ágiles del proceso. Es importante notar que todos los modelos de proceso ágil se apegan (en mayor o menor grado) al *Manifiesto para el desarrollo ágil de software* y a los principios descritos en la sección 3.3.1. Para mayores detalles, consulte las referencias mencionadas en cada sección o ingrese en la entrada "desarrollo de software ágil" de Wikipedia.¹²

3.5.1 Desarrollo adaptativo de software (DAS)

El *desarrollo adaptativo de software* (DAS) fue propuesto por Jim Highsmith (High00) como una técnica para elaborar software y sistemas complejos. Los fundamentos filosóficos del DAS se centran en la colaboración humana y en la colaboración propia del equipo. Highsmith argumenta que un enfoque de desarrollo adaptativo basado en la colaboración es "tanto una fuente de orden en nuestras complejas interacciones, como de disciplina e ingeniería". Él define un "ciclo de vida" del DAS (véase la figura 3.3) que incorpora tres fases: especulación, colaboración y aprendizaje.

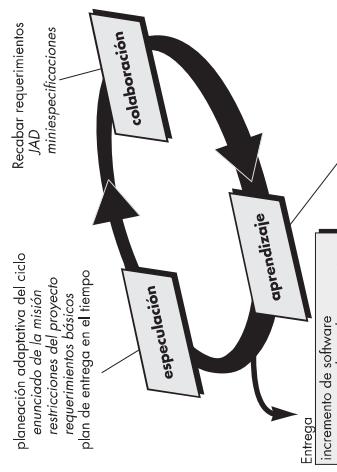


WebRef

En la dirección www.adaptivesys.com tienen más sobre el DAS.

Figura 3.3

Desarrollo adaptativo de software



¹¹ Esto no es algo malo. Antes de que uno o varios modelos se acepten como el estándar *de facto*, todos deben luchar por conquistar las mentes y corazones de los ingenieros de software. Los "ganadores" evolucionan hacia las mejores prácticas, mientras que los "perdedores" desaparecen o se funden con los modelos ganadores.

¹² Consulte http://en.wikipedia.org/wiki/Agile_software_development#Agile_methods.

Durante la especulación, se inicia el proyecto y se lleva a cabo la *planeación adaptativa del ciclo*. La especulación emplea la información de inicio del proyecto—enunciado de misión de los clientes, restricciones del proyecto (por ejemplo, fechas de entrega o descripciones de lista-rio) y requerimientos básicos— para definir el conjunto de ciclos de entrega (incrementos de software) que se requerirán para el proyecto.

No importa lo completo y previso que sea el plan del ciclo, será inevitable que cambie. Con base en la información obtenida al terminar el primer ciclo, el plan se revisa y se ajusta, de modo que el trabajo planeado se acomode mejor a la realidad en la que trabaja el equipo DAS.

Las personas motivadas usan la *colaboración* de manera que multiplica su talento y producción creativa más allá de sus números absolutos. Este enfoque es un tema recurrente en todos los métodos ágiles. Sin embargo, la colaboración no es fácil. Incluye la comunicación y el trabajo en equipo, pero también resalta el individualismo porque la creatividad individual desempeña un papel importante en el pensamiento colaborativo. Es cuestión, sobre todo, de confianza. Las personas que trabajan juntas deben confiar una en otra a fin de: 1) criticarse sin enojo, 2) ayu-darse sin resentimiento, 3) trabajar tan duro, o más, que como de costumbre, 4) tener el conjunto de aptitudes para contribuir al trabajo, y 5) comunicar los problemas o preocupaciones de manera que conduzcan a la acción efectiva.

Conforme los miembros de un equipo DAS comienzan a desarrollar los componentes que forman parte de un ciclo adaptativo, el énfasis se traslada al ‘aprendizaje’ de todo lo que hay en el avance hacia la terminación del ciclo. En realidad, Highsmith [Hig00] afirma que los desarrolladores de software sobreestiman con frecuencia su propia comprensión (de la tecnología, del proceso y del proyecto) y que el aprendizaje los ayudará a mejorar su nivel de entendimiento real. Los equipos DAS aprenden de tres maneras: grupos de enfoque (véase el capítulo 5), revisiones técnicas (véase el capítulo 14) y análisis post mortem del proyecto.

La filosofía DAS tiene un mérito, sin importar el modelo de proceso que se use. El énfasis general que hace el DAS en la dinámica de los equipos con organización propia, la colaboración interpersonal y el aprendizaje individual y del equipo generan equipos para proyectos de software que tienen una probabilidad de éxito mucho mayor.

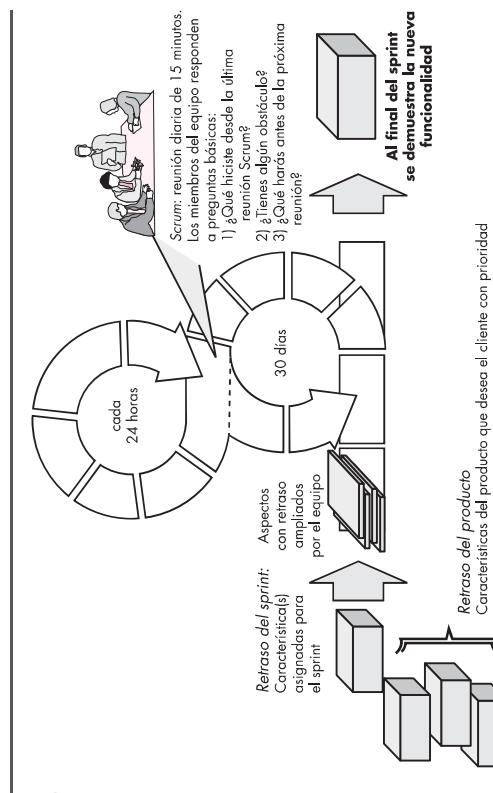
3.5.2 Scrum

Scrum (nombre que proviene de cierta jugada que tiene lugar durante un partido de rugby)¹³ es un método de desarrollo ágil de software concebido por Jeff Sutherland y su equipo de desarrollo a principios de la década de 1990. En años recientes, Schwaber y Beedle [Sch01] han desarrollado más los métodos Scrum.

Los principios Scrum son congruentes con el manifiesto ágil y se utilizan para guiar actividades de desarrollo dentro de un proceso de análisis que incorpora las siguientes actividades estructurales: requerimientos, análisis, diseño, evolución y entrega. Dentro de cada actividad estructural, las tareas del trabajo ocurren con un patrón del proceso (que se estudia en el párrafo siguiente) llamado *sprint*. El trabajo realizado dentro de un sprint (el número de éstos que requiere cada actividad estructural varía en función de la complejidad y tamaño del producto) se adapta al problema en cuestión y se define —y con frecuencia se modifica— en tiempo real por parte del equipo Scrum. El flujo general del proceso Scrum se ilustra en la figura 3.4.

Scrum acentúa el uso de un conjunto de patrones de proceso del software [Noy02] que han demostrado ser eficaces para proyectos con plazos de entrega muy apretados, requerimientos cambiantes y negocios críticos. Cada uno de estos patrones de proceso define un grupo de acciones de desarrollo:

Figura 3.4
Flujo del proceso Scrum



CONSEJO
La colaboración eficaz con el cliente sólo ocurre si entra cualquier actitud del tipo “nosotros y ellos”.

El DAS pone al énfasis en el aprendizaje como elemento clave para lograr un equipo con “organización propia”.
CLAVE

El DAS tiene un mérito, sin importar el modelo de proceso que se use. El énfasis general que hace el DAS en la dinámica de los equipos con organización propia, la colaboración interpersonal y el aprendizaje individual y del equipo generan equipos para proyectos de software que tienen una probabilidad de éxito mucho mayor.

3.5.2 Scrum

Scrum (nombre que proviene de cierta jugada que tiene lugar durante un partido de rugby)¹³ es un método de desarrollo ágil de software concebido por Jeff Sutherland y su equipo de desarrollo a principios de la década de 1990. En años recientes, Schwaber y Beedle [Sch01] han desarrollado más los métodos Scrum.

Los principios Scrum son congruentes con el manifiesto ágil y se utilizan para guiar actividades de desarrollo dentro de un proceso de análisis que incorpora las siguientes actividades estructurales: requerimientos, análisis, diseño, evolución y entrega. Dentro de cada actividad estructural, las tareas del trabajo ocurren con un patrón del proceso (que se estudia en el párrafo siguiente) llamado *sprint*. El trabajo realizado dentro de un sprint (el número de éstos que requiere cada actividad estructural varía en función de la complejidad y tamaño del producto) se adapta al problema en cuestión y se define —y con frecuencia se modifica— en tiempo real por parte del equipo Scrum. El flujo general del proceso Scrum se ilustra en la figura 3.4.

Scrum acentúa el uso de un conjunto de patrones de proceso del software [Noy02] que han demostrado ser eficaces para proyectos con plazos de entrega muy apretados, requerimientos cambiantes y negocios críticos. Cada uno de estos patrones de proceso define un grupo de acciones de desarrollo:



WebRef
En dirección www.controlthis.com hay información útil sobre Scrum.



CLAVE
Scrum incluye un conjunto de patrones del proceso que ponen el énfasis en las principales del proyecto, las unidades de trabajo agrupadas, la comunicación y la retroalimentación frecuente con el cliente.

¹³ Se forma un grupo de jugadores alrededor del balón y todos trabajan juntos (a veces con violencia) para moverlo a través del campo.

¹⁴ Una *caja de tiempo* es un término de la administración de proyectos (véase la parte 4 de este libro) que indica el tiempo que se ha asignado para cumplir alguna tarea.

Es importante notar que las demostraciones preliminares no contienen toda la funcionalidad planeada, sino que éstas se entregarán dentro de la caja de tiempo establecida.

Beedle y sus colegas [Bee99] presentan un análisis exhaustivo de estos patrones en el que dicen: "Scrum supone de entrada la existencia de casos...". Los patrones de proceso Scrum permiten que un equipo de software trabaje con éxito en un mundo en el que es imposible eliminar la incertidumbre.

3.5.3 Método de desarrollo de sistemas dinámicos (MDSD)



WebRef
En la dirección www.dsdm.org/ hay recursos útiles para el MDSD.

El *método de desarrollo de sistemas dinámicos* (MDSD) [Sta97] es un enfoque de desarrollo ágil de software que "proporciona una estructura para construir y dar mantenimiento a sistemas que cumplen restricciones apretadas de tiempo mediante la realización de prototipos incrementales en un ambiente controlado de proyectos" [CCS02]. La filosofía MDSD está tomada de una versión modificada de la regla de Pareto: 80 por ciento de una aplicación puede entregarse en 20 por ciento del tiempo que tomaría entregarla completa (100 por ciento).

El MDSD es un proceso iterativo de software en el que cada iteración sigue la regla de 80 por ciento. Es decir, se requiere solo suficiente trabajo para cada incremento con objeto de facilitar el paso al siguiente. Los detalles restantes se terminan más tarde, cuando se conocen los requerimientos del negocio y se han pedido y efectuado cambios.

El grupo PSDM Consortium (www.dsdm.org) es un conglomerado mundial de compañías que adoptan colectivamente el papel de "custodios" del método. El consorcio ha definido un modelo de proceso ágil, llamado *círculo de vida MDSD*, que define tres ciclos iterativos distintos, precedidos de dos actividades adicionales al ciclo de vida:

Estudio de factibilidad: establece los requerimientos y restricciones básicas del negocio, así como con la aplicación que se va a construir, para luego evaluar si la aplicación es un candidato viable para aplicarle el proceso MDSD.

Estudio del negocio: establece los requerimientos e información funcionales que permitirán la aplicación para dar valor al negocio; asimismo, define la arquitectura básica de la aplicación e identifica los requerimientos para darle mantenimiento.

Iteración del modelo funcional: produce un conjunto de prototipos incrementales que demuestran al cliente la funcionalidad. (Nota: todos los prototipos de MDSD están pensados para que evolucionen hacia la aplicación que se entrega). El objetivo de este ciclo iterativo es recabar requerimientos adicionales por medio de la obtención de retroalimentación de los usuarios cuando practican con el prototipo.

Diseño e iteración de la construcción: revisita los prototipos construidos durante la *iteración del modelo funcional* a fin de garantizar que en cada iteración se ha hecho ingeniería en forma que permita dar valor operativo del negocio a los usuarios finales; la *iteración del modelo funcional* y el diseño e iteración de la construcción ocurren de manera concurrente.

Implementación: coloca el incremento más reciente del software (un prototipo "operacional") en el ambiente de operación. Debe notarse que: 1) el incremento tal vez no sea el de 100% final, o 2) quizás se pidan cambios cuando el incremento se ponga en su lugar. En cualquier caso, el trabajo de desarrollo MDSD continúa y vuelve a la actividad de iteración del modelo funcional.

El MDSD se combina con XP (véase la sección 3.4) para dar un enfoque de combinación que define un modelo sólido del proceso (ciclo de vida MDSD) con las prácticas detalladas (XP) que se requieren para elaborar incrementos de software. Además, los conceptos DAS se adaptan a un modelo combinado del proceso.

3.5.4 Cristal

Alistair Cockburn [Coc05] creó la *familia Cristal de métodos ágiles*¹⁵ a fin de obtener un enfoque de desarrollo de software que premia la "manejabilidad" durante lo que Cockburn caracteriza como "un juego cooperativo con recursos limitados, de invención y comunicación, con el objetivo primario de entregar software útil que funcione y con la meta secundaria de plantear el siguiente juego" [Coc02].

Para lograr la manejabilidad, Cockburn y Highsmith definieron un conjunto de metodologías, cada una con elementos fundamentales comunes a todos y roles, patrones de proceso, producto del trabajo y prácticas que son únicas para cada uno. La familia Cristal en realidad es un conjunto de ejemplos de procesos ágiles que han demostrado ser efectivos para diferentes tipos de proyectos. El objetivo es permitir que equipos ágiles seleccionen al miembro de la familia Cristal más apropiado para su proyecto y ambiente.

3.5.5 Desarrollo impulsado por las características (DIC)

El *desarrollo impulsado por las características* (DIC) lo concibió originalmente Peter Coad y sus colegas [Coa99] como modelo práctico de proceso para la ingeniería de software orientada a objetos. Stephen Palmer y John Feilung [Pal02] ampliaron y mejoraron el trabajo de Coad con la descripción de un proceso adaptativo y ágil aplicable a proyectos de software de tamaño moderado y grande.

Igual que otros proyectos ágiles, DIC adopta una filosofía que: 1) pone el énfasis en la colaboración entre los integrantes de un equipo DIC; 2) administra la complejidad de los problemas y del proyecto con el uso de la descomposición basada en las características, seguida de la integración de incrementos de software, 3) comunica los detalles técnicos en forma verbal, gráfica y con medios basados en texto. El DIC pone el énfasis en las actividades de aseguramiento de la calidad del software mediante el estimulo de la estrategia de desarrollo incremental, el uso de inspecciones del diseño y del código, la aplicación de auditorias de aseguramiento de la calidad del software (véase el capítulo 16), el conjunto de mediciones y el uso de patrones (para el análisis, diseño y construcción).

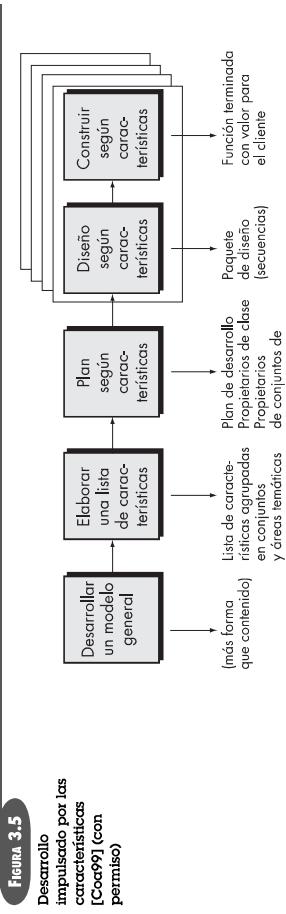
En el contexto del DIC, una *característica* "es una función valiosa para el cliente que puede implementarse en dos semanas o menos" [Coa99]. El énfasis en la definición de características proporciona los beneficios siguientes:

- Debido a que las características son bloques pequeños de funcionalidad que se entregan, los usuarios las describen con más facilidad, entienden cómo se relacionan entre sí y las revisan mejor en busca de ambigüedades, errores u omisiones.
- Las características se organizan por jerarquía de grupos relacionados con el negocio.
- Como una característica es el incremento de software DIC que se entrega, el equipo desarrolla características operativas cada dos semanas.
- El diseño y representación en código de las características son más fáciles de inspeccionar con eficacia porque éstas son pequeñas.
- La planificación, programación de actividades y seguimiento son determinadas por la jerarquía de características, y no por un conjunto de tareas de ingeniería de software adoptadas en forma arbitraria.

Coad y sus colegas [Coa99] sugieren el esquema siguiente para definir una característica:

<acción> el <resultado> <algoritmo> un <objeto>

¹⁵ El nombre "cristal" se deriva de los cristales de minerales, cada uno de los cuales tiene propiedades específicas de color, forma y dureza.



donde **<objeto>** es “una persona, lugar o cosa (incluso roles, momentos del tiempo o intervalos temporales, o descripciones parecidas a las entradas de un catálogo)”. Algunos ejemplos de características para una aplicación de comercio electrónico son los siguientes:

Agregar el producto al catálogo de compras

Mostrar las especificaciones técnicas del producto

Guardar la información de envío para el cliente

Un conjunto de características agrupa las que son similares en categorías relacionadas con el negocio y se define así:

<acción>-<ndo>-un <objeto>

Por ejemplo: *Haciendo una verificación del producto* es un conjunto de características que agruparía las que ya se mencionaron y otras más.

El enfoque DIC define cinco actividades estructurales “colaborativas” [Coa99] (en el enfoque DIC se llaman “procesos”), como se muestra en la figura 3.5.

El DIC pone más énfasis que otros métodos ágiles en los lineamientos y técnicas para la administración de proyectos. A medida que éstos aumentan su tamaño y complejidad, no es raro que la administración de proyectos *ad hoc* sea inadecuada. Para los desarrolladores, sus gerentes y otros participantes, es esencial entender el estado del proyecto, es decir, los avances realizados y los problemas que han surgido. Si la presión por cumplir el plazo de entrega es mucha, tiene importancia crítica determinar si la entrega de los incrementos del software está programada en forma adecuada. Para lograr esto, el DIC define seis puntos de referencia durante el diseño e implementación de una característica: “recorrer por el diseño, diseño, inspección del diseño, código, inspección del código, decisión de construir” [Coa99].

3.5.6 Desarrollo esbelto de software (DES)

El desarrollo esbelto de software (DES) adapta los principios de la manufactura esbelta al mundo de la ingeniería de software. Los principios de esbeltez que inspiran al proceso DES se resumen como sigue [Pop03], [Pop04]: *eliminar el desperdicio, generar calidad, crear conocimiento, aplazar el compromiso, entregar rápido, respetar a las personas y optimizar al todo*.

Es posible adaptar cada uno de estos principios al proceso del software. Por ejemplo, *eliminar el desperdicio* en el contexto de un proyecto de software ágil significa [Das05]: 1) no agregar características o funciones extrañas, 2) evaluar el costo y el efecto que tendrá en la programación de actividades cuya función es solo cumplir un requerimiento solicitado, 3) eliminar cualesquier etapas superficiales del proceso, 4) establecer mecanismos para mejorar la forma en la que los miembros

del equipo obtienen información, 5) asegurar que las pruebas detecten tantos errores como sea posible, 6) reducir el tiempo requerido para pedir y obtener una decisión que afecta al software o al proceso que se aplica para crearlo, y 7) simplificar la manera en la que se transmite la información a todos los participantes involucrados en el proceso.

Para un análisis detallado del DES y para conocer lineamientos prácticos a fin de implementar el proceso, debe consultarse [Pop06a] y [Pop06b].

3.5.7 Modelado ágil (MA)

Hay muchas situaciones en las que los ingenieros de software deben construir sistemas grandes de importancia crítica para el negocio. El alcance y complejidad de tales sistemas debe modelarse de modo que: 1) todos los actores entiendan mejor cuáles son las necesidades que deben satisfacerse, 2) el problema pueda dividirse con eficacia entre las personas que deben resolverlo, y 3) se asegure la calidad a medida que se hace la ingeniería y se construye el sistema.

En los últimos 30 años se ha propuesto una gran variedad de métodos de modelado y notación para la ingeniería de software con objeto de hacer el análisis y el diseño tanto en la arquitectura como en los componentes). Estos métodos tienen su mérito, pero se ha demostrado que son difíciles de aplicar y sostener (en muchos proyectos). Parte del problema es el “peso” de dichos métodos de modelación. Con esto se hace referencia al volumen de la notación que se requiere, al grado de formalismo sugerido, al tamaño absoluto de los modelos para proyectos grandes y a la dificultad de mantener el(s) modelo(s) conforme suceden los cambios. Sin embargo, el análisis y el modelado del diseño tienen muchos beneficios para los proyectos grandes, aunque no fuera más que porque hacen a esos proyectos intelectualmente más manejables. ¿Hay algún enfoque ágil para el modelado de la ingeniería de software que brinde una alternativa?

En el “sitio oficial de modelado ágil”, Scott Ambler [Amb02a] describe el *modelado ágil* (MA) del modo siguiente:

El modelado ágil (MA) es una metodología basada en la práctica para modelar y documentar con eficacia los sistemas basados en software. En pocas palabras, es un conjunto de valores, principios y prácticas para hacer modelos de software aplicables de manera eficaz y ligera a un proyecto de desarrollo de software. Los modelos ágiles son más eficaces que los tradicionales porque son sólidos/buenos, sin pretender ser perfectos.

El modelado ágil adopta todos los valores del manifiesto ágil. La filosofía de modelado ágil afirma que un equipo ágil debe tener la valentía para tomar decisiones que impliquen rechazar un diseño y reconstruirlo. El equipo también debe tener la humildad de reconocer que los tecnólogos no tienen todas las respuestas y que los expertos en el negocio y otros participantes deben ser respetados e incluidos.

Aunque el MA sugiere una amplia variedad de principios de modelado “fundamentales” y “suplementarios”, aquellos que son exclusivos del MA son los siguientes [Amb02a]:

Modelo con un propósito. Un desarrollador que use el MA debe tener en mente una meta específica (por ejemplo, comunicar información al cliente o ayudarlo a entender mejor algún aspecto del software) antes de crear el modelo. Una vez identificada la meta para el modelo, el tipo y nivel de detalle de la notación por usar serán más obvios.

Uso de modelos múltiples. Hay muchos modelos y notaciones diferentes que pueden usarse para describir el software. Para la mayoría de proyectos solo es esencial un pequeño subconjunto. El MA sugiere que para dar la perspectiva necesaria, cada modelo debe presentar un diferente aspecto del sistema y que sólo deben utilizarse aquellos modelos que den valor al público al que se dirigen.



WebRef

En la dirección www.agilemodeling.com hay información amplia sobre el modelo ágil.



Cita:

Jerry Seinfeld

“Hoy día fuí a la formación por una mañana para el refresher... y no fue fácil. Hubo todo una gran cantidad de productos. Al recorrer vi uno que era de acción rápido, pero no que era de largo duración... ¿Qué es más importante, el presente o el futuro?”



Viajar ligero. Conforme avanza el trabajo de ingeniería de software, conserve sólo aquellos modelos que agreguen valor a largo plazo y elimine los demás. Todo producto del trabajo que se conserve debe recibir mantenimiento cuando haya cambios. Esto representa una labor que hace lento al equipo. Ambler [Amb02a] afirma que "cada vez que se decide conservar un modelo, se pierde agilidad en nombre de la conveniencia de tener disponible esa información en forma abstracta para el equipo (y de ese modo mejorar potencialmente la comunicación dentro del equipo, así como con los participantes)".

El contenido es más importante que la representación. El modelo debe transmitir información al público al que se dirige. Un modelo con sintaxis perfecta que transmite poco contenido útil no es tan valioso como otro que tenga notación defectuosa, pero que, no obstante, provea contenido de valor para los usuarios.

Conocer los modelos y herramientas que se utilizan en su creación. Entender las fortalezas y debilidades de cada modelo y las herramientas que se emplean para crearlos.

Adaptación local. El enfoque de modelado debe adaptarse a las necesidades del equipo ágil.

Un segmento importante de la comunidad de ingeniería de software ha adoptado el lenguaje de unificado de modelado (UML, por sus siglas en inglés)¹⁶, como el método preferido para representar modelos del análisis y del diseño. El proceso unificado (véase el capítulo 2) fue desarrollado para proveer una estructura para la aplicación del UML. Scott Ambler [Amb06] desarrolló una versión simplificada del PU que integra su filosofía de modelado ágil.

3.5.8 El proceso unificado ágil (PUA)

El *proceso unificado ágil* (PUA) adopta una filosofía "en serie para lo grande" e "iterativa para lo pequeño" [Amb06] a fin de construir sistemas basados en computadora. Al adoptar las actividades en fase clásicas del PU –concepción, elaboración, construcción y transición–, el PUA brinda un revestimiento en serie (por ejemplo, una secuencia lineal de actividades de ingeniería de software) que permite que el equipo visualice el flujo general del proceso de un proyecto de software. Sin embargo, dentro de cada actividad, el equipo repite con objeto de alcanzar la agilidad y entregar tan rápido como sea posible incrementos de software significativos a los usuarios finales. Cada iteración del PUA aborda las actividades siguientes [Amb06]:

- **Modelado.** Se crean representaciones de UML de los dominios del negocio y el problema. No obstante, para conservar la agilidad, estos modelos deben ser "sólo suficientemente buenos" [Amb06] para permitir que el equipo avance.
- **Implementación.** Los modelos se traducen a código fuente.
- **Pruebas.** Igual que con la XP, el equipo diseña y ejecuta una serie de pruebas para detectar errores y garantizar que el código fuente cumple sus requerimientos.
- **Despliegue.** Como en la actividad general del proceso que se estudió en los capítulos 1 y 2, el despliegue en este contexto se centra en la entrega de un incremento de software y en la obtención de retroalimentamiento de los usuarios finales.
- **Configuración y administración del proyecto.** En el contexto del PUA, la administración de la configuración (véase el capítulo 22) incluye la administración del cambio y el riesgo, y el control de cualesquier productos del trabajo persistentes¹⁷ que produzca el equipo.

¹⁶ En el apéndice I se presenta un método breve para aprender UML.

¹⁷ Un *producto del trabajo persistente* es un modelo o documento o caso de prueba producido por el equipo y que se conservará durante un período indeterminado. No se eliminará una vez entregado el incremento de software.

Viajar ligero. La administración del proyecto da seguimiento y controla el avance del equipo y coordina sus actividades.

- **Administración del ambiente.** La administración del ambiente coordina una infraestructura del proceso que incluye estándares, herramientas y otra tecnología de apoyo de la que dispone el equipo.
- **Aunque el PUA tiene conexiones históricas y técnicas con el lenguaje unificado de modelado, es importante observar que el modelado UML puede usarse junto con cualesquier de los modelos de proceso ágil descritos en la sección 3.5.**

HERRAMIENTAS DE SOFTWARE

car herramientas que lo apoyan. Las que se mencionan a continuación tienen características que las hacen particularmente útiles para los proyectos ágiles.

OnTime. desarrollado por Axosoft (www.axosoft.com), presta apoyo a la administración de un proceso ágil para distintas actividades técnicas dentro del proceso.

Ideogramic. desarrollado por Ideogramic (www.ideogramic.com), es un conjunto de herramientas UML desarrolladas específicamente para usuarios dentro de un proceso ágil. Together Tool Set, distribuido por Borland (www.borland.com), proporciona un grupo de herramientas para apoyar muchas actividades técnicas dentro de XP y otros procesos ágiles.



Objetivo: El objetivo de las herramientas de desarrollo ágil es ayudar en uno o más aspectos de éste, con énfasis en facilitar la elaboración rápida de software funcional. Estas herramientas también pueden emplearse cuando se aplican modelos de proceso prescriptivo (véase el capítulo 2).

Mecánica: Las herramientas de mecánica varían. En general, las herramientas ágiles incluyen el apoyo automatizado para la planeación del proyecto, el desarrollo de casos y la obtención de requerimientos, el diseño rápido, la generación de código y la realización de pruebas.

Herramientas representativas:¹⁸

Nota: Debe a que el desarrollo ágil es un tema de moda, la mayoría de los vendedores de herramientas de software tratan de colo-

3.6 CONJUNTO DE HERRAMIENTAS PARA EL PROCESO ÁGIL

Algunos defensores de la filosofía ágil afirman que las herramientas automatizadas de software (por ejemplo, las de diseño) deben verse como un complemento menor de las actividades del equipo, y no como algo fundamental para el éxito. Sin embargo, Alistair Cockburn [Coc04] sugiere que las herramientas tienen un beneficio y que "los equipos ágiles favorecen el uso de herramientas que permiten el flujo rápido de entendimiento. Algunas de estas herramientas son sociales y comienzan incluso en la etapa de reclutamiento. Otras son tecnológicas y ayudan a que los equipos distribuidos simulen su presencia física. Muchas herramientas son físicas y permiten que las personas las manipulen en talleres".

Prácticamente todos los modelos de la filosofía ágil son elementos clave en la contratación del personal adecuado (reclutamiento), la colaboración en equipo, la comunicación con los participantes y la administración indirecta; por eso, Cockburn afirma que las "herramientas" que se abocan a dichos aspectos son factores críticos para el éxito de la agilidad. Por ejemplo, una "herramienta" de reclutamiento tal vez sea el requerimiento de que un prospecto a miembro del equipo pase algunas horas programando en pareja con alguien que ya es integrante del equipo. El "ajuste" se evalúa de inmediato.

Las "herramientas" de colaboración y comunicación por lo general son de baja tecnología e incorporan cualquier mecanismo ("proximidad física, pizarrones, tableros, tarjetas y notas ad-

¹⁸ Las herramientas mencionadas aquí no son obligatorias sólo son una muestra en esta categoría. En la mayoría de los casos, sus nombres son marcas registradas por sus respectivos desarrolladores.

heribles" [Coc04] que provea información y coordinación entre los desarrolladores ágiles. La comunicación activa se logra por medio de la dinámica del equipo (por ejemplo, la programación en parejas), mientras que la comunicación pasiva se consigue con "radiadores de información" (un tablero que muestre el estado general de los distintos componentes de un incremento). Las herramientas de administración de proyectos no ponen el énfasis en la gráfica de Gantt y la sustituyen con otras de valor agregado o "gráficas de pruebas creadas versus pasadas; otras herramientas ágiles se utilizan para optimizar el ambiente en el que trabaja el equipo ágil (por ejemplo, áreas más eficientes para reunirse), mejoran la cultura del equipo por medio de interacciones sociales (equipos con algo en común), dispositivos físicos (piramones electrónicos) y el mejoramiento del proceso (por ejemplo, la programación por parejas o la caja de tiempo)" [Coc04].

Algunas de las mencionadas son en verdad herramientas? Si, lo son, si facilitan el trabajo efectuado por un miembro del equipo ágil y mejoran la calidad del producto final.

3.7 RESUMEN

En una economía moderna, las condiciones del mercado cambian con rapidez, los clientes y usuarios finales necesitan evolucionar y surgen nuevas amenazas competitivas sin aviso previo. Los profesionales deben enfocar la ingeniería de software en forma que les permita mantenerse ágiles para definir procesos manejables, adaptativos y esbeltos que satisfagan las necesidades de los negocios modernos.

Una filosofía ágil para la ingeniería de software pone el énfasis en cuatro aspectos clave: la importancia de los equipos con organización propia que tienen el control sobre el trabajo que realizan, la comunicación y colaboración entre los miembros del equipo y entre los profesionales y sus clientes, el reconocimiento de que el cambio representa una oportunidad y la insistencia en la entrega rápida de software que satisface al consumidor. Los modelos de proceso ágil han sido diseñados para abordar cada uno de estos aspectos.

La programación extrema (XP) es el proceso ágil de más uso. Organizada con cuatro actividades estructurales: planeación y prueba, diseño, codificación y pruebas, la XP sugiere cierto número de técnicas innovadoras y poderosas que permiten a un equipo ágil generar entregas frecuentes de software que posee características y funcionalidad que han sido descritas y clasificadas según su prioridad por los participantes.

Otros modelos de proceso ágil también insisten en la colaboración humana y en la organización propia del equipo, pero definen sus actividades estructurales y seleccionan diferentes tipos de importancia. Por ejemplo, el DAS utiliza un proceso iterativo que incluye un ciclo de planeación adaptativa, métodos relativamente riguros para recabar requerimientos, y un ciclo de desarrollo iterativo que incorpora grupos de consumidores y revisiones técnicas formales como mecanismos de retroalimentación en tiempo real. El Scrum pone el énfasis en el uso de un conjunto de patrones de software que han demostrado ser eficaces para proyectos que tienen plazos de entrega apretados, requerimientos cambiantes o que se emplean en negocios críticos.

Cada patrón de proceso define un conjunto de tareas de desarrollo y permite al equipo Scrum construir un proceso que se adapte a las necesidades del proyecto. El método de desarrollo de sistemas dinámicos (MDSD) resalta el uso de la programación con caja de tiempo y sugiere que en cada incremento de software sólo se requiere el trabajo suficiente que facilite el paso al incremento que sigue. Cristal es una familia de modelos de proceso ágil que se adaptan a las características específicas del proyecto.

El desarrollo impulsado por las características (DIC) es algo más "ormal" que otros métodos ágiles, pero conserva su agilidad al centrar al equipo del proyecto en el desarrollo de características, funciones valiosas para el cliente que pueden implementarse en dos semanas o menos. El

PROBLEMAS Y PUNTOS POR EVALUAR

desarrollo esbelto de software (DES) ha adaptado los principios de la manufactura esbelta al mundo de la ingeniería de software. El modelado ágil (MA) sugiere que el modelado es esencial para todos los sistemas, pero que la complejidad, tipo y tamaño del modelo deben ajustarse al software que se va a elaborar. El proceso unificado ágil (PUA) adopta una filosofía "serial en lo grande" e "iterativo en lo pequeño" para la elaboración de software.

3.2. Describa con sus propias palabras la *agilidad* (para proyectos de software).

3.3. ¿Por qué un proceso iterativo hace más fácil administrar el cambio? Es iterativo todo proceso ágil analizado en este capítulo? ¿Es posible terminar un proyecto en solo una iteración y aún así conseguir que sea ágil? Explique sus respuestas.

3.4. ¿Podrá describirse cada uno de los procesos ágiles con el uso de las actividades estructurales mencionadas en el capítulo 2? Construya una tabla que mapee las actividades generales en las actividades definidas para cada proceso ágil.

3.5. Proponga un "principio de agilidad" más que ayudaría al equipo de ingeniería de software a ser aún más manejable.

3.6. Seleccione un principio de agilidad mencionado en la sección 3.3.1 y trate de determinar si está incluido en cada uno de los modelos de proceso presentados en este capítulo. [Nota: sólo se presentó el panorama general de estos modelos de proceso, por lo que tal vez no fuera posible determinar si un principio está incluido en uno o más de ellos, a menos que el lector hiciera una investigación (lo que no se requiere para este problema)].

3.7. ¿Por qué cambian tanto los requerimientos? Después de todo, ¿la gente no sabe lo que quiere?

3.8. La mayoría de modelos de proceso ágil recomiendan la comunicación cara a cara. No obstante, los miembros del equipo de software y sus clientes tal vez estén alejados geográficamente. ¿Piensa usted que esto implica que debe evitarse la separación geográfica? ¿Se le ocurren formas de resolver este problema?

3.9. Escríba una historia de usuario XP que describa la característica de "lugares favoritos" o "marcadores" disponible en la mayoría de navegadores Web.

3.10. ¿Qué es una solución en punta en XP?

3.11. Describa con sus propias palabras los conceptos de rediseño y programación en parejas de XP.

3.12. Haga otras lecturas y describa lo que es una caja de tiempo. ¿Cómo ayuda a un equipo DAS para que entrene incrementos de software en un corto período?

3.13. ¿Se logra el mismo resultado con la regla de 80% del MDSD y con el enfoque de la caja de tiempo del DAS?

3.14. Con el formato de patrón de proceso presentado en el capítulo 2, desarrolle uno para cualquiera de los patrones Scrum presentados en la sección 3.5.2.

3.15. ¿Por qué se le llama a Cristal familia de métodos ágiles?

3.16. Con el formato de característica DIC descrito en la sección 3.5.5, defina un conjunto de características para un navegador web. Luego desarrolle un conjunto de características para el primer conjunto.

3.17. Visite el sitio oficial de modelación ágil y elabore la lista completa de todos los principios fundamentales y secundarios del MA.

3.18. El conjunto de herramientas propuestas en la sección 3.6 da apoyo a muchos de los aspectos "suaves" de los métodos ágiles. Debido a que la comunicación es tan importante, recomienda un conjunto de herramientas reales que podría utilizarse para que los participantes de un equipo ágil se comuniquen mejor.

LECTURAS ADICIONALES Y FUENTES DE INFORMACIÓN

La filosofía general y principios que subyacen al desarrollo de software ágil se estudian a profundidad en muchos de los libros mencionados a lo largo de este capítulo. Además, los textos de Shaw y Warden (*The Art of Agile Development*, O'Reilly Media, Inc., 2008), Hunt (*Agile Software Faster*, Prentice-Hall, 2002) presentan análisis útiles del tema. Agarwal (*Better Software Faster*, Prentice-Hall, 2002) y Carmanno (*Managing Agile Projects*, Multi-Media Publications, 2005). Highsmith (*Agile Project Management: Creating Innovative Products*, Addison-Wesley, 2004) y Larman (*Agile and Iterative Development: A Manager's Guide*, Addison-Wesley, 2003) presentan el punto de vista de la administración y consideran ciertos aspectos de la administración de proyectos. Highsmith (*Agile Software Development Ecosystems*, Addison-Wesley, 2002) expone una encuesta acerca de los principios, procesos y prácticas ágiles. Booch y sus colegas (*Balancing Agility and Discipline*, Addison-Wesley, 2004) hacen un análisis fructífero del delicado equilibrio entre agilidad y disciplina.

Martin (*Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice-Hall, 2009) explica los principios, patrones y prácticas que se requieren para desarrollar "código limpio" en un ambiente de ingeniería del software ágil. Leffingwell (*Scaling Software Agility: Best Practices for Large Enterprises*, Addison-Wesley, 2007) estudia estrategias para ampliar las prácticas ágiles en proyectos grandes. Lippert y Rook (*Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*, Wiley, 2006) analizan el uso del rediseño cuando se aplica a sistemas grandes y complejos. Ståmleos y Sletos (*Agile Software Development Quality Assurance*, IGI Global, 2007) analizan las técnicas SQA que forman la filosofía ágil.

En la última década se han escrito decenas de libros sobre programación extrema. Beck (*Extreme Programming Explained: Embrace Change*, 2a. ed., Addison-Wesley, 2004) sigue siendo la referencia definitiva al respecto. Además, Jeffries y sus colegas (*Extreme Programming Installed*, Addison-Wesley, 2000), succi y Marchesi (*Extreme Programming Examined*, Addison-Wesley, 2001), Newkirk y Martin (*Extreme Programming in Practice*, Addison-Wesley, 2001) y Auer y sus colegas (*Extreme Programming Applied: Play to Win*, Addison-Wesley, 2001) hacen un análisis detallado de XP y dan una guía para aplicarla de la mejor forma. McBreen (*Questioning Extreme Programming*, Addison-Wesley, 2003) adopta un enfoque crítico sobre XP y define cuándo y dónde es apropiada. Un estudio profundo de la programación por parejas se presenta en McBreen (*Pair Programming Illuminated*, Addison-Wesley, 2003).

Highsmith (1990) analiza con detalle el DAS. Schwaber (*The Enterprise and Scrum*, Microsoft Press, 2007) estudia el empleo de Scrum para proyectos que tienen un efecto grande en los negocios. Los detalles de Scrum los estudian Schwaber y Beedle (*Agile Software Development with SCRUM*, Prentice-Hall, 2001). Algunos tratamientos útiles del MDSD han sido escritos por DSDM Consortium (*DSDM: Business Focused Development*, 2a. ed., Pearson Education, 2003) y Stapleton (*DSDM: The Method in Practice*, Addison-Wesley, 1997). Cockburn (*Crystall Clear*, Addison-Wesley, 2005) presenta un panorama excelente de la familia de procesos Cristal. Palmer y Feising (*Pal02*) dan un tratamiento detallado del DIC. Carmichael y Haywood (*Better Software Faster*, Prentice-Hall, 2002) proporcionan otro análisis útil del DIC, que incluye un recorrido, paso a paso, por la mecánica del proceso. Poppandieck y Poppandieck (*Lean Development: An Agile Toolkit for Software Development Managers*, Addison-Wesley, 2003) dan lineamientos para la administración y el control de proyectos ágiles. Ambler y Jeffries (*Agile Modeling*, Wiley, 2002) estudian el MA con cierta profundidad.

En internet existe una amplia variedad de fuentes de información sobre el desarrollo de software ágil. En el sitio web del libro hay una lista actualizada de referencias en la Red Mundial que son relevantes para el proceso ágil, en la dirección: www.mhhe.com/engcs/compsci/professman/professional/oic/ser.htm.