enormemente. Lorenz y Kidd sugieren que las aplicaciones con una GUI tienen entre dos y tres veces más clases de apoyo que clases clave. Las aplicaciones no GUI tienen entre una y dos veces más clases de apoyo que clases clave.

Número de subsistemas. Un *subsistema* es un agregado de clases que apoyan una función que es visible para el usuario final de un sistema. Una vez identificados los subsistemas, es más fácil plantear un calendario razonable en el cual el trabajo sobre los subsistemas se divide entre el personal del proyecto.

Para usarse de manera efectiva en un entorno de ingeniería del software orientado a objeto, es necesario recopilar métricas similares a las anotadas anteriormente, junto con medidas del proyecto, tales como esfuerzo empleado, errores y defectos descubiertos, y modelos o páginas de documentación producidos. Conforme crece la base de datos (después de haber completado algunos proyectos), las relaciones entre las medidas orientadas a objeto y las medidas del proyecto proporcionarán métricas que pueden auxiliar en la estimación del proyecto.

25.2.5 Métricas orientadas a caso de uso

Los casos de uso[6] se utilizan ampliamente como un método para describir los requerimientos en el dominio en el nivel del cliente o empresarial, que implican características y funciones del software. Parecería razonable usar el caso de uso como una medida de normalización similar a LOC o PF. Como los PF, el caso de uso se define al principio del proceso del software, lo que permite emplearlo para estimación antes de iniciar actividades significativas de modelado y construcción. Los casos de uso describen (de manera indirecta, al menos) las funciones y características visibles para el usuario que son requisitos básicos para un sistema. El caso de uso es independiente del lenguaje de programación. Además, el número de casos de uso es directamente proporcional al tamaño de la aplicación en LOC y al número de casos de prueba que tendrán que designarse para ejercitar por completo la aplicación.

Puesto que los casos de uso pueden crearse en niveles de abstracción enormemente diferentes, no hay "tamaño" estándar para un caso de uso. Sin una medida estándar de lo que es un caso de uso, su aplicación como medida de normalización (por ejemplo, esfuerzo empleado por caso de uso) causa suspicacia.

Los investigadores sugieren *puntos de caso de uso* (PCU) como un mecanismo para estimar el esfuerzo del proyecto y otras características. Los PCU son una función del número de actores y transacciones implicados por los modelos de caso de uso y su análogo al PF en algunas formas. Si se tiene más interés en este tema, véase [Cle06].

25.2.6 Métricas de proyecto *webapp*

El objetivo de todos los proyectos *webapp* es entregar al usuario final una combinación de contenido y funcionalidad. Las medidas y métricas usadas para proyectos tradicionales de ingeniería del software son difíciles de traducir directamente a *webapps*. Sin embargo, es posible desarrollar una base de datos que permita el acceso a medidas productivas y calidad internas derivadas de algunos proyectos. Entre las medidas que pueden recopilarse están:

Número de páginas web estáticas. Las páginas web con contenido estático (es decir, el usuario final no tiene control sobre el contenido que se despliega en la página) son las más comunes de todas las características *webapp*. Dichas páginas representan complejidad relativamente baja y por lo general requieren menos esfuerzo para construirlas que las páginas dinámicas. Esta medida proporciona un indicio del tamaño global de la aplicación y del esfuerzo requerido para desarrollarla.

6 Los casos de uso se introdujeron en los capítulos 5 y 6.

Número de páginas web dinámicas. Las páginas web con contenido dinámico (es decir, acciones del usuario final u otros factores externos dan como resultado contenido personalizado que se despliega en la página) son esenciales en toda aplicación de comercio electrónico, motores de búsqueda, aplicaciones financieras y muchas otras categorías *webapp*. Dichas páginas representan complejidad relativa más alta y requieren más esfuerzo para construirlas que las páginas estáticas. Esta medida proporciona un indicio del tamaño global de la aplicación y del esfuerzo requerido para desarrollarla.

Número de vínculos de página internos. Los vínculos de página internos son punteros que proporcionan un hipervínculo hacia alguna otra página web dentro de la *webapp*. Esta medida proporciona un indicio del grado de acoplamiento arquitectónico dentro de la *webapp*. Conforme el número de vínculos de página aumenta, el esfuerzo empleado en el diseño y construcción de la navegación también aumenta.

Número de objetos de datos persistentes. Una *webapp* puede tener acceso a uno o más objetos de datos persistentes (por ejemplo, una base de datos o archivo de datos). Conforme crece el número de objetos de datos persistentes, la complejidad de la *webapp* también crece y el esfuerzo para implementarla lo hace de manera proporcional.

Número de sistemas externos puestos en interfaz. Con frecuencia, las *webapps* deben tener interfaces con aplicaciones empresariales "de puerta trasera". Conforme crecen los requerimientos para interfaces, también crecen la complejidad y el desarrollo del sistema.

Número de objetos de contenido estáticos. Los objetos de contenido estáticos abarcan información basada en texto, gráfica, video, animación y audioestática, que se incorporan dentro de la *webapp*. Múltiples objetos de contenido pueden aparecer en una sola página web.

Número de objetos de contenido dinámicos. Los objetos de contenido dinámicos se generan con base en las acciones del usuario final y abarcan información basada en texto, gráfica, video, animación y audio, generada internamente, que se incorpora dentro de la *webapp*. Múltiples objetos de contenido pueden aparecer en una sola página web.

Número de funciones ejecutables. Una función ejecutable (por ejemplo, un guión o applet) proporciona cierto servicio computacional al usuario final. Conforme el número de funciones ejecutables aumenta, también crece el esfuerzo de modelado y construcción.

Cada una de las medidas anteriores puede determinarse en una etapa relativamente temprana. Por ejemplo, puede definirse una métrica que refleje el grado de personalización del usuario final que se requiere para la *webapp* y correlacionarla con el esfuerzo empleado en el proyecto y/o los errores descubiertos conforme se realizan revisiones y se aplican pruebas. Para lograr esto, se define

N_{pe} = número de páginas web estáticas
N_{pd} = número de páginas web dinámicas

Entonces,

$$\text{Índice de personalización, } C = \frac{N_{pd}}{N_{pd} + N_{pe}}$$

El valor de C varía de 0 a 1. Conforme C se hace más grande, el nivel de personalización de la *webapp* se convierte en un problema técnico considerable.

Similares métricas *webapp* pueden calcularse y correlacionarse con medidas del proyecto, como esfuerzo empleado, errores y defectos descubiertos y modelos o páginas de documenta-

ción producidos. Conforme crece la base de datos (después de completar algunos proyectos), las relaciones entre las medidas de la *webapp* y las medidas del proyecto proporcionarán indicadores que pueden auxiliar en la estimación del proyecto.

HERRAMIENTAS DE SOFTWARE

Métricas del proyecto y del proceso

Objetivo: Auxiliar en la definición, recolección, evaluación y reporte de medidas y métricas de software.

Mecánica: Cada herramienta varía en su aplicación, pero todas ofrecen mecanismos para recolectar y evaluar datos que conducen al cálculo de métricas de software.

Herramientas representativas:[7]

Function Point WORKBENCH, desarrollada por Charismatek (**www.charismatek.com.au**), ofrece un amplio arreglo de métricas orientadas a PF.

MetricCenter, desarrollada por Distributive Software (**www.distributive.com**), respalda procesos de automatización para recolección de datos, análisis, formateo de gráficos, generación de reportes y otras tareas de medición.

PSM Insight, desarrollada por Practical Software and Systems Measurement (**www.psmsc.com**), auxilia en la creación y en el análisis posterior de una base de datos de medición de proyecto.

SLIM tool set, desarrollada por QSM (**www.qsm.com**), proporciona un conjunto exhaustivo de métricas y herramientas de estimación.

SPR tool set, desarrollada por Software Productivity Research (**www.spr.com**), ofrece una colección exhaustiva de herramientas orientadas a PF.

TychoMetrics, desarrollada por Predicate Logic, Inc. (**www.predicate.com**), es una suite de herramientas que sirven para recopilar y reportar métricas de administración.

25.3 MÉTRICAS PARA CALIDAD DE SOFTWARE

El software es una entidad compleja. Por tanto, deben esperarse errores conforme se desarrollen productos operativos. Las métricas de proceso tienen la intención de mejorar el proceso del software, de modo que los errores se descubran en la forma más efectiva.

La meta dominante de la ingeniería del software es producir un sistema, aplicación o producto de alta calidad dentro de un marco temporal que satisfaga una necesidad de mercado. Para lograr esta meta, deben aplicarse métodos efectivos acoplados con herramientas modernas dentro del contexto de un proceso de software maduro. Además, un buen ingeniero de software (y los buenos gerentes de ingeniería del software) deben medir si la alta calidad es realizable.

La calidad de un sistema, aplicación o producto sólo es tan buena como los requerimientos que describen el problema, el diseño que modela la solución, el código que conduce a un programa ejecutable y las pruebas que ejercitan el software para descubrir errores. Conforme el software se somete a ingeniería, pueden usarse mediciones para valorar la calidad de los modelos de requerimientos y de diseño, el código fuente y los casos de prueba que se crearon. Para lograr esta valoración en tiempo real, las métricas de producto (capítulo 23) se aplican a fin de evaluar la calidad de los productos operativos de la ingeniería del software en forma objetiva, en lugar de subjetiva.

Un gerente de proyecto también debe evaluar la calidad conforme avanza el proyecto. Las métricas privadas recopiladas individualmente por los ingenieros del software se combinan para proporcionar resultados en el nivel del proyecto. Aunque muchas medidas de calidad pueden recopilarse, el empuje primario en el nivel del proyecto es medir los errores y defectos. Las métricas derivadas de estas medidas proporcionan un indicio de la efectividad de las actividades, individuales y grupales, de aseguramiento y control de la calidad del software.

Métricas como errores de producto operativo por punto de función, errores descubiertos por hora de revisión y errores descubiertos por prueba por hora proporcionan comprensión de la eficacia de cada una de las actividades implicadas por la métrica. Los datos de error también

7 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas existentes en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

pueden usarse para calcular la *eficiencia de remoción de defecto* (ERD) para cada actividad de marco conceptual del proceso. La ERD se estudia en la sección 25.3.3.

25.3.1 Medición de la calidad

Aunque existen muchas medidas de calidad del software,[8] la exactitud, capacidad de mantenimiento, integridad y usabilidad proporcionan útiles indicadores para el equipo del proyecto. Gilb [Gil88] sugiere definiciones y medidas para cada una.

WebRef

Una excelente fuente de información acerca de la calidad del software y sobre temas relacionados (incluidas métricas) puede encontrarse en **www.qualityworld.com**

Exactitud. Un programa debe operar correctamente o proporcionará poco valor a sus usuarios. La exactitud es el grado en el cual el software realiza la función requerida. La medida más común de la exactitud son los defectos por KLOC, donde un defecto se define como una falta verificada de acuerdo con los requerimientos. Cuando se considera la calidad global de un producto de software, los defectos son aquellos problemas reportados por un usuario del programa después de que el programa se liberó para su uso general. Con propósitos de valoración de calidad, los defectos se cuentan sobre un periodo estándar, por lo general un año.

Capacidad de mantenimiento. El mantenimiento y soporte del software representan más esfuerzo que cualquiera otra actividad de ingeniería del software. La capacidad de mantenimiento es la facilidad con la que un programa puede corregirse si se encuentra un error, la facilidad con que se adapta si su entorno cambia o de mejorar si el cliente quiere un cambio en requerimientos. No hay forma de medir directamente la capacidad de mantenimiento; por tanto, deben usarse medidas indirectas. Una métrica simple orientada a tiempo es el *tiempo medio al cambio* (TMC), el tiempo que tarda en analizarse la petición de cambio, diseñar una modificación adecuada, implementar el cambio, probarlo y distribuirlo a todos los usuarios. En promedio, los programas con capacidad de mantenimiento tendrán un TMC más bajo (para tipos de cambios equivalentes) que los que no tienen dicha capacidad.

Integridad. La integridad del software se ha vuelto cada vez más importante en la era de los ciberterroristas y hackers. Este atributo mide la habilidad de un sistema para resistir ataques (tanto accidentales como intencionales) a su seguridad. Los ataques pueden hacerse en los tres componentes de software: programas, datos y documentación.

Para medir la integridad, deben definirse dos atributos adicionales: amenaza y seguridad. *Amenaza* es la probabilidad (que puede estimarse o derivarse de evidencia empírica) de que un ataque de un tipo específico ocurrirá dentro de un tiempo dado. *Seguridad* es la probabilidad (que puede estimarse o derivarse de evidencia empírica) de que el ataque de un tipo específico se repelerá. La integridad de un sistema puede definirse entonces como:

$$\text{Integridad} = \Sigma[1 - (\text{amenaza} \times (1 - \text{seguridad}))]$$

Por ejemplo, si la amenaza (la probabilidad de que un ataque ocurrirá) es de 0.25 y la seguridad (la probabilidad de repeler un ataque) es de 0.95, la integridad del sistema es 0.99 (muy alta). Si, por otra parte, la probabilidad de amenaza es 0.50 y la probabilidad de repeler un ataque es de solamente 0.25, la integridad del sistema es 0.63 (inaceptablemente baja).

Usabilidad. Si un programa no es fácil de usar, con frecuencia está condenado al fracaso, incluso si las funciones que realiza son valiosas. La usabilidad es un intento por cuantificar la facilidad de uso y puede medirse en términos de las características que se presentaron en el capítulo 11.

8 En el capítulo 23 se presentó un análisis detallado de los factores que influyen en la calidad y en las métricas del software que pueden usarse para valorar la calidad del software.

Los cuatro factores recién descritos sólo son una muestra de los que se han propuesto como medidas para la calidad del software. El capítulo 23 considera este tema con detalles adicionales.

25.3.2 Eficiencia en la remoción del defecto

Una métrica de calidad que proporciona beneficio tanto en el nivel del proyecto como en el del proceso es la *eficiencia de remoción del defecto* (ERD). En esencia, la ERD es una medida de la habilidad de filtrado de las acciones de aseguramiento y control de la calidad según se aplican a lo largo de todas las actividades del marco conceptual del proceso.

Cuando se considera para un proyecto como un todo, la ERD se define en la forma siguiente:

$$\text{ERD} = \frac{E}{E + D}$$

donde E es el número de errores que se encontraron antes de entregar el software al usuario final y D es el número de defectos que se encontraron después de la entrega.

Si la ERD es baja conforme se avanza a través del análisis y el diseño, emplee algo de tiempo para mejorar la forma en la que realiza las revisiones técnicas.

El valor ideal para la ERD es 1. Es decir, no se encuentran defectos en el software. En realidad, D será mayor que 0, pero el valor de la ERD todavía puede tender a 1 conforme E aumenta para un valor dado de D. De hecho, conforme E aumenta, es probable que el valor final de D disminuirá (los errores se filtran antes de convertirse en defectos). Si se usa como una métrica que proporciona un indicio de la capacidad de filtrado de las actividades de control y aseguramiento de la calidad, la ERD alienta a un equipo de software a instituir técnicas para encontrar tantos errores como sea posible antes de entregar.

La ERD también puede usarse dentro del proyecto a fin de valorar la habilidad de un equipo para encontrar errores antes de que pasen a la siguiente actividad de marco conceptual o acción de ingeniería del software. Por ejemplo, el análisis de requerimientos produce un modelo de requerimientos que puede revisarse para encontrar y corregir errores. Aquellos que no se encuentran durante la revisión del modelo de requerimientos pasan al diseño (donde pueden o no encontrarse). Cuando se usan en este contexto, la ERD se redefine como

$$\text{ERD}_i = \frac{E_i}{E_i + E_{i+1}}$$

donde E_i es el número de errores encontrados durante la acción i de ingeniería del software y E_{i+1} es el número de errores encontrados durante la acción $i + 1$ de ingeniería del software que son rastreables por errores que no se descubrieron en la acción i de ingeniería del software.

Un objetivo de calidad para un equipo de software (o un ingeniero de software individual) es lograr que la ERD_i tienda a 1. Es decir, los errores deben filtrarse antes de que pasen a la siguiente actividad o acción.

CASASEGURA

Establecimiento de un enfoque de métricas

La escena: Oficina de Doug Miller dos días después de la reunión inicial acerca de las métricas de software.

Participantes: Doug Miller (gerente del equipo de ingeniería del software *CasaSegura*), y Vinod Raman y Jamie Lazar, miembros del equipo de ingeniería del software de producto.

La conversación:

Doug: ¿Los dos tuvieron oportunidad de aprender un poco acerca de las métricas de proceso y proyecto?

Vinod y Jamie: [Ambos afirman con la cabeza.]

Doug: Siempre es buena idea establecer metas cuando se adopta alguna métrica. ¿Cuáles son las suyas?

Vinod: Nuestras métricas deben enfocarse en la calidad. De hecho, nuestra meta global es mantener en un mínimo absoluto el número de errores que pasamos de una actividad de ingeniería del software a la siguiente.

Doug: Y estar muy seguros de mantener el número de defectos liberados con el producto tan cerca de cero como sea posible.

Vinod (afirma con la cabeza): Desde luego.

Jamie: Me gusta la ERD como métrica, y creo que podemos usarla para todo el proyecto, pero también conforme nos movemos de una actividad de marco conceptual a la siguiente. Nos alentará a descubrir errores en cada paso.

Vinod: También quiero recopilar el número de horas que pasamos en las revisiones.

Jamie: Y el esfuerzo global que usamos en cada tarea de ingeniería del software.

Doug: Puedes calcular una razón entre revisión y desarrollo... puede ser interesante.

Jamie: También me gustaría rastrear algunos datos de caso de uso. Como la cantidad de esfuerzo requerido para desarrollar un caso de uso, la cantidad de esfuerzo requerido para construir software para implementar un caso de uso y...

Doug (sonríe): Creí que íbamos a mantener esto simple.

Vinod: Deberíamos, pero una vez que entras en este asunto de las métricas, hay muchas cosas interesantes que observar.

Doug: Estoy de acuerdo, pero caminemos antes de correr y apeguémonos a la meta. Limiten los datos a recopilar cinco o seis ítems, y estamos listos para arrancar.

25.4 INTEGRACIÓN DE MÉTRICAS DENTRO DEL PROCESO DE SOFTWARE

La mayoría de los desarrolladores de software todavía no miden y, tristemente, la mayor parte tiene poco deseo de comenzar. Como se apuntó anteriormente en este capítulo, el problema es cultural. Intentar recopilar medidas donde nadie las recopiló en el pasado con frecuencia precipita la resistencia. "¿Por qué necesitamos hacer esto?", pregunta un gerente de proyecto asediado. "No le veo el caso", se queja un profesional saturado de trabajo.

En esta sección se consideran algunos argumentos para fomentar el uso de las métricas del software y se presenta un enfoque para instituir un programa de recopilación de métricas dentro de una organización de ingeniería del software. Pero antes de comenzar, algunas palabras de sabiduría (ahora con más de dos décadas de antigüedad) sugeridas por Grady y Caswell [Gra87]:

> Algunas de las cosas que se describen aquí sonarán muy sencillas. Sin embargo, en realidad, establecer un programa de métricas de software exitoso que abarque a toda la compañía es trabajo duro. Cuando se dice que es necesario esperar al menos tres años antes de que tendencias organizacionales amplias estén disponibles, se tiene alguna idea del ámbito de tal esfuerzo.

Vale la pena hacer caso a la advertencia que sugieren los autores, pero los beneficios de la medición son tan atractivos que el trabajo duro bien lo vale.

25.4.1 Argumentos para métricas de software

Cita:

"Mediante números, administramos las cosas en muchos aspectos de nuestras vidas... Dichos números nos dan comprensión y nos ayudan a guiar nuestras acciones."

Michael Mah y Larry Putnam

¿Por qué es tan importante medir el proceso de ingeniería del software y del producto (software) que da como resultado? La respuesta es relativamente obvia. Si no se mide, no hay forma real de determinar si se está mejorando. Y si no se mejora, se está perdido.

Al solicitar y evaluar las medidas de productividad y calidad, un equipo de software (y su administración) puede establecer metas significativas para mejorar el proceso de software. Anteriormente, en este libro, se indicó que el software es un tema empresarial estratégico para muchas compañías. Si puede mejorarse el proceso a través del cual se desarrolla, puede tenerse como resultado un impacto directo sobre la línea de referencia. Pero para establecer metas a fin de mejorar, debe entenderse el estado actual del desarrollo del software. Por tanto, la medición se utiliza para establecer una línea de referencia del proceso desde el cual puedan valorarse las mejoras.

Los rigores diarios del trabajo del proyecto del software dejan poco tiempo para el pensamiento estratégico. Los gerentes de proyecto de software se preocupan por conflictos más

mundanos (pero igualmente importantes): desarrollar estimaciones de proyecto significativas, producir sistemas de alta calidad, sacar el producto a tiempo. Al usar la medición para establecer una línea de referencia del proyecto, cada uno de estos conflictos se vuelve más manejable. Ya se mencionó que la línea de referencia sirve como base para la estimación. Adicionalmente, la colección de métricas de calidad permite a una organización "afinar" su proceso de software para remover las causas de defectos "menos vitales" que tienen mayor impacto sobre el desarrollo del software.[9]

25.4.2 Establecimiento de una línea de referencia

? ¿Qué es una línea de referencia de métricas y qué beneficio proporciona a un ingeniero del software?

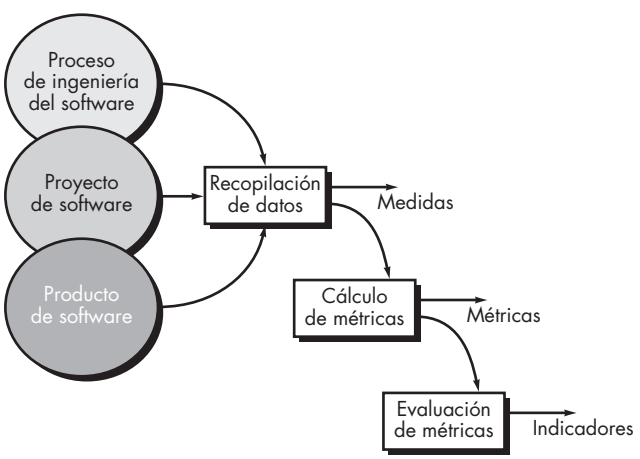
Al establecer una línea de referencia para métricas, pueden obtenerse beneficios en el proceso, el proyecto y el producto (técnico). Aunque la información que se recopila no necesita ser fundamentalmente diferente. Las mismas métricas pueden servir a muchos dominios. La línea de referencia de métricas consiste en los datos recopilados a partir de los proyectos de desarrollo de software y puede ser tan simple como se presenta en la tabla de la figura 25.2 o tan compleja como una base de datos exhaustiva que contiene decenas de medidas de proyecto y las métricas derivadas de ellas.

Para ser un auxiliar efectivo en el mejoramiento del proceso y/o estimación de costo y esfuerzo, los datos de la línea de referencia deben tener los siguientes atributos: 1) deben ser razonablemente precisos y deben evitarse "suposiciones" acerca de los proyectos anteriores, 2) deben recopilarse para tantos proyectos como sea posible, 3) deben ser consistentes (por ejemplo, una línea de código debe interpretarse consistentemente a través de todos los proyectos para los cuales se recopilan datos), 4) las aplicaciones deben ser similares al trabajo que debe estimarse: tiene poco sentido usar una línea de referencia para el trabajo de sistemas de información en lote a fin de estimar una aplicación incrustada en tiempo real.

25.4.3 Recolección, cálculo y evaluación de métricas

En la figura 25.3 se ilustra el proceso que se sigue para establecer una línea de referencia de métricas. De manera ideal, los datos necesarios para establecer una línea de referencia se recolectaron de una forma continua. Tristemente, éste rara vez es el caso. Por tanto, la recopilación de datos requiere una investigación histórica de los proyectos anteriores a fin de reconstruir los datos requeridos. Una vez recopiladas las medidas (sin duda, el paso más difícil), es posible el

FIGURA 25.3
Proceso de recopilación de métricas del software



9 Estas ideas se formalizaron en un enfoque llamado *aseguramiento estadístico de la calidad del software*.

cálculo de métricas. Dependiendo de la envergadura de las medidas recopiladas, las métricas pueden abarcar un amplio rango de métricas orientadas a aplicaciones (por ejemplo, LOC, PF, orientada a objeto, *webapp*), así como otras orientadas a calidad y proyecto. Finalmente, las métricas deben evaluarse y aplicarse durante la estimación, el trabajo técnico, el control del proyecto y la mejora del proceso. La evaluación de métricas se enfoca en las razones subyacentes para los resultados obtenidos y produce un conjunto de indicadores que guían al proyecto o al proceso.

PUNTO CLAVE

Los datos de métricas de línea de referencia deben recopilarse a partir de un gran muestreo representativo de proyectos de software anteriores.

25.5 MÉTRICAS PARA ORGANIZACIONES PEQUEÑAS

La gran mayoría de las organizaciones de desarrollo de software tienen menos de 20 personas en sus departamentos de software. No es razonable, y en la mayoría de los casos es irreal, esperar que tales organizaciones desarrollen programas de métricas de software exhaustivos. Sin embargo, sí es razonable sugerir que las organizaciones de software de todos los tamaños midan y luego usen las métricas resultantes para ayudar a mejorar su proceso de software local y la calidad y temporalidad de sus productos.

CONSEJO

Si apenas comienza a recopilar datos de métricas, recuerde mantenerlas simples. Si se entierra debajo de datos, sus esfuerzos de métricas fallarán.

Un enfoque de sentido común para la implementación de cualquier actividad de software relacionada con el software es: mantenerlo simple, personalizar para satisfacer las necesidades locales y asegurarse de que agrega valor. En los siguientes párrafos se examina la manera como se relacionan dichos lineamientos con las métricas para tiendas pequeñas.[10]

? ¿Cómo debe derivarse un conjunto de métricas de software "simples"?

"Mantenerlo simple" es un lineamiento que funciona razonablemente bien en muchas actividades. Pero, ¿cómo debe derivarse un conjunto de métricas de software "simples" que aun así proporcionen valor y cómo puede asegurarse de que dichas métricas simples satisfarán las necesidades específicas de su organización de software? Puede comenzarse con un enfoque en los resultados, no en la medición. Se consulta al grupo de software para definir un solo objetivo que requiera mejora. Por ejemplo, "reducir el tiempo para evaluar e implementar las peticiones de cambio". Una organización pequeña puede seleccionar el siguiente conjunto de medidas de fácil recolección:

- Tiempo transcurrido (horas o días) desde el momento en el que se hace una petición hasta que la evaluación está completa, t_{cola}.
- Esfuerzo (persona-horas) para realizar la evaluación, W_{eval}.
- Tiempo transcurrido (horas o días) desde la conclusión de la evaluación hasta la asignación de la orden de cambio al personal, t_{eval}.
- Esfuerzo requerido (persona-horas) para hacer el cambio, W_{cambio}.
- Tiempo requerido (horas o días) para hacer el cambio, t_{cambio}.
- Errores descubiertos durante el trabajo para hacer el cambio, E_{cambio}.
- Defectos descubiertos después de liberar el cambio al cliente base, D_{cambio}.

Una vez recopiladas dichas medidas para algunas peticiones de cambio, es posible calcular el tiempo transcurrido total desde la petición de cambio hasta la implementación del mismo y el porcentaje del tiempo transcurrido absorbido por la cola inicial, la evaluación y la asignación de cambio y su implementación. De igual modo, puede determinarse el porcentaje de esfuerzo requerido para evaluación e implementación. Dichas métricas pueden valorarse en el contexto de datos de calidad, E_{cambio} y D_{cambio}. Los porcentajes brindan comprensión acerca de dónde se frena el proceso de petición de cambio y pueden conducir a definir medidas de mejora del pro-

10 Este análisis es igualmente relevante para los equipos de software que adoptan un proceso de desarrollo de software ágil (capítulo 3).

ceso para reducir t_{cola}, W_{eval}, t_{eval}, W_{cambio} y/o E_{cambio}. Además, la eficiencia de remoción de defecto puede calcularse como

$$\text{ERD} = \frac{E_{cambio}}{E_{cambio} + D_{cambio}}$$

La ERD puede compararse con el tiempo transcurrido y con el esfuerzo total para determinar el impacto de las actividades de aseguramiento de la calidad sobre el tiempo y el esfuerzo requerido para hacer un cambio.

Para grupos pequeños, el costo que representa recopilar medidas y calcular métricas varía de 3 a 8 por ciento del presupuesto del proyecto durante la fase de aprendizaje, y luego disminuye a menos de 1 por ciento del presupuesto del proyecto después de que los ingenieros del software y los gerentes del proyecto se familiarizan con el programa de métricas [Gra99]. Dichos costos pueden mostrar un rendimiento sustancial sobre la inversión si la comprensión de los datos de las métricas conducen a una mejora de proceso significativa para la organización del software.

25.6 Establecimiento de un programa de métricas del software

El Software Engineering Institute (SEI) desarrolló un manual muy completo [Par96b] para establecer un programa de métrica de software "dirigido hacia la meta". El manual sugiere los siguientes pasos:

1. Identificar las metas empresariales.
2. Identificar lo que se quiere conocer o aprender.
3. Identificar las submetas.
4. Identificar las entidades y atributos relacionados con las submetas.
5. Formalizar las metas de medición.
6. Identificar preguntas cuantificables y los indicadores relacionados que se usarán para ayudar a lograr las metas de medición.
7. Identificar los elementos de datos que se recopilarán para construir los indicadores que ayuden a responder las preguntas.
8. Definir las medidas que se van a usar y hacer operativas estas definiciones.
9. Identificar las acciones que se tomarán para implementar las medidas.
10. Preparar un plan para la implantación de las medidas.

WebRef

Puede descargarse de **www.sei.cmu.edu** un *manual para medición de software dirigido hacia las metas.*

Un estudio detallado de estos pasos se aprecia mejor en el manual del SEI. Sin embargo, vale la pena un breve panorama de los puntos clave.

Puesto que el software apoya las funciones empresariales, diferencia los sistemas o productos basados en computadora y actúa como un producto en sí mismo, las metas definidas por la empresa casi siempre pueden rastrearse en las metas específicas de ingeniería del software. Por ejemplo, considere el producto *CasaSegura*. Al trabajar como equipo, los gerentes de ingeniería del software y de la empresa desarrollan una lista de metas empresariales prioritarias:

PUNTO CLAVE

Las métricas del software que elija deben activarse mediante las metas empresariales y técnicas que quiera lograr.

1. Mejorar la satisfacción de los clientes con los productos.
2. Hacer los productos más fáciles de usar.
3. Reducir el tiempo que tarda en salir un nuevo producto al mercado.
4. Facilitar el apoyo a los productos.
5. Mejorar la rentabilidad global.

La organización de software examina cada meta empresarial y pregunta: ¿qué actividades manejamos, ejecutamos o apoyamos y qué hacemos para mejorar dichas actividades? Para responder estas preguntas, el SEI recomienda la creación de una "lista de entidad-pregunta" en la que se anotan todas las cosas (entidades) existentes dentro del proceso de software que se gestionan o que influyen en la organización de software. Los ejemplos de entidades incluyen recursos de desarrollo, productos operativos, código fuente, casos de prueba, peticiones de cambio, tareas de ingeniería del software y calendarios. Para cada entidad mencionada, el personal de software desarrolla un conjunto de preguntas que valoran las características cuantitativas de la entidad (por ejemplo, tamaño, costo, tiempo para desarrollar). Las preguntas derivadas como consecuencia de la creación de una lista de entidad-pregunta conducen a la derivación de un conjunto de submetas que se relacionan directamente con las entidades creadas y con las actividades realizadas como parte del proceso de software.

Considere la cuarta meta: "Facilitar el apoyo a los productos". La siguiente lista de preguntas puede derivarse de esta meta [Par96b]:

- ¿Las peticiones de cambio de los clientes contienen la información que se requiere para evaluar adecuadamente el cambio y luego implementarlo en una forma oportuna?
- ¿Cuán grande es la acumulación de trabajo debida a la petición de cambio?
- ¿El tiempo de respuesta para corregir errores es aceptable con base en las necesidades del cliente?
- ¿Se sigue el proceso de control de cambio (capítulo 22)?
- ¿Los cambios de alta prioridad se implementan en forma oportuna?

A partir de estas preguntas, la organización de software puede derivar la siguiente submeta: *mejorar el rendimiento del proceso de gestión del cambio*. Entonces, se identifican las entidades y atributos del proceso de software que son relevantes para la submeta y se delinean las metas de medición asociadas con ellas.

El SEI [Par96b] proporciona lineamientos detallados para los pasos del 6 al 10 de su enfoque de medición dirigido a metas. En esencia, las metas de medición se refinan en preguntas que se desglosan aún más en entidades y atributos que entonces se refinan en métricas.

INFORMACIÓN

Establecimiento de un programa de métricas

El Software Productivity Center **(www.spc.ca)** sugiere un enfoque de ocho pasos para establecer un programa de métricas dentro de una organización de software, que puedan usarse como alternativa al enfoque SEI descrito en la sección 25.6. Su enfoque se resume en este recuadro.

1. Comprender el proceso de software existente.
   Identificar actividades de marco conceptual (capítulo 2).
   Describir la información de entrada para cada actividad.
   Definir las tareas asociadas con cada actividad.
   Anotar las funciones de aseguramiento de calidad.
   Mencionar los productos operativos que se producen.
2. Definir las metas por lograr mediante el establecimiento de un programa de métricas.
   Ejemplos: mejorar la precisión de la estimación, mejorar la calidad del producto.
3. Identificar las métricas requeridas para lograr las metas.
   Definir preguntas, por ejemplo, *¿cuántos errores encontrados en una actividad de marco conceptual pueden rastrearse en la actividad de marco conceptual precedente?*
   Crear medidas y métricas que ayudarán a responder dichas preguntas.
4. Identificar las medidas y métricas que se van a recopilar y calcular.
5. Establecer un proceso de recolección de medición al responder estas preguntas:
   ¿Cuál es la fuente de las mediciones?
   ¿Pueden usarse herramientas para recopilar los datos?
   ¿Quién es responsable de recopilar los datos?
   ¿Cuándo se recopilan y registran datos?
   ¿Cómo se almacenan los datos?

¿Qué mecanismos de validación se usan para garantizar que los datos son correctos?

6. Adquirir herramientas adecuadas para auxiliar en la recopilación y valoración.
7. Establecer una base de datos de métricas.
   Establecer la sofisticación relativa de la base de datos.
   Explorar el uso de herramientas relacionadas (por ejemplo, un repositorio SCM, capítulo 26).
   Evaluar los productos de base de datos existentes.
8. Definir mecanismos de retroalimentación adecuados.
   ¿Quién requiere información de métricas actuales?
   ¿Cómo se entregará la información?
   ¿Cuál es el formato de la información?

Una descripción considerablemente más detallada de estos ocho pasos puede descargarse de **www.spc.ca/resources/metrics/**

25.7 Resumen

La medición permite a gerentes y profesionales mejorar el proceso de software; auxiliar en la planificación, rastreo y control de proyectos de software y valorar la calidad del producto (software) que se elabora. Las medidas de atributos específicos del proceso, proyecto y producto se usan para calcular las métricas de software. Dichas métricas pueden analizarse para proporcionar indicadores que guíen las acciones administrativas y técnicas.

Las métricas de proceso permiten que una organización adopte una visión estratégica al proporcionar comprensión acerca de la efectividad de un proceso de software. Las métricas de proyecto son tácticas. Permiten que un gerente de proyecto adapte el flujo de trabajo del proyecto y el enfoque técnico en tiempo real.

Las métricas orientadas a tamaño y a función se usan a lo largo de la industria. Las primeras usan la línea de código como un factor de normalización para otras medidas, como persona-meses o defectos. El punto de función se deriva de las medidas del dominio de información y de una valoración subjetiva de la complejidad del problema. Además, pueden usarse métricas orientadas a objeto y a *webapp*.

Las métricas de calidad del software, como las métricas de productividad, se enfocan en el proceso, el proyecto y el producto. Al desarrollar y analizar una línea de referencia de métricas para la calidad, una organización puede corregir aquellas áreas del proceso de software que sean la causa de los defectos de software.

La medición da como resultado un cambio cultural. La recopilación de datos, cálculo de métricas y análisis de métricas son los tres pasos que deben implementarse para comenzar un programa de métricas. En general, un enfoque dirigido a metas ayuda a una organización a enfocarse en las métricas correctas para su empresa. Al crear una línea de referencia de métricas, una base de datos que contenga mediciones de proceso y producto, los ingenieros de software y sus gerentes pueden obtener mejor comprensión del trabajo que hacen y del producto que elaboran.

Problemas y puntos por evaluar

25.1. Describa con sus palabras la diferencia entre métricas de proceso y de proyecto.

25.2. ¿Por qué algunas métricas de software deben mantenerse "privadas"? Ofrezca cinco ejemplos de tres métricas que deban ser privadas. Brinde ejemplos de tres métricas que deban ser públicas.

25.3. ¿Qué es una media indirecta y por qué tales mediciones son comunes en el trabajo con métricas de software?

25.4. Grady sugiere una etiqueta para las métricas de software. ¿Puede agregar tres reglas más a las anotadas en la sección 25.1.1?

25.5. El equipo A encontró 342 errores durante el proceso de ingeniería del software antes de la liberación. El equipo B encontró 184 errores. ¿Qué medidas adicionales tendrían que realizarse a los proyectos A y B para determinar cuál de los equipos eliminó errores de manera más eficiente? ¿Qué métricas propondría para ayudar a realizar esta determinación? ¿Qué datos históricos pueden ser útiles?

25.6. Presente un argumento contra las líneas de código como medida para la productividad del software. ¿Su caso se sostendría cuando se consideren decenas o cientos de proyectos?

25.7. Calcule el valor de punto de función para un proyecto con las siguientes características de dominio de información:

Número de entradas de usuario: 32
Número de salidas de usuario: 60
Número de consultas de usuario: 24
Número de archivos: 8
Número de interfaces externas: 2

Suponga que todos los valores de ajuste de complejidad son promedios. Use el algoritmo mencionado en el capítulo 23.

25.8. Con la tabla que se presenta en la sección 25.2.3, plantee un argumento contra el uso de lenguaje ensamblador con base en la funcionalidad que entrega por enunciado de código. Nuevamente con la tabla, analice por qué C++ representaría una mejor alternativa que C.

25.9. El software que se usa para controlar una fotocopiadora requiere 32 000 líneas de C y 4 200 líneas de Smalltalk. Estime el número de puntos de función para el software dentro de la fotocopiadora.

25.10. Un equipo de ingeniería web construye una *webapp* de comercio electrónico que contiene 145 páginas individuales. De éstas, 65 son dinámicas, es decir, se generan internamente con base en entrada del usuario final. ¿Cuál es el índice de personalización para esta aplicación?

25.11. Una *webapp* y su entorno de apoyo no están completamente fortificados contra ataques. Los ingenieros web estiman que la probabilidad de repeler un ataque es de sólo 30 por ciento. El sistema no contiene información sensible o controvertida, de modo que la probabilidad de amenaza es de 25 por ciento. ¿Cuál es la integridad de la *webapp*?

25.12. En la conclusión de un proyecto, se determinó que se encontraron 30 errores durante la actividad de modelado y 12 durante la actividad de construcción, que fueron rastreables en errores que no se descubrieron en la actividad de modelado. ¿Cuál es la ERD para la actividad de modelado?

25.13. Un equipo de software entrega un incremento de software a los usuarios finales. Éstos descubren ocho defectos durante el primer mes de uso. Antes de la liberación, el equipo de software encontró 242 errores durante las revisiones técnicas formales y todas las tareas de prueba. ¿Cuál es la ERD global para el proyecto después de un mes de uso?

Lecturas y fuentes de información adicionales

El mejoramiento del proceso de software (MPS) recibió una cantidad significativa de atención durante las dos décadas pasadas. Dado que la medición y las métricas de software son clave para el mejoramiento exitoso del proceso de software, muchos libros acerca de MPS también estudian métricas. Rico (*ROI of Software Process Improvement*, J. Ross Publishing, 2004) ofrece un análisis a profundidad del MPS y de las métricas que pueden ayudar a una organización a lograrlo. Ebert *et al.* (*Best Practices in Software Measurement*, Springer, 2004) abordan el uso de medición dentro del contexto de estándares ISO y CMMI. Kan (*Metrics and Models in Software Quality Engineering*, 2a. ed., Addison-Wesley, 2002) presenta una colección de métricas relevantes.

Ebert y Dumke (*Software Measurement*, Springer, 2007) proporcionan un tratamiento útil de medición y métrica para proyectos IT. McGarry *et al.* (*Practical Software Measurement*, Addison-Wesley, 2001) presentan consejos profundos para valorar el proceso de software. Una valiosa colección de ensayos fueron editados por Haug y colegas (*Software Process Improvement: Metrics, Measurement, and Process Modeling,* Springer-Verlag, 2001). Florac y Carlton (*Measuring the Software Process,* Addison-Wesley, 1999) y Fenton y Pfleeger (*Software Metrics: A Rigorous and Practical Approach,* Revised, Brooks/Cole Publishers, 1998) estudian cómo pueden usarse las métricas del software para proporcionar los indicadores necesarios a fin de mejorar el proceso de software.

Laird y Brennan (*Software Measurement and Estimation*, Wiley-IEEE Computer Society Press, 2006) y Goodman (*Software Metrics: Best practices for Successful IT Management*, Rothstein Associates, Inc., 2004), analizan el uso de métricas de software para administración y estimación de proyectos. Putnam y Myers (*Five Core Metrics*, Dorset House, 2003) recurren a una base de datos de más de 6 000 proyectos de software para demostrar cómo pueden usarse cinco métricas centrales (tiempo, esfuerzo, tamaño, confiabilidad y productividad de proceso) para controlar los proyectos de software. Maxwell (*Applied Statistics for Software Managers*, Prentice-Hall, 2003), presenta técnicas para analizar datos de proyecto de software. Munson (*Software Engineering Measurement*, Auerbach, 2003) estudia una amplia matriz de conflictos de medición en ingeniería del software. Jones (*Software Assessments, Benchmarks and Best Practices*, Addison-Wesley, 2000) describe tanto la medición cuantitativa como los factores cualitativos que ayudan a una organización a valorar sus procesos y prácticas de software.

La medición del punto de función se ha convertido en una técnica ampliamente usada en muchas áreas del trabajo en ingeniería del software. Parthasarathy (*Practical Software Estimation: Function Point Methods for Insourced and Outsourced Projects*, Addison-Wesley, 2007) ofrece una guía exhaustiva. Garmus y Herron (*Function Point Analysis: Measurement Practices for Successful Software Projects*, Addison-Wesley, 2000) analizan las métricas de proceso con énfasis en el análisis de punto de función.

Relativamente poco se ha publicado acerca de las métricas para trabajo en ingeniería web. Sin embargo, Kaushik *(Web Analytics: An Hour a Day,* Sybex, 2007), Stern (*Web Metrics: Proven Methods for Measuring Web Site Success,* Wiley, 2002), Inan y Kean (*Measuring the Success of Your Website,* Longman, 2002), y Nobles y Grady (*Web Site Analysis and Reporting,* Premier Press, 2001) abordan las métricas web desde una perspectiva empresarial y de mercadotecnia.

El IEEE (*Symposium on Software Metrics*, publicación anual) resume las investigaciones más recientes en el área de métricas. En internet, está disponible una gran variedad de fuentes de información acerca de las métricas de procesos y de proyectos. Una lista actualizada de referencias en la World Wide Web que son relevantes para las métricas de procesos y de proyectos puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

CAPÍTULO

26

ESTIMACIÓN PARA PROYECTOS DE SOFTWARE

La administración de los proyectos de software comienza con un conjunto de actividades que de manera colectiva se llaman *planificación de proyecto*. Antes de que el proyecto pueda comenzar, el equipo de software debe estimar el trabajo que se va a realizar, los recursos que se requerirán y el tiempo que transcurrirá de principio a fin. Una vez completadas dichas actividades, el equipo de software debe establecer un calendario del proyecto que defina las tareas e hitos de la ingeniería de software, que identifique quién es responsable de realizar cada tarea y especifique las dependencias entre tareas que puedan imponer una fuerte demora sobre el avance.

En una excelente guía para "sobrevivir al proyecto de software", Steve McConnell [McC98] presenta una visión del mundo real de la planificación del proyecto:

> Muchos trabajadores técnicos preferirían realizar trabajo técnico en lugar de pasar tiempo planificando. Muchos gerentes técnicos no tienen suficiente capacitación en administración técnica como para sentirse seguros de que su planificación mejorará el resultado de un proyecto. Dado que ninguna parte quiere hacer planes, con frecuencia nunca se hacen.
>
> Pero fallar en la planificación es uno de los errores más cruciales que un proyecto puede tener... la planificación efectiva es necesaria para resolver problemas corriente arriba [tempranamente en el proyecto] a bajo costo, en lugar de corriente abajo [tardíamente en el proyecto] a alto costo. El proyecto promedio emplea *80 por ciento* de su tiempo en "poner al día": corregir errores que se cometieron anteriormente en el proyecto.

UNA MIRADA RÁPIDA

¿Qué es? Se establece una necesidad real para el software; los participantes están a bordo, los ingenieros de software están listos para comenzar y el proyecto está a punto de iniciar. Pero, ¿cómo se procede? La planificación de proyectos de software abarca cinco grandes actividades: estimación, calendarización, análisis de riesgos, planificación de gestión de la calidad y planificación de gestión del cambio. En el contexto de este capítulo, sólo se considera la estimación, el intento por determinar cuánto dinero, esfuerzo, recursos y tiempo tomará construir un sistema o producto específico basado en software.

¿Quién lo hace? Los gerentes de proyecto de software, con la información solicitada a los participantes del proyecto y datos de métricas de software recopiladas de proyectos anteriores.

¿Por qué es importante? ¿Construiría una casa sin saber más o menos cuánto gastará, las tareas que necesita realizar y el cronograma para el trabajo que se va a realizar? Desde luego que no y, dado que la mayoría de los sistemas y productos basados en computadora cuestan considerablemente más que construir una gran casa, parece razonable realizar una estimación antes de comenzar a crear el software.

¿Cuáles son los pasos? La estimación comienza con una descripción del ámbito del problema. Luego éste se descompone en un conjunto de problemas más pequeños y cada uno de éstos se estima, usando como guías datos históricos y experiencia. La complejidad y el riesgo del problema se consideran antes de realizar una estimación final.

¿Cuál es el producto final? La generación de una tabla simple que delinea las tareas que se van a realizar, las funciones por implementar y el costo, esfuerzo y tiempo involucrados para cada tarea.

¿Cómo me aseguro de que lo hice bien? Eso es difícil, porque en realidad no se sabe hasta que el proyecto se completa. Sin embargo, si se tiene experiencia y se sigue un enfoque sistemático, si genera estimaciones usando datos históricos sólidos, si crea puntos de datos de estimación con al menos dos métodos diferentes, si establece un calendario realista y continuamente lo adapta conforme el proyecto avanza, puede estar seguro de que está haciendo su mejor esfuerzo.

McConnell argumenta que todo equipo puede encontrar el tiempo para planificar (y adaptar el plan a lo largo del proyecto), tomando simplemente un pequeño porcentaje del tiempo que se hubiera empleado en rehacer lo que ocurre porque no se llevó a cabo la planificación.

26.1 Observaciones acerca de las estimaciones

La planificación requiere adoptar un compromiso inicial, aun cuando es probable que este "compromiso" resulte erróneo. Siempre que se hacen estimaciones se mira hacia el futuro y se acepta cierto grado de incertidumbre habitual. En palabras de Frederick Brooks [Bro95]:

> ... nuestras técnicas de estimación están pobremente desarrolladas. Más seriamente, reflejan una suposición no expresada que es muy falsa: que todo irá bien [...] puesto que se tiene incertidumbre acerca de las estimaciones, los gerentes de software con frecuencia carecen de la tenacidad cortés para hacer lo que la gente espera de un buen producto.

Cita:

"Los buenos enfoques de estimación y los datos históricos sólidos ofrecen la mejor esperanza de que realmente se triunfará sobre demandas imposibles."

Caper Jones

Aunque las estimaciones son tanto un arte como una ciencia, esta importante acción no necesita realizarse en forma azarosa. Existen técnicas útiles para estimación de tiempo y esfuerzo. Las métricas de proceso y proyecto pueden proporcionar perspectiva histórica y poderosa entrada para la generación de estimaciones cuantitativas. Las experiencias pasadas (de todas las personas involucradas) pueden auxiliar sin medida conforme se desarrollen y revisen las estimaciones. Puesto que éstas tienden los cimientos de todas las acciones de planificación del proyecto, y la planificación del proyecto ofrece el mapa de caminos para la ingeniería de software exitosa, estaríamos mal aconsejados si nos embarcáramos sin ella.

La estimación de recursos, costo y calendario para un esfuerzo de ingeniería de software requiere experiencia, acceso a buena información histórica (métricas) y coraje para comprometerse con las predicciones cuantitativas cuando todo lo que existe es información cualitativa. La estimación porta un riesgo inherente,[1] y éste conduce a incertidumbre.

La *complejidad del proyecto* tiene un fuerte efecto sobre la incertidumbre inherente a la planificación. Sin embargo, la complejidad es una medida relativa que es afectada por la familiaridad con el esfuerzo pasado. El profesional que por primera vez desarrolla una sofisticada aplicación de comercio electrónico puede considerarla excesivamente compleja. Sin embargo, un equipo de ingeniería web que desarrolla su décima *webapp* de comercio electrónico considerará tal trabajo común y corriente. Se han propuesto [Zus97] algunas medidas cuantitativas de complejidad de software. Éstas se aplican en el nivel de diseño o código y, por tanto, son difíciles de usar durante la planificación del software (antes de una salida de diseño y código). No obstante, es posible establecer otras valoraciones de complejidad más subjetivas (por ejemplo, los factores de ajuste de complejidad de punto de función descritos en el capítulo 23) en las primeras etapas del proceso de planificación.

PUNTO CLAVE

Complejidad del proyecto, tamaño del proyecto y grado de incertidumbre estructural afectan la confiabilidad de las estimaciones.

El *tamaño del proyecto* es otro factor importante que puede afectar la precisión y la eficacia de las estimaciones. Conforme aumenta el tamaño, la interdependencia entre varios elementos del software crece rápidamente.[2] La descomposición del problema, un importante enfoque de la estimación, se vuelve más difícil porque el refinamiento de los elementos del problema todavía puede ser formidable. Para parafrasear la ley de Murphy: "lo que puede salir mal, saldrá mal", y si hay más cosas que pueden fallar, más cosas fallarán.

El *grado de incertidumbre estructural* también tiene un efecto sobre el riesgo de estimación. En este contexto, estructura se refiere al grado en el cual se solidificaron los requisitos, la faci-

1 En el capítulo 28 se presentan técnicas sistemáticas para el análisis de riesgos.

2 Con frecuencia, el tamaño aumenta debido al "lento avance del ámbito", que ocurre cuando cambian los requisitos del problema. El aumento en el tamaño del proyecto puede tener un impacto geométrico sobre el costo y el calendario del proyecto (Michael Mah, comunicación personal).

lidad con la que se dividieron las funciones y la naturaleza jerárquica de la información que debe procesarse.

Cita:

"Es distintivo de una mente instruida descansar satisfecha con el grado de precisión que la naturaleza del sujeto admite, y no buscar exactitud cuando sólo es posible una aproximación a la verdad."

Aristóteles

La disponibilidad de información histórica tiene fuerte influencia sobre el riesgo de estimación. Al mirar hacia atrás, puede emular las cosas que funcionaron y mejorar las áreas donde surgieron problemas. Cuando hay disponibles métricas de software exhaustivas (capítulo 25) para proyectos anteriores, pueden hacerse estimaciones con mayor precisión, así como establecerse calendarios para evitar las dificultades pasadas y el riesgo global se reduce.

El riesgo de estimación se mide por el grado de incertidumbre en las estimaciones cuantitativas establecidas para recursos, costo y calendario. Si el ámbito del proyecto se comprende pobremente o si los requisitos del proyecto están sujetos a cambio, la incertidumbre y el riesgo en la estimación se vuelven peligrosamente altos. Como planificador, usted y el cliente deben reconocer que la variabilidad en los requisitos del software significa inestabilidad en costo y calendario.

Sin embargo, no debe volverse obsesivo acerca de la estimación. Los modernos enfoques de ingeniería de software (por ejemplo, modelos de proceso evolutivos) toman una visión iterativa del desarrollo. En tales enfoques es posible, aunque no siempre políticamente aceptable, revisitar la estimación (conforme se conoce más información) y revisarla cuando el cliente hace cambios a los requisitos.

26.2 El proceso de planificación del proyecto

CONSEJO

Mientas más conozca, mejor estimará. En consecuencia, actualice sus estimaciones conforme avance el proyecto.

El objetivo de la planificación del proyecto de software es proporcionar un marco conceptual que permita al gerente hacer estimaciones razonables de recursos, costo y calendario. Además, las estimaciones deben intentar definir los escenarios de mejor caso y peor caso, de modo que los resultados del proyecto puedan acotarse. Aunque hay un grado inherente de incertidumbre, el equipo de software se embarca en un plan que se haya establecido como consecuencia de dichas tareas. Por tanto, el plan debe adaptarse y actualizarse conforme avanza el proyecto. En las siguientes secciones se estudia cada una de las acciones asociadas con la planificación del proyecto de software.

Conjunto de tareas

Conjunto de tareas para planificación de proyectos

1. Establecer ámbito del proyecto.
2. Determinar la factibilidad.
3. Analizar los riesgos (capítulo 28).
4. Definir recursos requeridos.
   a) Determinar recursos humanos requeridos.
   b) Definir recursos de software reutilizables.
   c) Identificar recursos ambientales.
5. Estimar costo y esfuerzo.
   a) Descomponer el problema.
   b) Desarrollar dos o más estimaciones usando tamaño, puntos de función, tareas de proceso o casos de uso.
   c) Reconciliar las estimaciones.
6. Desarrollar un calendario del proyecto (capítulo 27).
   a) Establecer un conjunto de tareas significativas.
   b) Definir una red de tareas.
   c) Usar herramientas de calendarización para desarrollar un cronograma.
   d) Definir mecanismos de seguimiento de calendario.

26.3 Ámbito y factibilidad del software

El *ámbito del software* describe las funciones y características que se entregan a los usuarios finales; los datos que son entrada y salida; el "contenido" que se presenta a los usuarios como consecuencia de usar el software y el desempeño, las restricciones, las interfaces y la confiabilidad que se *ligan* al sistema. El ámbito se define usando una de dos técnicas:

1. Una descripción narrativa del ámbito del software se desarrolla después de la comunicación con todos los participantes.
2. Los usuarios finales desarrollan un conjunto de casos de uso.[3]

La factibilidad del proyecto es importante, pero una consideración de la necesidad empresarial lo es incluso más. No es bueno construir un sistema o producto de alta tecnología que nadie quiere.

Las funciones descritas en el enunciado del ámbito (o dentro de los casos de uso) se evalúan y en algunos casos se desglosan para proporcionar más detalle, previamente al comienzo de la estimación. Puesto que las estimaciones de costo y calendario están funcionalmente orientadas, con frecuencia es útil cierto grado de descomposición. Las consideraciones de rendimiento abarcan los requisitos de procesamiento y de tiempo de respuesta. Las restricciones identifican los límites colocados en el software por parte de hardware externo, memoria disponible u otros sistemas existentes.

Una vez identificado el ámbito (con la concurrencia del cliente), es razonable preguntar: ¿puede construirse software para satisfacer este ámbito? ¿El proyecto es factible? Con mucha frecuencia, los ingenieros de software rápidamente desechan estas preguntas (o son presionados para desecharlas por gerentes impacientes u otros participantes), sólo para encontrarse enlodados en un proyecto que está condenado desde el principio. Putnam y Myers [Put97a] abordan este conflicto cuando escriben:

> [No] todo lo imaginable es factible, ni siquiera en software, tan evanescente como pueda aparecer a los profanos. Por el contrario, la factibilidad del software tiene cuatro dimensiones sólidas: *Tecnología*: ¿Un proyecto es técnicamente factible? ¿Está dentro del estado del arte? ¿Pueden reducirse los defectos en un nivel que coincida con las necesidades de la aplicación? *Finanzas*: ¿Es financieramente factible? ¿El desarrollo puede completarse a un costo que la organización de software, su cliente o el mercado puede pagar? *Tiempo*: ¿El tiempo del proyecto para llegar al mercado vencerá a la competencia? *Recursos*: ¿La organización tiene los recursos necesarios para triunfar?

Putnam y Myers sugieren de manera correcta que establecer el ámbito no es suficiente. Una vez comprendido éste, debe trabajarse para determinar si puede hacerse dentro de las dimensiones recién anotadas. Ésta es una parte vital, aunque con frecuencia pasada por alto, del proceso de estimación.

26.4 Recursos

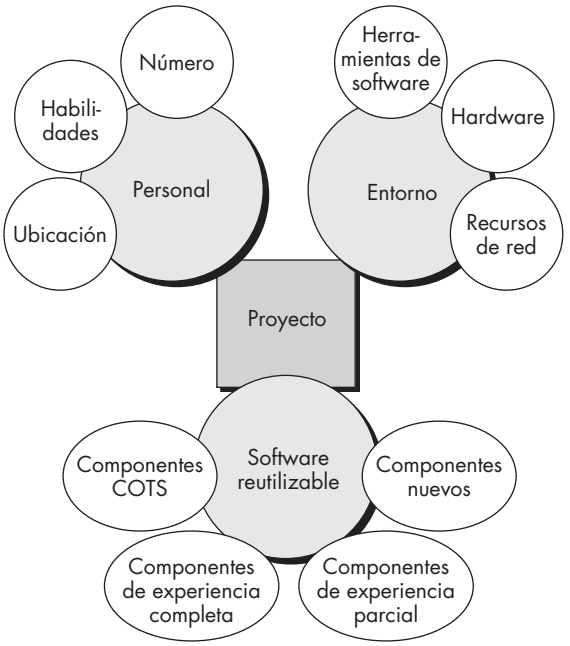
La segunda tarea en la planificación es la estimación de los recursos requeridos para lograr el esfuerzo de desarrollo del software. La figura 26.1 muestra las tres principales categorías de los recursos de la ingeniería de software: personal, componentes de software reutilizables y entorno de desarrollo (herramientas de hardware y software). Cada recurso se especifica con cuatro características: descripción del recurso, un enunciado de disponibilidad, momento en el que se requerirá el recurso y duración del tiempo que se aplicará el recurso. Las últimas dos características pueden verse como una *ventana temporal*. La disponibilidad del recurso para una ventana específica debe establecerse en el tiempo práctico más temprano.

26.4.1 Recursos humanos

El planificador comienza por evaluar el ámbito del software y seleccionando las habilidades requeridas para completar el desarrollo. Se especifican tanto la posición organizacional (por ejemplo, gerente, ingeniero de software ejecutivo) como la especialidad (por ejemplo, telecomunicaciones, base de datos, cliente-servidor). Para proyectos relativamente pequeños (algunos persona-meses), un solo individuo puede realizar todas las tareas de ingeniería de software y

3 Los casos de uso se estudiaron con detalle a lo largo de la parte 2 de este libro. Un caso de uso es una descripción basada en el escenario de la interacción del usuario con el software desde el punto de vista del usuario.

FIGURA 26.1
Recursos del proyecto



consultar especialistas según lo requiera. Para proyectos más grandes, el equipo de software puede dispersarse geográficamente a través de algunas ubicaciones diferentes. Por tanto, se debe especificar la ubicación de cada recurso humano.

El número de personas requeridas para un proyecto de software puede determinarse sólo después de hacer una estimación del esfuerzo de desarrollo (por ejemplo, persona-meses). Más adelante, en este capítulo, se estudian técnicas para estimar el esfuerzo.

26.4.2 Recursos de software reutilizables

La ingeniería de software basada en componentes (ISBC)[4] pone el énfasis en la reusabilidad; es decir, en la creación y reutilización de bloques constructores de software. Tales bloques constructores, con frecuencia llamados *componentes*, deben catalogarse para facilitar su referencia, estandarizarse para facilitar su aplicación y validarse para facilitar su integración. Bennatan [Ben00] sugiere cuatro categorías de recursos de software que deben considerarse conforme avanza la planificación:

Nunca olvide que integrar varios componentes reutilizables puede ser un reto considerable. Peor aún, el problema de la integración sale a la superficie conforme varios componentes se actualizan.

Componentes comerciales. Es el software existente que puede adquirirse de una tercera parte o de un proyecto anterior. Los componentes COTS (por las siglas en inglés de "anaqueles comerciales") se compran de una tercera parte, están listos para su uso en el proyecto actual y están completamente validados.

Componentes de experiencia completa. Son especificaciones, diseños, código o datos de prueba existentes, desarrollados para proyectos anteriores que son similares al software que se va a construir para el proyecto en cuestión. Los miembros del equipo de software tienen mucha experiencia en el área de aplicación representada por dichos componentes. Por tanto, las modificaciones requeridas para los componentes de experiencia completa tendrán un riesgo relativamente bajo.

4 La ISBC se consideró en el capítulo 10.

Componentes de experiencia parcial. Son especificaciones, diseños, código o datos de prueba existentes, desarrollados para proyectos anteriores que se relacionan con el software que se va a construir para el proyecto en cuestión, pero que requerirán modificación sustancial. Los miembros del equipo de software sólo tienen experiencia limitada en el área de aplicación especificada por dichos componentes. Por tanto, las modificaciones requeridas por los componentes de experiencia parcial entrañan un buen grado de riesgo.

Componentes nuevos. Son componentes de software que el equipo de software debe construir específicamente para las necesidades del proyecto en cuestión.

Irónicamente, los componentes de software reutilizables con frecuencia se desprecian durante la planificación, sólo para convertirse en una preocupación suprema más tarde, durante el avance del software. Es mejor especificar tempranamente los requisitos de recurso del software. De esta forma puede realizarse una evaluación técnica de las alternativas, así como la adquisición oportuna.

26.4.3 Recursos ambientales

El entorno que soporta a un proyecto de software, con frecuencia llamado *entorno de ingeniería de software* (EIS), incorpora hardware y software. El hardware proporciona una plataforma que soporta las herramientas (software) requeridas para producir los productos operativos que son resultado de la buena práctica de la ingeniería de software.[5] Puesto que la mayoría de las organizaciones de software tienen múltiples circunscripciones que requieren acceso al EIS, debe prescribirse la ventana temporal requerida para hardware y software y verificar que dichos recursos estarán disponibles.

Cuando un sistema basado en computadora (que incorpore hardware y software especializados) debe someterse a ingeniería, el equipo de software puede requerir acceso a elementos de hardware que se va a desarrollar por otros equipos de ingeniería. Por ejemplo, el software para un dispositivo robótico utilizado dentro de una célula de fabricación puede requerir un robot específico (por ejemplo, un soldador robótico) como parte del paso de prueba de validación; un proyecto de software para plantilla de página avanzada acaso necesite un sistema de impresión digital de alta velocidad en algún momento durante el desarrollo. Cada elemento de hardware debe especificarse como parte de la planificación.

26.5 Estimación de proyectos de software

Cita:

"En una época de outsourcing y creciente competencia, la capacidad para estimar con mayor precisión... ha surgido como un factor crítico de éxito para muchos grupos de TI."

Rob Thomsett

La estimación de costo y esfuerzo del software nunca será una ciencia exacta. Demasiadas variables (humanas, técnicas, ambientales, políticas) pueden afectar el costo final del software y el esfuerzo aplicado para su desarrollo. Sin embargo, la estimación del proyecto de software puede transformarse de un arte oscuro a una serie de pasos sistemáticos que proporcionen estimaciones con riesgo aceptable. Para lograr estimaciones confiables de costo y esfuerzo, surgen algunas opciones:

1. Retrase la estimación hasta avanzado el proyecto (obviamente, ¡puede lograr estimaciones 100 por ciento precisas después de que el proyecto esté completo!).
2. Base las estimaciones en proyectos similares que ya estén completos.
3. Use técnicas de descomposición relativamente simples para generar estimaciones de costo y esfuerzo de proyecto.
4. Use uno o más modelos empíricos para estimación de costo y esfuerzo de software.

5 Otro hardware, el *entorno meta*, es la computadora sobre la cual se ejecutará el software cuando éste se haya liberado al usuario final.

Desafortunadamente, la primera opción, aunque atractiva, no es práctica. Las estimaciones de costo deben proporcionarse por anticipado. No obstante, debe reconocer que mientras más espere, más conocerá, y mientras más conozca, menos probabilidades tendrá de cometer errores serios en sus estimaciones.

Cita:

"Es muy difícil hacer una defensa vigorosa, plausible y arriesgada de una estimación que se infiera sin método cuantitativo, apoyada por pocos datos y certificada principalmente por las corazonadas de los gerentes."

Fred Brooks

La segunda opción puede funcionar razonablemente bien si el proyecto actual es muy similar a esfuerzos anteriores y otros factores que influyen en el proyecto (por ejemplo, el cliente, condiciones del negocio, entorno de ingeniería de software, fechas límite) son aproximadamente equivalentes. Desafortunadamente, la experiencia pasada no siempre es buen indicador de resultados futuros.

Las opciones restantes son enfoques viables para la estimación del proyecto de software. De manera ideal, las técnicas anotadas para cada opción deben aplicarse en cascada y cada una es una comprobación cruzada para las demás. Las técnicas de descomposición tienen un enfoque de "divide y vencerás" para la estimación del proyecto. Al descomponer un proyecto en funciones principales y actividades de ingeniería de software relacionadas, la estimación de costo y esfuerzo puede realizarse en forma escalonada. Los modelos de estimación empírica pueden usarse para complementar las técnicas de descomposición y ofrecer un enfoque de estimación potencialmente valioso por derecho propio. Un modelo se basa en la experiencia (datos históricos) y toma la forma

$$d = f(v_i)$$

donde d es uno de los valores estimados (por ejemplo, esfuerzo, costo, duración del proyecto) y v_i son parámetros independientes seleccionados (por ejemplo, LOC o PF estimadas).

Las herramientas de estimación automatizadas implementan una o más técnicas de descomposición o modelos empíricos y proporcionan una atractiva opción para estimar. En tales sistemas se describen las características de la organización de desarrollo (por ejemplo, experiencia, entorno) y el software que se va a desarrollar. Las estimaciones de costo y esfuerzo se infieren de dichos datos.

Cada una de las opciones de estimación de costo del software viables sólo es tan buena como los datos históricos usados para generar la estimación. Si no existen datos históricos, el cálculo descansa sobre un cimiento muy inseguro. En el capítulo 25 se examinan las características de algunas de las métricas de software que proporcionan la base para los datos de estimación históricos.

26.6 TÉCNICAS DE DESCOMPOSICIÓN

La estimación del proyecto de software es una forma de resolución de problemas y, en la mayoría de los casos, el problema por resolver; es decir, desarrollar una estimación de costo y esfuerzo para un proyecto de software es muy complejo como para considerarse en una sola pieza. Por esta razón, debe descomponerse el problema y volver a caracterizarlo como un conjunto de problemas más pequeños (y, esperanzadoramente, más manejables).

En el capítulo 24, el enfoque de descomposición se analizó desde dos puntos de vista diferentes: descomposición del problema y descomposición del proceso. La estimación usa una o ambas formas de división. Pero antes de hacer una estimación, debe entenderse el ámbito del software que se va a construir y generar una estimación de su "tamaño".

PUNTO CLAVE

El "tamaño" del software que se va a construir puede estimarse usando una medida directa, LOC, o indirecta, PF.

26.6.1 Dimensionamiento del software

La precisión de una estimación de proyecto de software se basa en algunas cosas: 1) el grado en el que se estimó adecuadamente el tamaño del producto que se va a construir, 2) la habilidad para traducir la estimación de tamaño en esfuerzo humano, tiempo calendario y dinero (una

función de la disponibilidad de métricas de software confiables de proyectos anteriores), 3) el grado en el que el plan del proyecto refleja las habilidades del equipo de software y 4) la estabilidad de los requisitos del producto y el entorno que soporta el esfuerzo de ingeniería de software.

En esta sección se considera el problema del *dimensionamiento del software*. Puesto que una estimación de proyecto sólo es tan buena como la estimación del tamaño del trabajo que se va a realizar, el dimensionamiento representa el primer gran desafío como planificador. En el contexto de la planificación del software, el tamaño se refiere a un resultado cuantificable del proyecto de software. Si se toma un enfoque directo, el tamaño puede medirse en líneas de código (LOC). Si se elige un enfoque indirecto, el tamaño se representa como puntos de función (PF).

Putnam y Myers [Put92] sugieren cuatro enfoques diferentes para el problema de dimensionamiento:

¿Cómo se dimensiona el software que se planea construir?

- *Dimensionamiento de "lógica difusa"*. Este enfoque usa las técnicas de razonamiento aproximadas que son la piedra angular de la lógica difusa. Para aplicar este enfoque, el planificador debe identificar el tipo de aplicación, establecer su magnitud en una escala cualitativa y luego refinar la magnitud dentro del rango original.
- *Dimensionamiento del punto de función*. El planificador desarrolla estimaciones de las características del dominio de información que se estudiaron en el capítulo 23.
- *Dimensionamiento de componente estándar*. El software está compuesto de algunos "componentes estándares" diferentes que son genéricos a un área de aplicación particular. Por ejemplo, los componentes estándares para un sistema de información son subsistemas, módulos, pantallas, reportes, programas interactivos, programas en lote, archivos, LOC e instrucciones en el nivel objeto. El planificador del proyecto estima el número de ocurrencias de cada componente estándar y luego usa datos de proyecto históricos para estimar el tamaño entregado por componente estándar.
- *Dimensionamiento del cambio*. Este enfoque se usa cuando un proyecto abarca el uso de software existente que debe modificarse en alguna forma como parte de un proyecto. El planificador estima el número y tipo (por ejemplo, reuso, código agregado, cambio de código, código borrado) de las modificaciones que deben lograrse.

Putnam y Myers sugieren que los resultados de cada uno de estos enfoques de dimensionamiento se combinen estadísticamente para crear una estimación de *tres puntos* o *valor esperado*. Esto se logra al desarrollar valores para tamaño optimistas (bajos), más probables y pesimistas (altos), y al combinarlos usando la ecuación 26.1, descrita en la sección 26.6.2.

26.6.2 Estimación basada en problema

En el capítulo 25 se describieron las líneas de código y los puntos de función como medidas a partir de las cuales pueden calcularse métricas de productividad. Los datos LOC y PF se usan en dos formas durante la estimación del proyecto de software: 1) como variables de estimación para "dimensionar" cada elemento del software y 2) como métricas de referencia recopiladas de proyectos pasados y utilizadas en conjunto con variables de estimación para desarrollar proyecciones de costo y esfuerzo.

¿Que tienen en común las estimaciones basadas en LOC y en PF?

Las estimaciones LOC y PF son técnicas de estimación distintas, aunque ambas tienen algunas características en común. Comience con un enunciado acotado del ámbito del software y a partir de este enunciado intente descomponer el enunciado de ámbito en funciones problema que puedan estimarse cada una de manera individual. De modo alternativo, puede elegir otro componente para dimensionamiento, como clases u objetos, cambios o procesos empresariales afectados.

Las métricas de productividad de referencia (por ejemplo, LOC/pm o PF/pm[6]) se aplican entonces a la variable de estimación adecuada y se infiere el costo o esfuerzo para la función. Las estimaciones de función se combinan para producir una estimación global para todo el proyecto.

Cuando recopile métricas de productividad para proyectos, asegúrese de establecer una taxonomía de tipos de proyecto. Esto le permitirá calcular promedios específicos de dominio y hacer estimaciones más precisas.

Sin embargo, es importante observar que con frecuencia existe una sustancial dispersión en las métricas de productividad para una organización, lo que hace sospechoso el uso de una sola métrica de referencia para la productividad. En general, los promedios de LOC/pm o PF/pm deben calcularse por dominio de proyecto. Es decir, los proyectos deben agruparse por tamaño de equipo, área de aplicación, complejidad y otros parámetros relevantes. Luego se calculan los promedios de dominio local. Cuando estime un nuevo proyecto, primero debe asignarlo a un dominio y después debe usar un promedio de dominio adecuado para productividad anterior a la generación de la estimación.

Las técnicas de estimación LOC y PF difieren en el nivel de detalle requerido para descomposición y en la meta de la partición. Cuando se usa LOC como la variable de estimación, la descomposición es absolutamente esencial y con frecuencia lleva a considerables niveles de detalle. Mientras mayor sea el grado de partición, es más probable que puedan desarrollarse estimaciones de LOC razonablemente precisas.

Para estimaciones PF, la descomposición funciona de modo diferente. En lugar de enfocarse en la función, se estima cada una de las características del dominio de información (entradas, salidas, archivos de datos, consultas e interfaces externas), así como los 14 valores de ajuste de complejidad que se estudiaron en el capítulo 23. Entonces las estimaciones resultantes pueden usarse para inferir un valor PF que pueda ligarse a datos pasados y usarse para generar una estimación.

Sin importar la variable de estimación que se utilice, debe comenzar por estimar un rango de valores para cada función o valor de dominio de información. Con el uso de datos históricos o (cuando todo lo demás falle) de la intuición, estime un valor de tamaño optimista, más probable y pesimista para cada función o conteo para cada valor de dominio de información. Cuando se especifica un rango de valores, se proporciona un indicio implícito del grado de incertidumbre.

¿Cómo se calcula el "valor esperado" para el tamaño del software?

Entonces puede calcularse un valor de tres puntos o esperado. El *valor esperado* para la variable de estimación (tamaño) S puede calcularse como un promedio ponderado de las estimaciones optimista (S_{opt}), más probable (S_m) y pesimista (S_{pes}). Por ejemplo,

$$S = \frac{S_{opt} + 4S_m + S_{pes}}{6} \tag{26.1}$$

le da más crédito a la estimación "más probable" y sigue una distribución de probabilidad beta. Se supone que hay una probabilidad muy pequeña de que el resultado de tamaño real se ubicará afuera de los valores optimista o pesimista.

Una vez determinado el valor esperado para la variable de estimación se aplican datos de productividad históricos LOC o PF. ¿Las estimaciones son correctas? La única respuesta razonable a esta pregunta es "no puede estar seguro". Cualquier técnica de estimación, sin importar su sofisticación, debe verificarse con otro enfoque. Incluso así, deben prevalecer el sentido común y la experiencia.

26.6.3 Un ejemplo de estimación basada en LOC

Como ejemplo de técnicas de estimación LOC y PF basadas en problema, considere un paquete de software que se va a desarrollar para una aplicación de diseño asistido por computadora para componentes mecánicos. El software debe ejecutarse en una estación de trabajo de ingeniería y tener interfaz con varios periféricos de gráficos de computadora, incluido un ratón, digitaliza-

6 El acrónimo *pm* significa persona-mes de esfuerzo.

dora, pantalla a color de alta resolución e impresora láser. Es posible desarrollar un enunciado preliminar del ámbito del software:

> El software CAD mecánico aceptará datos geométricos bidimensionales y tridimensionales de un ingeniero. El ingeniero interactuará y controlará el sistema CAD a través de una interfaz de usuario que mostrará características de buen diseño de interfaz hombre/máquina. Todos los datos geométricos y otra información de apoyo se mantendrán en una base de datos CAD. Los módulos de análisis de diseño se desarrollarán para producir la salida requerida, que se desplegará en varios dispositivos gráficos. El software se diseñará para controlar e interactuar con dispositivos periféricos que incluyen un ratón, digitalizadora, impresora láser y plotter.

Muchas aplicaciones modernas residen en una red o son parte de una arquitectura cliente-servidor. Por tanto, asegúrese de que sus estimaciones incluyen el esfuerzo requerido para desarrollar "infraestructura" de software.

Este enunciado de ámbito es preliminar, no está acotado. Cada oración tendría que expandirse para proporcionar detalle concreto y acotamiento cuantitativo. Por ejemplo, antes de comenzar la estimación, el planificador debe determinar qué significa "características de buen diseño de interfaz hombre/máquina" o cuáles serán el tamaño y sofisticación de la "base de datos CAD".

Para los propósitos señalados, suponga que ocurrió mayor refinamiento y que se identifican las principales funciones del software mencionadas en la figura 26.2. Después de la técnica de descomposición para LOC se elabora una tabla de estimación (figura 26.2). Para cada función se desarrollan estimaciones para un rango de LOC. Por ejemplo, el rango de estimaciones LOC para la función de análisis geométrico 3D es optimista, 4 600 LOC; más probablemente, 6 900 LOC, y pesimista, 8 600 LOC. Al aplicar la ecuación 26.1, el valor esperado para la función de análisis geométrico 3D es 6 800 LOC. Otras estimaciones se infieren en forma similar. Al sumar verticalmente en la columna LOC estimada, para el sistema CAD se establece un estimado de 33 200 líneas de código.

No sucumba a la tentación de usar este resultado como su estimación de proyecto. Debe inferir otro resultado usando un enfoque diferente.

Una revisión de los datos históricos indica que la productividad organizacional promedio para los sistemas de este tipo es 620 LOC/pm. Con base en una tarifa de mano de obra sobrecargada de US$8 000 por mes, el costo por línea de código es aproximadamente US$13. Con base en la estimación LOC y los datos de productividad históricos, el costo de proyecto total estimado es US$431 000 y el esfuerzo estimado es 54 persona-meses.[7]

26.6.4 Un ejemplo de estimación basada en PF

La descomposición para estimación basada en PF se enfoca en valores de dominio de información en lugar de en funciones del software. Con base en la tabla que se presentó en la figura 26.3 se estimarían entradas, salidas, consultas, archivos e interfaces externas para el software CAD.

FIGURA 26.2
Tabla de estimación para los métodos LOC

Función	LOC estimadas
Interfaz de usuario y facilidades de control (IUFC)	2 300
Análisis geométrico bidimensional (AG2D)	5 300
Análisis geométrico tridimensional (AG3D)	6 800
Gestión de base de datos (GBD)	3 350
Facilidades de despliegue de gráficos de computadora (FDGC)	4 950
Función de control periférico (FCP)	2 100
Módulos de análisis de diseño (MAD)	8 400
Líneas de código estimadas	*33 200*

7 Las estimaciones se redondean a US$1 000 y persona-mes más cercanos. Mayor precisión es innecesaria e irreal, dadas las limitaciones de la precisión de la estimación.

CasaSegura

Estimación

La escena: Oficina de Doug Miller mientras comienza la planificación del proyecto.

Participantes: Doug Miller (gerente del equipo de ingeniería de software *CasaSegura*) y Vinod Raman, Jamie Lazar y otros miembros del equipo de ingeniería de software del producto.

La conversación:

Doug: Necesitamos desarrollar una estimación del esfuerzo para el proyecto y luego debemos definir un microcalendario para el primer incremento y un macrocalendario para los incrementos restantes.

Vinod (afirma con la cabeza): Muy bien, pero todavía no hemos definido algún incremento.

Doug: Cierto, pero es por eso por lo que necesitamos estimar.

Jamie (frunce el ceño): ¿Quieres saber cuánto tiempo nos tomará?

Doug: Esto es lo que necesito. Primero, necesitamos descomponer funcionalmente el software *CasaSegura*... en un nivel superior... luego tenemos que determinar el número de líneas de código que tomará cada función... luego...

Jamie: ¡Vaya! ¿Cómo se supone que haremos eso?

Vinod: Yo lo hice en proyectos anteriores. Comienzas con casos de uso, determinas la funcionalidad requerida para implementar cada uno, estimas el conteo de LOC para cada pieza de la función. El mejor enfoque es hacer que todos estimen de manera independiente y luego comparar los resultados.

Doug: O puedes hacer una descomposición funcional de todo el proyecto.

Jamie: Pero eso llevará mucho tiempo y ya teníamos que empezar.

Vinod: No... puede hacerse en pocas horas... esta mañana, de hecho.

Doug: Estoy de acuerdo... no podemos esperar exactitud, sólo una idea aproximada de cuál será el tamaño de *CasaSegura*.

Jamie: Creo que sólo debemos estimar el esfuerzo... es todo.

Doug: También haremos eso. Luego usen ambas estimaciones como comprobación cruzada.

Vinod: Vamos a hacerlo...

FIGURA 26.3
Estimación de información de valores de dominio

Valor de dominio de información	Opt.	Probable	Pes.	Conteo est.	Peso	Conteo PF
Número de entradas externas	20	24	30	24	4	97
Número de salidas externas	12	15	22	16	5	78
Número de consultas externas	16	22	28	22	5	88
Número de archivos lógicos internos	4	4	5	4	10	42
Número de archivos de interfaz externos	2	2	3	2	7	15
Conteo total						*320*

Un valor PF se calcula usando la técnica analizada en el capítulo 23. Para los propósitos de esta estimación se supone que el factor de ponderación de complejidad es el promedio. La figura 26.3 presenta los resultados de esta estimación.

Cada uno de los factores de ponderación de complejidad se estima y el factor de ajuste de valor se calcula como se describe en el capítulo 23:

Factor	Valor
Respaldo y recuperación	4
Comunicaciones de datos	2
Procesamiento distribuido	0
Rendimiento crítico	4
Existencia de entorno operativo	3
Entrada de datos en línea	4

Transacción de entrada sobre múltiples pantallas	5
Archivos maestros actualizados en línea	3
Complejo de valores de dominio de información	5
Complejo de procesamiento interno	5
Código diseñado para reuso	4
Conversión/instalación en diseño	3
Instalaciones múltiples	5
Aplicación diseñada para cambio	5
Factor de ajuste de valor	**1.17**

Finalmente, se infiere el número estimado de PF:

$$FP_{estimado} = \text{conteo total} \times [0.65 + 0.01 \times \Sigma(F_i)] = 375$$

La productividad organizacional promedio para sistemas de este tipo es 6.5 PF/pm. Con base en una tarifa de mano de obra sobrecargada de US$8 000 por mes, el costo por PF es aproximadamente US$1 230. Con base en el PF estimado y los datos de productividad históricos, el costo de proyecto estimado total es US$461 000 y el esfuerzo estimado es 58 personas-meses.

26.6.5 Estimación basada en proceso

La técnica más común para estimar un proyecto es basar la estimación sobre el proceso que se usará. Es decir, el proceso se descompone en un conjunto relativamente pequeño de tareas y se estima el esfuerzo requerido para lograr cada tarea.

Como en las técnicas basadas en problemas, la estimación basada en proceso comienza con un delineado de las funciones de software obtenidas del ámbito del proyecto. Para cada función debe realizarse una serie de actividades de marco conceptual. Las funciones y actividades de marco conceptual relacionadas[8] pueden representarse como parte de una tabla similar a la que se presentó en la figura 26.4.

Una vez fusionadas las funciones del problema y las actividades de proceso, se estima el esfuerzo (por ejemplo, persona-mes) que se requerirá para lograr cada actividad de proceso de

FIGURA 26.4
Tabla de estimación basada en proceso

Actividad →	CC	Planificación	Análisis de riesgo	Ingeniería		Construcción/liberación		CE	Totales
Tarea →				Análisis	Diseño	Código	Prueba		
Función									
↓									
UICF				0.50	2.50	0.40	5.00	n/a	8.40
2DGA				0.75	4.00	0.60	2.00	n/a	7.35
3DGA				0.50	4.00	1.00	3.00	n/a	8.50
CGDF				0.50	3.00	1.00	1.50	n/a	6.00
DBM				0.50	3.00	0.75	1.50	n/a	5.75
PCF				0.25	2.00	0.50	1.50	n/a	4.25
DAM				0.50	2.00	0.50	2.00	n/a	5.00
Totales	0.25	0.25	0.25	3.50	20.50	4.50	16.50		46.00
% esfuerzo	1%	1%	1%	8%	45%	10%	36%		

CC = comunicación cliente CE = evaluación cliente

8 Las actividades de marco conceptual elegidas para este proyecto difieren un poco de las actividades genéricas estudiadas en el capítulo 2. Son: comunicación con el cliente (CC), planificación, análisis de riesgos, ingeniería y construcción/liberación.

CONSEJO

Si el tiempo lo permite, use granularidad más fina cuando especifique las tareas de la figura 26.4. Por ejemplo, descomponga el análisis en sus principales tareas y estime cada una por separado.

software para cada función del software. Dichos datos constituyen la matriz central de la tabla de la figura 26.4. Las tarifas de mano de obra promedio (es decir, esfuerzo costo/unidad) se aplican entonces al esfuerzo estimado para cada actividad del proceso. Es muy probable que la tarifa de mano de obra varíe para cada tarea. El personal ejecutivo está enormemente involucrado en las primeras actividades de marco conceptual y por lo general son más costosos que el personal no ejecutivo involucrado en la construcción y liberación.

Los costos y esfuerzos para cada función y actividad de marco conceptual se calculan igual que el último paso. Si la estimación basada en proceso se realiza independientemente de la estimación LOC o PF, entonces se tienen dos o tres estimaciones para costo y esfuerzo que pueden compararse y reconciliarse. Si ambos conjuntos de estimaciones muestran concordancia razonable, hay buenas razones para creer que las estimaciones son confiables. Si, por otra parte, los resultados de dichas técnicas de descomposición muestran poca concordancia, debe realizarse mayor investigación y análisis.

26.6.6 Un ejemplo de estimación basada en proceso

Para ilustrar el uso de la estimación basada en proceso, considere el software CAD que se presentó en la sección 26.6.3. La configuración del sistema y todas las funciones de software permanecen sin cambios y se indican por ámbito de proyecto.

Cita:

"Es mejor comprender el fondo de una estimación antes de usarla."

Barry Boehm y Richard Fairley

En la tabla completa basada en proceso que se muestra en la figura 26.4, las estimaciones de esfuerzo (en persona-meses) para cada actividad de ingeniería de software se proporcionan para cada función del software CAD (abreviada por conveniencia). Las actividades de ingeniería y construcción/liberación se subdividen en las principales tareas de ingeniería de software que se muestran. Para comunicación con el cliente, planificación y análisis de riesgo se proporcionan estimaciones burdas de esfuerzo. Las mismas se anotan en la hilera total al fondo de la tabla. Los totales horizontal y vertical proporcionan un indicio del esfuerzo estimado requerido para análisis, diseño, código y prueba. Debe observarse que 53 por ciento de todo el esfuerzo se emplea en tareas de ingeniería frontales (análisis de requisitos y diseño), lo que indica la relativa importancia de este trabajo.

Con base en una tarifa promedio de mano de obra sobrecargada de US$8 000 por mes, el costo total estimado promedio es US$368 000, y el esfuerzo estimado es 46 persona-mes. Si se desea, las tarifas de mano de obra podrían asociarse con cada actividad de marco conceptual o tarea de ingeniería de software y calcularse por separado.

26.6.7 Estimación con casos de uso

Como se señaló a lo largo de la parte 2 de este libro, los casos de uso brindan a un equipo de software comprensión acerca del ámbito y los requisitos del software. Sin embargo, desarrollar un enfoque de estimación con casos de uso es problemático por las siguientes razones [Smi99]:

? ¿Por qué es difícil desarrollar una técnica de estimación usando casos de uso?

- Los casos de uso se describen usando muchos formatos y estilos diferentes; no existe una forma estándar.
- Los casos de uso representan una visión externa (la visión del usuario) del software y, por tanto, pueden escribirse en muchos niveles de abstracción diferentes.
- Los casos de uso no abordan la complejidad de las funciones y de las características que se describen.
- Los casos de uso pueden describir comportamiento complejo (por ejemplo, interacciones) que involucran muchas funciones y características.

A diferencia de una LOC o de un punto de función, el "caso de uso" de una persona puede requerir meses de esfuerzo, mientras que el de otra puede implementarse en un día o dos.

Aunque algunos investigadores consideran los casos de uso como una entrada de estimación, a la fecha ningún método de estimación probado ha surgido.[9] Smith [Smi99] sugiere que los casos de uso pueden usarse para estimación, pero sólo si se consideran dentro del contexto de la "jerarquía estructural" donde se usan para describir.

Smith argumenta que cualquier nivel de esta jerarquía estructural puede describirse mediante no más de 10 casos de uso. Cada uno de éstos abarcaría no más de 30 escenarios distintos. Obviamente, los casos de uso que describen un sistema grande se escriben en un nivel de abstracción mucho más alto (y representan considerablemente más esfuerzo de desarrollo) que los que describen un solo subsistema. En consecuencia, antes de poder usar los casos de uso para estimación, se establece el nivel dentro de la jerarquía estructural, se determina la longitud promedio (en páginas) de cada caso de uso, se define el tipo de software (por ejemplo, tiempo real, empresarial, ingeniería/científico, *webapp*, incrustado) y se considera una arquitectura burda para el sistema. Una vez establecidas dichas características pueden usarse datos empíricos para establecer el número estimado de LOC o PF por caso de uso (por cada nivel de la jerarquía). Entonces se usan datos históricos a fin de calcular el esfuerzo requerido para desarrollar el sistema.

Para ilustrar cómo puede realizarse este cálculo, considere la siguiente relación:[10]

$$\text{LOC estimadas} = N \times \text{LOC}_{\text{prom}} + [(S_a/S_h - 1) + (P_a/P_h - 1)] \times \text{LOC}_{\text{ajuste}} \tag{26.2}$$

donde

N = número real de casos de uso
LOC_{prom} = LOC promedio históricas por caso de uso para este tipo de subsistema
$\text{LOC}_{\text{ajuste}}$ = representa un ajuste con base en n por ciento de LOC_{prom}, donde n se define localmente y representa la diferencia entre este proyecto y los proyectos "promedio"
S_a = escenarios reales por caso de uso
S_h = escenarios promedio por caso de uso para este tipo de subsistema
P_a = páginas reales por caso de uso
P_h = páginas promedio por caso de uso para este tipo de subsistema

La expresión 26.2 podría usarse para desarrollar una estimación burda del número de LOC con base en el número real de casos de uso ajustados por el número de escenarios y la longitud de página de los casos de uso. El ajuste representa hasta n por ciento de las LOC promedio históricas por caso de uso.

26.6.8 Un ejemplo de estimación basada en caso de uso

El software CAD introducido en la sección 26.6.3 se compone de tres grupos de subsistemas: subsistema de interfaz de usuario (incluye UICF), grupo de subsistemas de ingeniería (incluye los subsistemas 2DGA, 3DGA y DAM) y grupo de subsistemas de infraestructura (incluye los subsistemas CGDF y PCF). Seis casos de uso describen el subsistema de interfaz de usuario. Cada uno se describe mediante no más de 10 escenarios y tiene una longitud promedio de seis páginas. El grupo de subsistemas de ingeniería se describe mediante 10 casos de uso (se considera que están en un nivel superior de la jerarquía estructural). Cada uno de estos casos de uso tiene no más de 20 escenarios asociados con él y una longitud promedio de ocho páginas. Finalmente,

9 Trabajo reciente en la inferencia de *puntos de casos de uso* [Cle06] a final de cuentas puede conducir a un enfoque de estimación utilizable, usando casos de uso.

10 Es importante observar que la expresión 26.2 se usa exclusivamente con propósitos ilustrativos. Como todos los modelos de estimación, debe validarse localmente antes de que pueda usarse con confianza.

FIGURA 26.5
Estimación de caso de uso

	casos de uso	escenarios	páginas	escenarios	páginas	LOC	LOC estimadas
Subsistema de interfaz de usuario	6	10	6	12	5	560	3 366
Grupo de subsistemas de ingeniería	10	20	8	16	8	3 100	31 233
Grupo de subsistemas de infraestructura	5	6	5	10	6	1 650	7 970
Total LOC estimadas							42 568

el grupo de subsistemas de infraestructura se describe mediante cinco casos de uso con un promedio de sólo seis escenarios y una longitud promedio de cinco páginas.

Usando la relación anotada en la expresión 26.2, con n = 30 por ciento, se elaboró la tabla que se muestra en la figura 26.5. Observe la primera hilera de la tabla; los datos históricos indican que el software UI requiere un promedio de 800 LOC por caso de uso cuando el caso de uso no tiene más de 12 escenarios y se describe en menos de cinco páginas. Dichos datos se ajustan razonablemente bien para el sistema CAD. Por tanto, la estimación LOC para el subsistema de interfaz de usuario se calcula con la expresión 26.2. Usando el mismo enfoque se hacen estimaciones para los grupos de subsistemas de ingeniería e infraestructura. La figura 26.5 resume las estimaciones e indica que el tamaño global del CAD se estima en 42 500 LOC.

Con 620 LOC/pm como la productividad promedio para sistemas de este tipo y una tarifa de mano de obra sobrecargada de $8 000 por mes, el costo por línea de código es aproximadamente US$13. Con base en la estimación de caso de uso y los datos de productividad históricos, el costo total estimado del proyecto es US$552 000 y el esfuerzo estimado es 68 persona-meses.

26.6.9 Reconciliación de estimaciones

Cita:

"Los métodos complicados pueden no producir una estimación más precisa, en particular cuando los desarrolladores pueden incorporar su propia intuición en la estimación."

Philip Johnson *et al.*

Las técnicas de estimación estudiadas en las secciones anteriores dan como resultado estimaciones múltiples que deben reconciliarse para producir una sola estimación de esfuerzo, duración de proyecto o costo. Para ilustrar este procedimiento de reconciliación, considere de nuevo el software CAD introducido en la sección 26.6.3.

El esfuerzo total estimado para el software CAD varía de uno bajo de 46 persona-meses (inferido con el enfoque de estimación basado en proceso) a uno alto de 68 persona-meses (inferido con estimación de caso de uso). La estimación promedio (usando los cuatro enfoques) es de 56 persona-meses. La variación de la estimación promedio es aproximadamente 18 por ciento en el lado bajo y 21 por ciento en el alto.

¿Qué ocurre cuando es pobre la concordancia entre las estimaciones? La respuesta a esta pregunta requiere una reevaluación de la información usada para hacer las estimaciones. Las estimaciones ampliamente divergentes con frecuencia pueden tener una de dos causas: 1) el ámbito del proyecto no se entiende adecuadamente o el planificador lo malinterpretó o 2) los datos de productividad usados por las técnicas de estimación basadas en problema son inadecuadas para la aplicación, obsoletos (ya no reflejan con precisión la organización de ingeniería de software) o se aplicaron mal. Debe determinar la causa de la divergencia y luego reconciliar las estimaciones.

INFORMACIÓN

Técnicas de estimación automatizada para proyectos de software

Las herramientas de estimación automatizadas permiten al planificador estimar costo y esfuerzo, y realizar análisis "y... si" para variables de proyecto importantes, como fecha de entrega o personal. Aunque existen muchas herramientas automatizadas (vea la barra lateral más adelante en este capítulo), todas muestran las mismas características generales y todas realizan las siguientes seis funciones genéricas [Jon96]:

1. *Dimensionamiento de entregas de proyecto.* Estimación del "tamaño" de uno o más productos operativos de software. Los

productos operativos incluyen la representación externa del software (por ejemplo, pantalla, reportes), el software en sí (por ejemplo, KLOC), funcionalidad entregada (por ejemplo, puntos de función) e información descriptiva (por ejemplo, documentos).
2. *Selección de actividades de proyecto*. Selección del marco conceptual de proceso adecuado y especificación del conjunto de tareas de ingeniería de software.
3. *Predicción de niveles de personal*. Especificación del número de personas que estarán disponibles para hacer el trabajo. Puesto que la relación entre personal disponible y trabajo (esfuerzo predicho) es enormemente no lineal, ésta es una entrada importante.
4. *Predicción de esfuerzo de software*. Las herramientas de estimación usan uno o más modelos (sección 26.7) que relacionan el tamaño de las entregas del proyecto con el esfuerzo requerido para producirlos.
5. *Predicción del costo de software*. Dados los resultados del paso 4, los costos pueden calcularse asignando tarifas de mano de obra a las actividades de proyecto anotadas en el paso 2.
6. *Predicción de calendarios de software*. Cuando se conocen el esfuerzo, el nivel de personal y las actividades del proyecto, puede producirse un calendario tentativo para asignar la mano de obra a través de las actividades de ingeniería de software con base en modelos recomendados para distribución de esfuerzo, lo que se estudia más adelante en este capítulo.

Cuando diferentes herramientas de estimación se aplican a los mismos datos de proyecto, puede encontrarse una variación relativamente grande en los resultados estimados. Más importante, en ocasiones, los valores predichos son significativamente diferentes a los valores reales. Esto refuerza la noción de que la salida de las herramientas de estimación debe usarse como un "punto de datos" de los cuales se derivan estimaciones, no como la única fuente para una estimación.

26.7 Modelos de estimación empíricos

Un modelo de estimación para software de computadora usa fórmulas empíricamente inferidas para predecir el esfuerzo como una función de LOC o PF.[11] Los valores para LOC o PF se estiman usando el enfoque descrito en las secciones 26.6.3 y 26.6.4. Pero en lugar de usar las tablas descritas en dichas secciones, los valores resultantes de LOC o PF se alimentan con el modelo de estimación.

PUNTO CLAVE

Un modelo de estimación refleja la población de proyectos de los cuales se derivó. Por tanto, el modelo es sensible al dominio.

Los datos empíricos que soportan a la mayoría de los modelos de estimación se infieren de una muestra limitada de proyectos. Por esta razón, ningún modelo de estimación es adecuado para todas las clases de software y en todos los entornos de desarrollo. Por tanto, se deben usar juiciosamente los resultados obtenidos de tales modelos.

Un modelo de estimación debe calibrarse para que refleje las condiciones locales. El modelo debe probarse aplicando los datos recopilados de los proyectos completados, alimentando los datos en el modelo y luego comparando los resultados reales con los predichos. Si la concordancia es pobre, el modelo debe afinarse y volverse a probar antes de poder usarse.

26.7.1 La estructura de los modelos de estimación

Un modelo de estimación típico se infiere usando análisis de regresión sobre los datos recopilados de proyectos de software anteriores. La estructura global de tales modelos toma la forma [Mat94]

$$E = A + B \times (e_v)^C \tag{26.3}$$

donde A, B y C son constantes derivadas empíricamente, E es esfuerzo en persona-meses y e_v es la variable de estimación (LOC o PF). Además de la relación anotada en la ecuación 26.3, la mayoría de los modelos de estimación tienen alguna forma de componente de ajuste de proyecto que permite que E se ajuste mediante otras características del proyecto (por ejemplo,

11 En la sección 26.6.6 se sugiere un modelo empírico que use casos de uso como la variable independiente. Sin embargo, a la fecha, en la literatura han aparecido relativamente pocos.

complejidad del problema, experiencia del personal, entorno de desarrollo). Entre los muchos modelos de estimación orientados a LOC propuestos en la literatura, están:

CONSEJO

Ninguno de estos modelos debe usarse sin calibración cuidadosa en su entorno.

$E = 5.2 \times (\text{KLOC})^{0.91}$	Modelo Walston-Felix
$E = 5.5 + 0.73 \times (\text{KLOC})^{1.16}$	Modelo Bailey-Basili
$E = 3.2 \times (\text{KLOC})^{1.05}$	Modelo Boehm simple
$E = 5.288 \times (\text{KLOC})^{1.047}$	Modelo Doty para KLOC > 9

También se han propuesto modelos orientados a PF. En ellos se incluyen:

$E = -91.4 + 0.355\ \text{PF}$	Modelo Albrecht y Gaffney
$E = -37 + 0.96\ \text{PF}$	Modelo Kemerer
$E = -12.88 + 0.405\ \text{PF}$	Pequeño modelo de regresión de proyecto

Un examen rápido a dichos modelos indica que cada uno producirá un resultado diferente para los mismos valores de LOC o PF. La implicación es clara. ¡Los modelos de estimación deben calibrarse para las necesidades locales!

26.7.2 El modelo COCOMO II

WebRef

En **sunset.usc.edu/research/COCOMOII/cocomo_main.html**, puede obtenerse información detallada acerca de COCOMO II, incluido software descargable.

En su libro clásico acerca de "economía de la ingeniería de software", Barry Boehm [Boe81] introdujo una jerarquía de modelos de estimación de software que llevan el nombre COCOMO, por *COnstructive COst MOdel*: modelo constructivo de costos. El modelo COCOMO original se convirtió en uno de los modelos de estimación de costo más ampliamente utilizados y estudiados en la industria. Evolucionó hacia un modelo de estimación más exhaustivo, llamado COCOMO II [Boe00]. Como su predecesor, COCOMO II en realidad es una jerarquía de modelos de estimación que aborda las áreas siguientes:

- *Modelo de composición de aplicación*. Se usa durante las primeras etapas de la ingeniería de software, cuando son primordiales la elaboración de prototipos de las interfaces de usuario, la consideración de la interacción del software y el sistema, la valoración del rendimiento y la evaluación de la madurez de la tecnología.
- *Modelo de etapa temprana de diseño*. Se usa una vez estabilizados los requisitos y establecida la arquitectura básica del software.
- *Modelo de etapa postarquitectónica*. Se usa durante la construcción del software.

Como todos los modelos de estimación para software, los modelos COCOMO II requieren información sobre dimensionamiento. Como parte de la jerarquía del modelo, están disponibles tres diferentes opciones de dimensionamiento: puntos objeto, puntos de función y líneas de código fuente.

? ¿Qué es un punto de objeto?

El modelo de composición de aplicación COCOMO II usa puntos de objeto y se ilustra en los siguientes párrafos. Debe observarse que otros modelos de estimación, más sofisticados (que usan PF y KLOC), también están disponibles como parte de COCOMO II.

Como los puntos de función, el *punto de objeto* es una medida de software indirecta que se calcula usando conteos del número de 1) pantallas (en la interfaz de usuario), 2) reportes y 3)

FIGURA 26.6

Ponderación de complejidad para tipos de objeto.
Fuente: [Boe96]

Tipo de objeto	Peso de complejidad		
	Simple	Medio	Difícil
Pantalla	1	2	3
Reporte	2	5	8
Componente 3GL			10

componentes que probablemente se requieran para construir la aplicación. Cada instancia de objeto (por ejemplo, una pantalla o reporte) se clasifica en uno de tres niveles de complejidad (simple, medio o difícil), usando criterios sugeridos por Boehm [Boe96]. En esencia, la complejidad es una función del número y de la fuente de las tablas de datos de cliente y servidor que se requieren para generar la pantalla o el reporte y el número de vistas o secciones que se presentan como parte de la pantalla o del reporte.

Una vez determinada la complejidad, el número de pantallas, reportes y componentes se ponderan de acuerdo con la tabla que se ilustra en la figura 26.6. Entonces se determina el conteo de puntos de objeto multiplicando el número original de instancias de objeto por el factor de ponderación que hay en la figura y se suman para obtener un conteo total de puntos de objeto. Cuando debe aplicarse desarrollo basado en componente o reuso de software general, se estima el porcentaje de reuso (%reuso) y el conteo de puntos de objeto se ajusta:

$$\text{NOP} = (\text{puntos de objeto}) \times [(100 - \%\text{reuso})/100]$$

donde NOP se define como nuevos puntos de objeto.

Para derivar una estimación del esfuerzo con base en el valor NOP calculado, debe derivarse una "tasa de productividad". La figura 26.7 presenta la tasa de productividad

$$\text{PROD} = \frac{\text{NOP}}{\text{persona-mes}}$$

para diferentes niveles de experiencia del desarrollador y de madurez del entorno de desarrollo.

Una vez determinada la tasa de productividad se calcula una estimación del esfuerzo del proyecto usando

$$\text{Esfuerzo estimado} = \frac{\text{NOP}}{\text{PROD}}$$

En modelos COCOMO II más avanzados,[12] se requieren varios factores de escala, controladores de costo y procedimientos de ajuste. Una discusión completa de éstos está más allá del ámbito de este libro. Si tiene más interés, vea [Boe00] o visite el sitio web de COCOMO II.

26.7.3 La ecuación del software

La *ecuación del software* [Put92] es un modelo dinámico multivariable que supone una distribución de esfuerzo específica durante la vida de un proyecto de desarrollo de software. El modelo

FIGURA 26.7 **Tasa de productividad para puntos de objeto.**
Fuente: [Boe96].

Experiencia/capacidad del desarrollador	Muy baja	Baja	Nominal	Alta	Muy alta
Madurez/capacidad del entorno	Muy baja	Baja	Nominal	Alta	Muy alta
PROD	4	7	13	25	50

12 Como se señaló anteriormente, estos modelos usan conteos PF y KLOC para la variable tamaño.

se infirió a partir de datos de productividad recopilados por más de 4 000 proyectos de software contemporáneos. Con base en dichos datos, se infiere un modelo de estimación de la forma

$$E = \frac{\text{LOC} \times B^{0.333}}{P^3} \times \frac{1}{t^4} \tag{26.4}$$

donde

WebRef

En **www.qsm.com** puede encontrarse información acerca de herramientas de estimación de costo de software que evolucionaron a partir de la ecuación de software.

E = esfuerzo en persona-meses o persona-años
t = duración del proyecto en meses o años
B = "factor de habilidades especiales"[13]
P = "parámetro de productividad" que refleja: madurez global del proceso y prácticas administrativas, la medida en la que se usan buenas prácticas de ingeniería de software, el nivel de lenguajes de programación utilizado, el estado del entorno de software, las habilidades y experiencia del equipo de software y la complejidad de la aplicación.

Valores típicos pueden ser $P = 2\,000$ para desarrollo de un software incrustado en tiempo real, $P = 10\,000$ para software de telecomunicaciones y sistemas y $P = 28\,000$ para aplicaciones de sistemas empresariales. El parámetro de productividad puede inferirse para condiciones locales, usando datos históricos recopilados de esfuerzos de desarrollo anteriores.

Debe observarse que la ecuación de software tiene dos parámetros independientes: 1) una estimación del tamaño y 2) una indicación de la duración del proyecto en meses o años calendario.

Para simplificar el proceso de estimación y usar una forma más común para sus modelos de estimación, Putnam y Myers [Put92] sugieren un conjunto de ecuaciones derivadas de la ecuación de software. El tiempo mínimo de desarrollo se define como

$$t_{\text{mín}} = 8.14 \frac{\text{LOC}}{p^{0.43}} \text{ en meses para } t_{\text{mín}} > 6 \text{ meses} \tag{26.5a}$$

$$E = 180\, Bt^3 \text{ en persona-meses para } E \geq 20 \text{ persona-meses} \tag{26.5b}$$

Observe que t en la ecuación 26.5b se representa en años.

Con la ecuación 26.5, para $P = 12\,000$ (el valor recomendado para software científico) para el software CAD que se estudió anteriormente en este capítulo,

$$t_{\text{mín}} = 8.14 \times \frac{33\,200}{12\,000^{0.43}} = 12.6 \text{ meses calendario}$$

$$E = 180 \times 0.28 \times (1.05)^3 = 58 \text{ persona-meses}$$

Los resultados de la ecuación de software corresponden favorablemente con las estimaciones desarrolladas en la sección 26.6. Como el modelo COCOMO anotado en la sección 26.7.2, la ecuación de software sigue evolucionando. Un análisis más a fondo de una versión extendida de este enfoque de estimación puede encontrarse en [Put97b].

26.8 Estimación para proyectos orientados a objetos

Vale la pena complementar los métodos de estimación de costo de software convencional con una técnica que se diseñó explícitamente para software OO. Lorenz y Kidd [Lor94] sugieren el siguiente enfoque:

13 B aumenta lentamente conforme "crecen la necesidad para integración, pruebas, aseguramiento de la calidad, documentación y habilidades administrativas" [Put92]. Para programas pequeños (KLOC = 5 a 15), $B = 0.16$. Para programas mayores de 70 KLOC, $B = 0.39$.

1. Desarrollar estimaciones usando descomposición de esfuerzo, análisis PF y cualquier otro método que sea aplicable para aplicaciones convencionales.
2. Usar el modelo de requisitos (capítulo 6), desarrollar casos de uso y determinar un conteo. Reconocer que el número de casos de uso puede cambiar conforme avance el proyecto.
3. A partir del modelo de requisitos, determinar el número de clases clave (llamadas clases de análisis en el capítulo 6).
4. Categorizar el tipo de interfaz para la aplicación y desarrollar un multiplicador para clases de apoyo:

Tipo de interfaz	Multiplicador
No GUI	2.0
Interfaz de usuario basada en texto	2.25
GUI	2.5
GUI compleja	3.0

   Multiplique el número de clases clave (paso 3) por el multiplicador a fin de obtener una estimación para el número de clases de apoyo.
5. Multiplicar el número total de clases (clave + apoyo) por el número promedio de unidades de trabajo por clase. Lorenz y Kidd sugieren 15 a 20 persona-días por clase.
6. Comprobación cruzada de la estimación basada en clase, multiplicando el número promedio de unidades de trabajo por caso de uso.

26.9 Técnicas de estimación especializadas

Las técnicas de estimación estudiadas en las secciones 26.6 a 26.8 pueden usarse para cualquier proyecto de software. Sin embargo, cuando un equipo de software encuentra una duración de proyecto extremadamente corta (semanas en lugar de meses) en la que es probable tener un torrente continuo de cambios, la planificación del proyecto en general y la estimación en particular deben abreviarse.[14] En las siguientes secciones se examinan dos técnicas de estimación especializadas.

26.9.1 Estimación para desarrollo ágil

Puesto que los requisitos para un proyecto ágil (capítulo 3) se definen mediante un conjunto de escenarios de usuario (por ejemplo, "historias" en programación extrema), es posible desarrollar un enfoque de estimación que sea informal, razonablemente disciplinado y significativo dentro del contexto de la planificación del proyecto para cada incremento de software. La estimación para proyectos ágiles usa un enfoque de descomposición que abarca los siguientes pasos:

? ¿Cómo se desarrollan las estimaciones cuando se aplica un proceso ágil?

1. Cada escenario de usuario (el equivalente de un minicaso de uso creado al comienzo mismo de un proyecto por los usuarios finales u otros participantes) se considera por separado con propósitos de estimación.
2. El escenario se descompone en el conjunto de tareas de ingeniería de software que será necesario desarrollar.

14 "Abreviar" *no* significa eliminar. Incluso los proyectos de corta duración deben planificarse, y la estimación es el cimiento de la planificación sólida.

3a. El esfuerzo requerido por cada tarea se estima por separado. Nota: La estimación puede basarse en datos históricos, un modelo empírico o la "experiencia".

3b. De manera alternativa, el "volumen" del escenario puede estimarse en LOC, PF o alguna otra medida orientada a volumen (por ejemplo, conteo de casos de uso).

4a. Las estimaciones para cada tarea se suman a fin de crear una estimación para el escenario.

4b. De manera alternativa, la estimación de volumen para el escenario se traduce en esfuerzo, usando datos históricos.

5. Las estimaciones de esfuerzo para todos los escenarios que se implementan para un incremento de software determinado se suman a fin de desarrollar la estimación del esfuerzo para el incremento.

PUNTO CLAVE

En el contexto de la estimación para proyectos ágiles, "volumen" es una estimación del tamaño global de un escenario de usuario en LOC o PF.

Puesto que la duración del proyecto requerido para el desarrollo de un incremento de software es muy corta (por lo general de tres a seis semanas), este enfoque de estimación tiene dos propósitos: 1) asegurarse de que el número de escenarios que se van a incluir en el incremento se ajusta a los recursos disponibles y 2) establecer una base para asignar esfuerzo conforme se desarrolla el incremento.

26.9.2 Estimación para webapp

Los *webapps* adoptan con frecuencia el modelo de proceso ágil. Puede usarse una medida de punto de función modificada, junto con los pasos que se destacan en la sección 26.9.1, a fin de desarrollar una estimación para la *webapp*. Roetzheim [Roe00] sugiere el siguiente enfoque cuando adapta puntos de función para estimación de *webapps*:

- *Entradas* son cada pantalla o formulario de entrada (por ejemplo, CGI o Java), cada pantalla de mantenimiento y, si usa una metáfora de etiquetas de libreta, cualquier etiqueta.
- *Salidas* son cada página web estática, cada guión de página web dinámica (por ejemplo, ASP, ISAPI u otro guión DHTML) y cada reporte (ya sea basado en web o administrativo por naturaleza).
- *Tablas* son cada tabla lógica en la base de datos más, si usa XML para almacenar datos en un archivo, cada objeto XML (o colección de atributos XML).
- Las *interfaces* conservan su definición como archivos lógicos (por ejemplo, formatos de registro único) dentro de las fronteras del sistema.
- *Consultas* son cada una de las publicaciones externas o el uso de una interfaz orientada a mensaje. Un ejemplo usual son las referencias externas DCOM o COM.

Los puntos de función (interpretados en la forma señalada) son un indicador razonable de volumen para una *webapp*.

Mendes *et al.* [Men01] sugieren que el volumen de una *webapp* se determina mejor al recolectar medidas (llamadas "variables predictoras") asociadas con la aplicación (por ejemplo, conteo de página, conteo de medios, conteo de función), características de su página web (por ejemplo, complejidad de página, complejidad de vinculación, complejidad gráfica), características de medios (por ejemplo, duración de los medios) y características funcionales (por ejemplo, longitud de código, longitud de código reutilizado). Dichas medidas pueden usarse para desarrollar modelos de estimación empíricos para esfuerzo de proyecto total, esfuerzo de autoría de página, esfuerzo de autoría de medios y esfuerzo de guiones. Sin embargo, todavía falta trabajo por hacer antes de que tales modelos puedan usarse con confianza.

HERRAMIENTAS DE SOFTWARE

Estimaciones de esfuerzo y costo

Objetivo: El objetivo de las herramientas de estimación de esfuerzo y costo es brindar a un equipo de proyecto estimaciones del esfuerzo requerido, duración del proyecto y costo, de manera que aborde las características específicas del proyecto a mano y el entorno donde se construirá el proyecto.

Mecánica: En general, las herramientas de estimación de costo usan una base de datos histórica inferida de proyectos locales y datos recopilados a través de la industria, y un modelo empírico (por ejemplo, COCOMO II) que se utiliza para derivar estimaciones de esfuerzo, duración y costo. Las características del proyecto y el entorno de desarrollo son entradas y la herramienta proporciona un rango de salidas de estimación.

Herramientas representativas:[15]

Costar, desarrollada por Softstar Systems (**www.softstarsystems.com**), usa el modelo COCOMO II para desarrollar estimaciones de software.

CostXpert, desarrollada por Cost Xpert Group, Inc. (**www.costxpert.com**) integra múltiples modelos de estimación y una base de datos histórica de proyectos.

Estimate Professional, desarrollada por el Software Productivity Centre, Inc. (**www.spc.com**), se basa en el COCOMO II y en el modelo SLIM.

Knowledge Plan, desarrollado por Software Productivity Research (**www.spr.com**), usa entrada de puntos de función como el principal controlador para un paquete de estimación completo.

Price S, desarrollada por Price Systems (**www.pricesystems.com**), es una de las herramientas de estimación más antiguas y de más uso para proyectos de desarrollo de software a gran escala.

SEER/SEM, desarrollada por Galorath, Inc. (**www.galorath.com**), proporciona una amplia capacidad de estimación, análisis de sensibilidad, valoración de riesgo y otras características.

SLIM-Estimate, desarrollada por QSM (**www.qsm.com**), se apoya en una exhaustiva "base de conocimiento industrial" para proporcionar una "comprobación de sanidad" para estimaciones que se infieren usando datos locales.

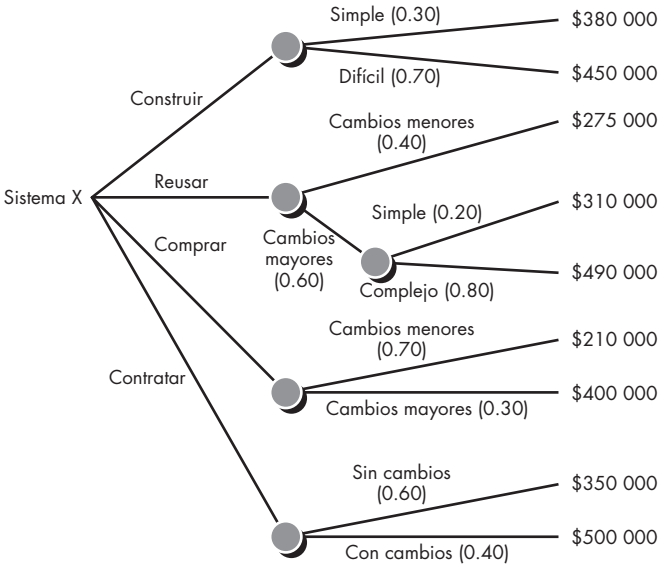
26.10 LA DECISIÓN HACER/COMPRAR

En muchas áreas de aplicación, con frecuencia es más efectivo en costo adquirir, en lugar de desarrollar, software de computadora. Los gerentes de ingeniería de software se enfrentan a la decisión hacer/comprar que puede complicarse todavía más por algunas opciones de adquisición: 1) el software puede comprarse (o licenciarse) de manera comercial, 2) los componentes de software de "experiencia completa" o "experiencia parcial" (vea la sección 26.4.2) pueden adquirirse y luego modificarse e integrarse para satisfacer necesidades específicas o 3) el software puede construirse a la medida por parte de un contratista externo para satisfacer las especificaciones del comprador.

Los pasos involucrados en la adquisición del software se definen por lo crucial del software que se va a comprar y por el costo final. En algunos casos (por ejemplo, software de PC de bajo costo), es menos costoso comprar y experimentar que realizar una larga evaluación de potenciales paquetes de software. En el análisis final, la decisión hacer/comprar se toma con base en las siguientes condiciones: 1) ¿La fecha de entrega del producto de software será más próxima que la del software que se desarrolle internamente? 2) ¿El costo de adquisición más el costo de personalización será menor que el costo que implica desarrollar el software internamente? 3) ¿El costo del apoyo exterior (por ejemplo, un contrato de mantenimiento) será menor que el costo del apoyo interno? Estas condiciones se aplican para cada una de las opciones de adquisición.

15 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

FIGURA 26.8
Árbol de decisiones para apoyar la decisión hacer/comprar



26.10.1 Creación de un árbol de decisión

¿Existe alguna manera sistemática de recorrer las opciones asociadas con la decisión hacer/comprar?

Los pasos recién descritos pueden aumentar usando técnicas estadísticas como el análisis de árbol de decisión.[16] Por ejemplo, la figura 26.8 muestra un árbol de decisión para un sistema X basado en software. En este caso, la organización de ingeniería de software puede: 1) construir el sistema X desde cero, 2) reusar componentes de experiencia parcial existentes para construir el sistema, 3) comprar un producto de software disponible y modificarlo para satisfacer las necesidades locales o 4) contratar el desarrollo del software a un proveedor externo.

Si el sistema se construirá desde cero, hay una probabilidad de 70 por ciento de que el trabajo será difícil. Al usar las técnicas de estimación estudiadas en este capítulo, el planificador de proyecto estima que un esfuerzo de desarrollo difícil costará US$450 000. Un esfuerzo de desarrollo "simple" se estima que cuesta US$380 000. El valor esperado por costo, calculado a lo largo de cualquier rama del árbol de decisión, es

$$\text{Costo esperado} = \Sigma \text{ (probabilidad de ruta)}_i \times \text{(costo de ruta estimado)}_i$$

donde *i* es la ruta del árbol de decisión. Para la ruta de construcción,

$$\text{Costo esperado}_{\text{construir}} = 0.30 \text{ (\$380K)} + 0.70 \text{ (\$450K)} = \$429\text{K}$$

Siguiendo otras rutas del árbol de decisión, también se muestran los costos proyectos para reúso, compra y contrato, bajo varias circunstancias. Los costos esperados para dichas rutas son

$$\text{Costo esperado}_{\text{reuso}} = 0.40 \text{ (\$275K)} + 0.60 \text{ [0.20 (\$310K)} + 0.80 \text{ (\$490K)]} = \$382\text{K}$$
$$\text{Costo esperado}_{\text{comprar}} = 0.70 \text{ (\$210K)} + 0.30 \text{ (\$400K)} = \$267\text{K}$$
$$\text{Costo esperado}_{\text{contratar}} = 0.60 \text{ (\$350K)} + 0.40 \text{ (\$500K)} = \$410\text{K}$$

Con base en la probabilidad y los costos proyectados que se anotaron en la figura 26.8, el costo esperado más bajo es la opción "comprar".

Sin embargo, es importante observar que deben considerarse muchos criterios, no sólo el costo, durante el proceso de toma de decisión. Disponibilidad, experiencia del desarrollador/

16 Una valiosa introducción al análisis del árbol de decisiones puede encontrarse en **http://en.wikipedia.org/wiki/Decision_tree**

proveedor/contratista, conformidad con los requisitos, "políticas" locales y la probabilidad de cambiar son sólo algunos de los criterios que pueden afectar la decisión final de construir, reusar, comprar o contratar.

26.10.2 *Outsourcing*

Tarde o temprano, toda compañía que desarrolla software de computadora plantea una pregunta fundamental: ¿hay alguna forma en la que puedan conseguirse el software y los sistemas necesarios a un precio más bajo? La respuesta a esta pregunta no es simple y las discusiones emocionales que ocurren en respuesta a la pregunta siempre conducen a una sola palabra: *outsourcing*.

Como concepto, el *outsourcing* (la subcontratación) es extremadamente simple. Las actividades de ingeniería de software se contratan a una tercera parte, que hace el trabajo a un costo más bajo y, con un poco de suerte, con mayor calidad. El trabajo de software realizado dentro de una compañía se reduce a una actividad de administración de contrato.[17]

Cita:

"Como regla, el outsourcing requiere una administración incluso más habilidosa que el desarrollo en casa."

Steve McConnell

La decisión por el *outsourcing* puede ser estratégica o táctica. En el nivel estratégico, los gerentes empresariales consideran si una porción significativa de todo el trabajo de software puede contratarse a otros. En el nivel táctico, un gerente de proyecto determina si parte o todo un proyecto puede lograrse mejor al subcontratar el trabajo de software.

Sin importar la amplitud del enfoque, la decisión de *outsourcing* con frecuencia es financiera. Un estudio detallado del análisis financiero para el *outsourcing* está más allá del ámbito de este libro y mejor se deja a otros (por ejemplo, [Min95]). Sin embargo, vale la pena una breve revisión de los pros y los contras de la decisión.

En el lado positivo, el ahorro en costo usualmente puede lograrse reduciendo el número de personal de software y las instalaciones (por ejemplo, computadoras, infraestructura) que lo apoyan. En el lado negativo, una compañía pierde cierto control sobre el software que necesita. Dado que el software es una tecnología que diferencia sus sistemas, servicios y productos, una compañía corre el riesgo de poner la fe de su competitividad en las manos de una tercera persona.

La tendencia hacia el *outsourcing* indudablemente continuará. La única forma de detenerla es reconocer que el trabajo de software es extremadamente competitivo en todos los niveles. La única manera de sobrevivir es volverse tan competitivo como los mismos proveedores de *outsourcing*.

CASASEGURA

Outsourcing

La escena: Sala de juntas en CPI Corporation, cuando el proyecto comienza.

Participantes: Mal Golden, gerente ejecutivo, desarrollo de producto; Lee Warren, gerente de ingeniería; Joe Camalleri, VP ejecutivo, desarrollo empresarial; y Doug Miller, gerente de proyecto, ingeniería de software.

La conversación:

Joe: Estamos considerando la subcontratación de la porción de ingeniería de software del producto *CasaSegura*.

Doug (consternado): ¿Cuándo pasó esto?

Lee: Conseguimos una cotización de un desarrollador externo. Viene con 30 por ciento por abajo de lo que tu grupo pare-

17 El *outsourcing* puede verse de manera más general como cualquier actividad que conduce a la adquisición del software o de los componentes de software desde una fuente externa a la organización de ingeniería de software.

ce cree que costará. Vélo. [Extiende la cotización a Doug, quien la lee.]

Mal: Como sabes, Doug, intentamos mantener bajos los costos y 30 por ciento es 30 por ciento. Además, estas personas vienen muy recomendadas.

Doug (toma un respiro e intenta permanecer tranquilo): Me tomaron por sorpresa, pero antes de que tomen una decisión final, ¿puedo comentar algo?

Joe (afirma con la cabeza): Claro, adelante.

Doug: No hemos trabajado con esta compañía de *outsourcing*, ¿cierto?

Mal: Sí, pero...

Doug: Y señalan que cualquier cambio en las especificaciones se facturará con una tarifa adicional, ¿cierto?

Joe (frunce el ceño): Cierto, pero esperamos que las cosas serán razonablemente estables.

Doug: Mala suposición, Joe.

Joe: Bueno...

Doug: Es probable que liberemos nuevas versiones de este producto dentro de algunos años. Y es razonable suponer que el software proporcionará muchas de las nuevas características, ¿cierto?

[Todos afirman con la cabeza.]

Doug: ¿Alguna vez hemos coordinado un proyecto internacional?

Lee (observa preocupado): No, pero me dijeron...

Doug (intenta contener su enojo): Así que lo que me están diciendo es: 1) estamos a punto de trabajar con un proveedor desconocido, 2) los costos por hacer esto no son tan bajos como parecen, 3) nos comprometemos *de facto* para trabajar con ellos durante muchas liberaciones de producto, sin importar lo que hagan la primera vez y 4) vamos a aprender sobre la marcha lo relativo a un proyecto internacional.

[Todos permanecen en silencio.]

Doug: Chicos... Creo que éste es un error, y me gustaría que lo reconsideraran durante un día. Tenemos mucho más control si hacemos el trabajo en casa. Tenemos la experiencia y puedo garantizarles que no quiero que nos cueste mucho más... el riesgo será más bajo y sé que ustedes tienen aversión al riesgo, como yo.

Joe (frunce el ceño): Hiciste buenas observaciones, pero tú tienes un interés personal para mantener este proyecto en casa.

Doug: Es verdad, pero eso no cambia los hechos.

Joe (suspira): Está bien, consideremos esto durante un día o dos; lo pensaremos un poco más y nos reuniremos de nuevo para una decisión final. Doug, ¿puedo hablar contigo en privado?

Doug: Seguro... Realmente quiero estar seguro de que hacemos lo correcto.

26.11 Resumen

Un planificador de proyecto de software debe estimar tres cosas antes de comenzar un proyecto: cuánto tardará, cuánto esfuerzo se requerirá y cuántas personas se involucrarán. Además, el planificador debe predecir los recursos (hardware y software) que se requerirán y el riesgo involucrado.

El enunciado del ámbito ayuda al planificador a desarrollar estimaciones usando una o más técnicas que se clasifican en dos amplias categorías: descomposición y modelado empírico. Las técnicas de descomposición requieren una delineación de las principales funciones del software, seguidas por estimaciones de: 1) el número de LOC, 2) valores seleccionados dentro del dominio de información, 3) el número de casos de uso, 4) el número de persona-meses requeridos para implementar cada función o 5) el número de persona-meses requeridos para cada actividad de ingeniería de software. Las técnicas empíricas usan expresiones derivadas empíricamente para esfuerzo y tiempo a fin de predecir dichas cantidades de proyecto. Las herramientas automatizadas pueden usarse para implementar un modelo empírico específico.

Las estimaciones de proyecto precisas por lo general usan al menos dos de las tres técnicas recién anotadas. Al comparar y reconciliar las estimaciones desarrolladas usando diferentes técnicas, el planificador tiene más probabilidad de derivar una estimación precisa. La estimación de proyecto de software nunca puede ser una ciencia exacta, pero una combinación de buenos datos históricos y técnicas sistemáticas pueden mejorar la precisión de la estimación.

Problemas y puntos por evaluar

26.1. Suponga que usted es el gerente de proyecto de una compañía que construye software para robots caseros. Se le contrata a fin de construir el software para un robot que pode el césped para el propietario de

una casa. Escriba un enunciado para el ámbito que describa el software. Asegúrese de que su enunciado de ámbito esté acotado. Si no está familiarizado con los robots, haga un poco de investigación antes de comenzar a escribirlo. Además, establezca sus suposiciones acerca del hardware que se requerirá. Alternativa: Sustituya el robot podadora con otro problema que sea de su interés.

26.2. La complejidad del proyecto de software se analiza brevemente en la sección 26.1. Elabore una lista de características de software (por ejemplo, operación concurrente, salida gráfica) que afecten la complejidad de un proyecto. Priorice la lista.

26.3. El rendimiento es una importante consideración durante la planificación. Analice cómo puede interpretarse de manera diferente el rendimiento, dependiendo del área de aplicación del software.

26.4. Haga una descomposición funcional del software de robot que describió en el problema 26.1. Estime el tamaño de cada función en LOC. Si supone que su organización produce 450 LOC/pm con una tarifa de mano de obra sobrecargada de US$7 000 por persona-mes, estime el esfuerzo y el costo requeridos para construir el software, usando la técnica de estimación basada en LOC descrita en este capítulo.

26.5. Use el modelo COCOMO II para estimar el esfuerzo requerido para construir software para un simple ATM que produce 12 pantallas, 10 reportes y que requerirá aproximadamente 80 componentes de software. Suponga complejidad promedio y madurez desarrollador/entorno promedio. Use el modelo de composición de aplicación con puntos de objeto.

26.6. Use la ecuación de software para estimar el software del robot podadora. Suponga que la ecuación 26.4 es aplicable y que $P = 8\,000$.

26.7. Compare las estimaciones de esfuerzo inferidas en los problemas 26.4 y 26.6. ¿Cuál es la desviación estándar y cómo afecta esto a su grado de certidumbre acerca de la estimación?

26.8. Usando los resultados obtenidos en el problema 26.7, determine si es razonable esperar que el software pueda construirse dentro de los siguientes seis meses y cuántas personas tendrían que emplearse para realizar el trabajo.

26.9. Desarrolle un modelo de hoja de cálculo que implemente una o más de las técnicas de estimación descritas en este capítulo. De manera alternativa, adquiera uno o más modelos en línea para estimación de fuentes en la web.

26.10. Para un equipo de proyecto: desarrolle una herramienta de software que implemente cada una de las técnicas de estimación desarrolladas en este capítulo.

26.11. Parece extraño que las estimaciones de costo y calendario se desarrollen durante la planificación del proyecto de software, antes del análisis detallado de los requisitos de software o de realizar el diseño. ¿Por qué cree que se haga esto? ¿Existen circunstancias para no hacerlo?

26.12. Vuelva a calcular los valores esperados que se anotaron para el árbol de decisión en la figura 26.8 y suponga que cada rama tiene una probabilidad 50-50. ¿Esto cambiaría su decisión final?

Lecturas y fuentes de información adicionales

La mayoría de los libros de administración de proyectos de software contienen análisis acerca de la estimación del proyecto. The Project Management Institute (*PMBOK Guide,* PMI, 2001), Wysoki *et al*. (*Effective Project Management,* Wiley, 2000), Lewis (*Project Planning Scheduling and Control,* 3a. ed., McGraw-Hill, 2000), Bennatan (*On Time, Within Budget: Software Project Management Practices and Techniques,* 3a. ed., Wiley, 2000) y Phillips [Phi98] proporcionan útiles lineamientos de estimación.

McConnell (*Software Estimation: Demystifying the Black Art*, Microsoft Press, 2006) escribió una guía pragmática que proporciona valiosos lineamientos para todos aquellos que deban estimar el costo del software. Parthasarathy (*Practical Software Estimation*, Addison-Wesley, 2007) enfatiza los puntos de función como una métrica de estimación. Laird y Brennan (*Software Measurement and Estimation: A Practical Approach*, Wiley-IEEE Computer Society Press, 2006) aborda las mediciones y su uso en la estimación del software. Pfleeger (*Software Cost Estimation and Sizing Methods, Issues, and Guidelines*, RAND Corporation, 2005) desarrolló un manual abreviado que aborda muchos fundamentos de estimación. Jones (*Estimating Software Costs*, 2a. ed., McGraw-Hill, 2007) escribió uno de los tratamientos más exhaustivos de los modelos y los datos que son aplicables a la estimación del software en todo dominio de aplicación. Coombs (*IT Project Estimation*, Cambridge University Press, 2002) y Roetzheim y Beasley (*Software Project Cost and Schedule Estimating: Best*

Practices, Prentice-Hall, 1997) presentan muchos modelos útiles y sugieren lineamientos, paso a paso, para generar las mejores estimaciones posibles.

En internet, está disponible una gran variedad de fuentes de información acerca de la estimación del software. Una lista actualizada de referencias en la World Wide Web que son relevantes para la estimación del software puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

CAPÍTULO

27 CALENDARIZACIÓN DEL PROYECTO

CONCEPTOS CLAVE

A finales de los años sesenta del siglo pasado, se eligió a un entusiasta joven ingeniero para que "escribiera" un programa de computadora para una aplicación de fabricación automatizada. La razón para su selección fue simple. Él era la única persona en su grupo técnico que asistió a un seminario de programación de computadoras. Sabía los pros y los contras del lenguaje ensamblador y de FORTRAN, pero nada conocía acerca de ingeniería de software incluso menos acerca de la calendarización y el seguimiento de proyectos.

Su jefe le dio los manuales apropiados y una descripción verbal de lo que tenía que hacer. Se le informó que el proyecto debía estar terminado en dos meses.

Leyó los manuales, consideró su enfoque y comenzó a escribir el código. Después de dos semanas, el jefe lo llamó a su oficina y le preguntó sobre cómo iban las cosas.

"Realmente grandiosas", dijo el ingeniero con entusiasmo juvenil. "Esto fue mucho más simple de lo que pensé. Probablemente tenga ya un avance de 75 por ciento".

El jefe sonrió y alentó al joven ingeniero a seguir con el buen trabajo. Planearon reunirse de nuevo en una semana.

Una semana después, el jefe llamó al ingeniero a su oficina y le preguntó: "¿Dónde estamos?"

"Todo está bien", dijo el joven, "pero encontré algunos tropiezos. Los allanaré y pronto estaré de vuelta en el camino".

"¿Qué te parece la fecha límite?", preguntó el jefe.

"No hay problema", dijo el ingeniero. "tengo un avance de cerca de 90 por ciento".

UNA MIRADA RÁPIDA

¿Qué es? Ya seleccionó un modelo de proceso adecuado, identificó las tareas de ingeniería de software que deben realizarse, estimó la cantidad de trabajo y el número de personas, conoce la fecha límite e incluso consideró los riesgos. Ahora es momento de unir los puntos. Es decir, tiene que crear una red de tareas de ingeniería de software que le permitirán concluir el trabajo a tiempo. Una vez creada la red, tiene que asignar responsables para cada tarea, asegurarse de que se realizan todas ellas y adaptar la red conforme los riesgos que habrá cuando se convierta en realidad. En pocas palabras, eso es la calendarización y el seguimiento del proyecto de software.

¿Quién lo hace? En el nivel del proyecto, los gerentes de proyecto de software que usan la información solicitada a los ingenieros de software. En lo individual, los mismos ingenieros de software.

¿Por qué es importante? Para construir un sistema complejo, muchas tareas de ingeniería de software ocurren en paralelo, y el resultado del trabajo realizado durante una tarea puede tener un profundo efecto sobre el trabajo que se va a realizar en otra tarea. Estas interdependencias son muy difíciles de comprender sin un calendario. También es virtualmente imposible valorar el avance en un proyecto de software regular o grande sin un calendario detallado.

¿Cuáles son los pasos? Las tareas de ingeniería de software dictadas por el modelo de proceso del software se refinan en función de la funcionalidad que se va a construir. Se asigna esfuerzo y duración determinados a cada tarea y se crea una red de tareas (también llamada "red de actividad") de manera que permita al equipo de software alcanzar la fecha de entrega establecida.

¿Cuál es el producto final? Se produce el calendario del proyecto y la información relacionada.

¿Cómo me aseguro de que lo hice bien? La calendarización adecuada requiere que: 1) todas las tareas aparezcan en la red, 2) el esfuerzo y la calendarización se asignen de manera inteligente a cada tarea, 3) las interdependencias entre tareas se indiquen de manera adecuada, 4) se asignen los recursos para el trabajo que se va a realizar y 5) se proporcionen hitos cercanamente espaciados de modo que pueda darse seguimiento al progreso.

Si el lector ha trabajado en el mundo del software durante algunos años, puede terminar la historia. No es de sorprender que el joven ingeniero[1] se quedara en 90 por ciento de avance durante todo el proyecto y terminara (con la ayuda de otros) sólo un mes más tarde.

Esta historia se ha repetido decenas de miles de veces entre los desarrolladores de software durante las pasadas cinco décadas. La gran pregunta es por qué.

27.1 Conceptos básicos

Aunque existen muchas razones por las que el software se entrega tardíamente, la mayoría pueden rastrearse en una o más de las siguientes causas fundamentales:

- Una fecha límite irreal establecida por alguien externo al equipo de software y que fuerza a los gerentes y profesionales.
- Requerimientos del cliente variables que no se reflejan en cambios del calendario.
- Una honesta subestimación de la cantidad de esfuerzo y/o número de recursos que se requerirán para hacer el trabajo.
- Riesgos predecibles y/o impredecibles que no se consideraron cuando comenzó el proyecto.
- Dificultades humanas que no podían preverse por anticipado.
- Falta de comunicación entre el personal del proyecto que da como resultado demoras.
- Falta de comunicación entre el equipo de trabajo que se traduce en retrasos.
- Una falla por parte de la administración del proyecto para reconocer que el proyecto tiene retrasos en el calendario y una falta de acción para corregir el problema.

Cita:

"Los calendarios excesivos o irracionales son probablemente la influencia más destructiva en todo el software."

Capers Jones

Las fechas límite agresivas (léase "irreales") son un hecho de la vida en el negocio del software. En ocasiones, tales fechas límite se demandan por razones legítimas, desde el punto de vista de la persona que las establece. Pero el sentido común dice que la legitimidad también debe percibirla el personal que hace el trabajo.

Napoleón dijo una vez: "Cualquier comandante que se comprometa a llevar a cabo un plan que considere defectuoso está equivocado; debe plantear sus razones, insistir en que se cambie el plan y finalmente ofrecer formalmente su renuncia en lugar de ser el instrumento de la derrota de su ejército". Éstas son duras palabras que muchos gerentes de proyecto de software deberían ponderar.

Las actividades de estimación estudiadas en el capítulo 26 y las técnicas de calendarización descritas en este capítulo con frecuencia se implementan bajo la restricción de una fecha límite definida. Si las mejores estimaciones indican que la fecha límite es irreal, un gerente de proyecto competente debe "proteger a su equipo contra la presión excesiva [calendario]... [y] devolver la presión a quienes la originaron" [Pag85].

Cita:

"Me encantan las fechas límite. Me gusta el zumbido que producen cuando pasan volando."

Douglas Adams

Para ilustrar lo anterior, suponga que a su equipo de software se le pide construir un controlador en tiempo real para un instrumento de diagnóstico médico que debe introducirse en el mercado en nueve meses. Después de realizar la estimación y el análisis de riesgo cuidadosamente (capítulo 28), llega a la conclusión de que el software, como se solicitó, requerirá 14 meses para su creación con el personal que se tiene disponible. ¿Cómo procedería?

Es irreal marchar hacia la oficina del cliente (en este caso el probable cliente es mercadotecnia/ventas) y demandar que se cambie la fecha de entrega. Las presiones del mercado externo dictaron la fecha y el producto debe liberarse. Es igualmente temerario rechazar el compromiso de realizar el trabajo (desde el punto de vista profesional). De modo que, ¿qué hacer? Ante esta situación, los autores recomiendan los siguientes pasos:

1 En caso de que el lector se lo pregunte, esta historia es autobiográfica.

1. Realice una estimación detallada, usando datos históricos de proyectos anteriores. Determine el esfuerzo y la duración estimados para el proyecto.
2. Con el modelo de proceso incremental (capítulo 2), desarrolle una estrategia de ingeniería de software que entregue funcionalidad crucial hacia la fecha límite impuesta, pero desarrolle otra estrategia para otra entrega de software con funcionalidad hasta más tarde. Documente el plan.

? ¿Qué debe hacer cuando los gerentes demandan una fecha límite que es imposible de cumplir?

3. Reúnase con el cliente y (con la estimación detallada) explique por qué la fecha límite es irreal. Asegúrese de señalar que todas las estimaciones se basan en el rendimiento de proyectos anteriores. También asegúrese de indicar el porcentaje de mejora que se requeriría para lograr cumplir en la fecha límite, como se plantea originalmente.[2] El siguiente comentario puede ser apropiado como respuesta:

   Creo que podemos tener problemas con la fecha de entrega para el software controlador XYZ. A cada uno de ustedes le entrego una distribución abreviada de las tasas de desarrollo para proyectos de software anteriores y una estimación que se hizo de maneras distintas. Observarán que supuse una mejora de 20 por ciento en tasas de desarrollo anteriores, pero todavía tenemos una fecha de entrega que es de 14 meses en lugar de nueve.

4. Ofrezca la estrategia de desarrollo incremental como una alternativa:

   Tenemos algunas opciones, y me gustaría que tomaran una decisión con base en ellas. Primero, podemos aumentar el presupuesto y conseguir recursos adicionales, de modo que podremos tener listo este trabajo en nueve meses. Pero entiendo que esto aumentará el riesgo de empobrecer la calidad debido a las fechas tan apretadas.[3] Segundo, podemos remover algunas funciones y capacidades del software que se solicitan. Esto hará que la versión preliminar del producto sea un poco menos funcional, pero puede anunciarse toda la funcionalidad y entregarla durante el periodo de 14 meses. Tercero, podemos prescindir de la realidad y querer que el proyecto esté completo en nueve meses. Acabaremos sin tener algo que pueda entregarse al cliente. La tercera opción, y creo que estarán de acuerdo, es inaceptable. La historia pasada y nuestras mejores estimaciones dicen que es irreal y que representa una receta para el desastre.

Habrá algunos gruñidos, pero si se presenta una estimación sólida con base en buenos datos históricos, es probable que se elijan las versiones negociadas de las opciones 1 o 2. La fecha límite irreal se evapora.

27.2 Calendarización del proyecto

Alguna vez se le preguntó a Fred Brooks cómo es que los proyectos de software se atrasan en su calendario. Su respuesta fue tan simple como profunda: "un día a la vez".

La realidad de un proyecto técnico (ya sea que implique construir una hidroeléctrica o desarrollar un sistema operativo) es que cientos de pequeñas tareas deben ocurrir para lograr una meta más grande. Algunas de esas tareas yacen fuera de la corriente principal y pueden completarse sin preocuparse acerca de su impacto sobre la fecha de conclusión del proyecto. Otras se encuentran en la "ruta crítica". Si las tareas "críticas" se retrasan en el calendario, la fecha de conclusión de todo el proyecto se pone en riesgo.

CONSEJO

Las tareas requeridas para lograr el objetivo de un gerente de proyecto no deben realizarse manualmente. Existen muchas excelentes herramientas de calendarización. Úselas.

Como gerente de proyecto, su objetivo es definir todas las tareas del proyecto, construir una red que muestre sus interdependencias, identificar las que son cruciales dentro de la red y luego monitorear su progreso para asegurar que la demora se reconoce "en el momento". Para lograr

2 Si la mejora requerida es de 10 a 25 por ciento, en realidad puede tener listo el trabajo. Pero, muy probablemente, la mejora requerida en el rendimiento del equipo será mayor a 50 por ciento. Ésta es una expectativa irreal.

3 También puede argumentar que aumentar el número de personas no reduce el tiempo de manera proporcional.

esto, debe tener un calendario que se haya definido en un grado de resolución que permita monitorear el progreso y controlar el proyecto.

La *calendarización del proyecto de software* es una acción que distribuye el esfuerzo estimado a través de la duración planificada del proyecto, asignando el esfuerzo a tareas específicas de ingeniería del software. Sin embargo, es importante observar que el calendario evoluciona con el tiempo. Durante las primeras etapas de la planificación del proyecto se desarrolla un calendario macroscópico. Este tipo de calendario identifica las principales actividades de marco conceptual de proceso y las funciones de producto a las cuales se aplican. Conforme el proyecto avanza, cada entrada en el calendario macroscópico se desglosa en un calendario detallado. Aquí, acciones y tareas de software específicas (requeridas para lograr una actividad) se identifican y calendarizan.

Cita:

"La calendarización demasiado optimista no da como resultado calendarios reales más cortos; da como resultado unos más largos."

Steve McConnell

La calendarización para proyectos de ingeniería de software puede verse desde dos perspectivas más bien diferentes. En la primera, ya se estableció una fecha final (e irrevocable) para liberar un sistema basado en computadora. La organización de software se restringe para distribuir el esfuerzo dentro del marco temporal prescrito. La segunda visión de la calendarización del software supone que se han discutido límites cronológicos burdos, pero que la fecha final la establece la organización de ingeniería del software. El esfuerzo se distribuye para hacer mejor uso de los recursos y se define una fecha final después de un cuidadoso análisis del software. Por desgracia, la primera situación se encuentra con mucha más frecuencia que la segunda.

27.2.1 Principios básicos

Como otras áreas de la ingeniería de software, algunos principios básicos guían la calendarización del proyecto de software:

PUNTO CLAVE

Cuando desarrolle un calendario, divida el trabajo, anote las interdependencias entre tareas, asigne esfuerzo y tiempo a cada tarea, y defina responsabilidades, resultados e hitos.

Compartimentalización. El proyecto debe compartimentalizarse en algunas actividades y tareas manejables. Para lograr la compartimentalización se desglosan tanto el producto como el proceso.

Interdependencia. Debe determinarse la interdependencia de cada actividad o tarea compartimentalizada. Algunas tareas deben sucederse en secuencia, mientras que otras pueden ocurrir en paralelo. Algunas actividades no pueden comenzar hasta que esté disponible el producto operativo producido por otra. Otras pueden realizarse de manera independiente.

Asignación de tiempo. A cada tarea por calendarizar debe asignársele cierto número de unidades de trabajo (por ejemplo, persona-días de esfuerzo). Además, a cada tarea debe asignársele una fecha de comienzo y una de conclusión, en función de las interdependencias y de si el trabajo se realizará sobre una base de tiempo completo o parcial.

Validación de esfuerzo. Todo proyecto tiene un número definido de personas en el equipo de software. Conforme ocurre la asignación de tiempo, debe asegurarse de que, en un momento determinado, no se ha calendarizado más que el número de personal asignado. Por ejemplo, considere un proyecto que tiene tres ingenieros de software asignados (tres personas-días están disponibles por día de esfuerzo asignado[4]). En un día determinado deben lograrse siete tareas concurrentes. Cada tarea requiere 0.50 persona-días de esfuerzo. Hay más esfuerzo por asignar que personas disponibles para hacer el trabajo.

Responsabilidades definidas. Cada tarea por calendarizar debe asignarse a un miembro de equipo específico.

4 En realidad, están disponibles menos de tres personas-días de esfuerzo debido a juntas no relacionadas, enfermedad, vacaciones y varias otras razones. Sin embargo, para nuestros propósitos, se supone 100 por ciento de disponibilidad.

Resultados definidos. Cada tarea que se calendarice debe tener un resultado definido. Para proyectos de software, el resultado usualmente es un producto operativo (por ejemplo, el diseño de un componente) o una parte de un producto operativo. Los productos operativos con frecuencia se combinan con productos operativos entregables.

Hitos definidos. Cada tarea o grupo de tareas debe asociarse con un hito del proyecto. Un hito se logra cuando uno o más productos operativos se revisan en su calidad (capítulo 15) y se aprueban.

Cada uno de estos principios se aplica conforme evoluciona el calendario del proyecto.

27.2.2 Relación entre personal y esfuerzo

En un pequeño proyecto de desarrollo de software, una sola persona puede analizar requerimientos, elaborar diseño, generar código y realizar pruebas. Conforme el tamaño de un proyecto aumenta, más personas deben involucrarse. (¡Rara vez uno puede darse el lujo de abordar un esfuerzo de 10 persona-años con una persona que trabaje durante 10 años!).

CONSEJO

Si debe agregar personal a un proyecto retrasado, asegúrese de que se les asigna trabajo que esté muy compartimentalizado.

Existe un mito común que todavía creen muchos gerentes responsables de proyectos de desarrollo de software: "Si nos atrasamos en el calendario, siempre podemos agregar más programadores y ponernos al corriente en el proyecto más adelante". Por desgracia, agregar personal tardíamente en un proyecto con frecuencia tiene efectos perturbadores sobre el proyecto, lo que hace que el calendario se deteriore todavía más. Las personas que se agregan deben aprender el sistema y las que les enseñan son las mismas personas que hacían el trabajo. Mientras enseñan no trabajan y el proyecto se atrasa aún más.

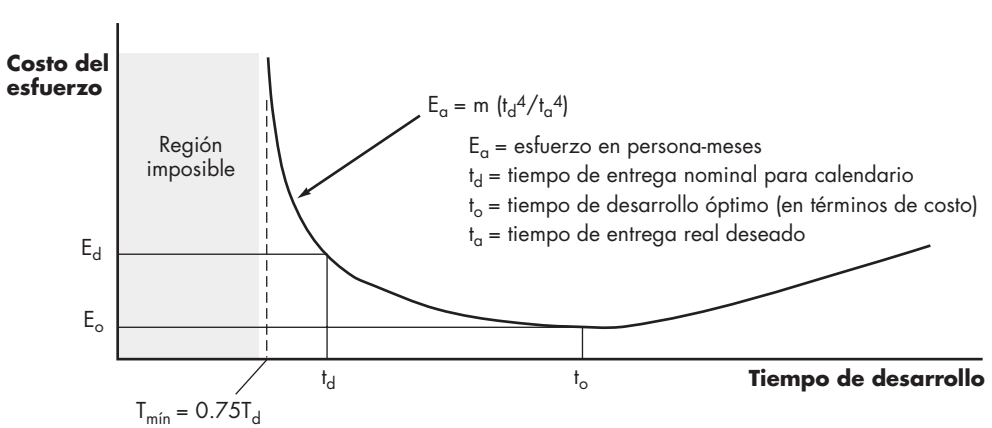
Además del tiempo que tardan en aprender el sistema, más personas aumentan el número de rutas de comunicación y la complejidad de la comunicación a lo largo de un proyecto. Aunque la comunicación es absolutamente esencial para el desarrollo de software exitoso, toda nueva ruta de comunicación requiere esfuerzo adicional y, por tanto, tiempo adicional.

Con los años, los datos empíricos y el análisis teórico han demostrado que los calendarios de proyecto son elásticos. Es decir: es posible comprimir en cierta medida la fecha de conclusión de un proyecto deseado (al agregar recursos adicionales). También lo es extender una fecha de conclusión (al reducir el número de recursos).

La *curva Putnam-Norden-Rayleigh (PNR)*[5] proporciona un indicio de la relación entre esfuerzo aplicado y tiempo de entrega para un proyecto de software. En la figura 27.1 se muestra una

FIGURA 27.1

Relación entre esfuerzo y tiempo de entrega



5 La investigación original puede encontrarse en [Nor70] y [Put78].

PUNTO CLAVE

Si la entrega puede retrasarse, la curva PNR indica que los costos del proyecto pueden reducirse sustancialmente.

versión de la curva, que representa el esfuerzo del proyecto como función del tiempo de entrega. La curva indica un valor mínimo t_o que señala el costo mínimo para entrega (es decir, el tiempo de entrega que resultará en el menor esfuerzo empleado). Conforme avanza hacia la izquierda de t_o (es decir, conforme se intenta acelerar la entrega), la curva se eleva de manera no lineal.

Como ejemplo, suponga que un equipo de proyecto estimó que se requerirá un nivel de esfuerzo E_d para lograr un tiempo de entrega nominal t_d que es óptimo en términos de calendario y recursos disponibles. Aunque es posible acelerar la entrega, la curva se eleva de manera muy pronunciada hacia la izquierda de t_d. De hecho, la curva PNR indica que el tiempo de entrega del proyecto no puede comprimirse mucho más allá de $0.75t_d$. Si se intenta mayor compresión, el proyecto se mueve hacia la "región imposible" y el riesgo de fracaso se vuelve muy alto. La curva PNR también indica la opción de entrega de costo más bajo, $t_o = 2t_d$. Lo que se observa aquí es que la entrega demorada del proyecto puede reducir los costos significativamente. Desde luego, esto debe ponderarse contra el costo empresarial asociado con la demora.

La ecuación de software [Put92] que se introdujo en el capítulo 26 se infirió a partir de la curva PNR y demuestra la relación enormemente no lineal entre el tiempo cronológico para completar un proyecto y el esfuerzo humano que se aplica al mismo. El número de líneas de código entregadas (enunciados fuente), L, se relaciona con el esfuerzo y el tiempo de desarrollo mediante la ecuación:

$$L = P \times E^{1/3} t^{4/3}$$

donde E es el esfuerzo de desarrollo en persona-meses, P es un parámetro de productividad que refleja varios factores que conducen a trabajo de ingeniería de software de alta calidad (por lo general valores de P que varían entre 2 000 y 12 000) y t es la duración del proyecto en meses calendario.

CONSEJO

Conforme la fecha límite del proyecto se acerca, se llega a un punto donde el trabajo no puede completarse en calendario, sin importar el número de personas que hagan el trabajo. Enfrente la realidad y defina una nueva fecha de entrega.

Al reordenar esta ecuación de software puede llegarse a una expresión para el esfuerzo de desarrollo E:

$$E = \frac{L^3}{P^3 t^4} \tag{27.1}$$

donde E es el esfuerzo empleado (en persona-años) durante todo el ciclo de vida para el desarrollo y mantenimiento del software y t es el tiempo de desarrollo en años. La ecuación para esfuerzo de desarrollo puede relacionarse con el costo de desarrollo mediante la inclusión de un factor de tarifa de mano de obra sobrecargada ($/persona-año).

Esto conduce a algunos resultados interesantes. Considere un complejo proyecto de software en tiempo real estimado en 33 000 LOC, 12 persona-años de esfuerzo. Si ocho personas se asignan al equipo del proyecto, éste puede completarse en aproximadamente 1.3 años. Sin embargo, la fecha final se extiende a 1.75 años; la naturaleza enormemente no lineal del modelo descrito en la ecuación 27.1 produce:

$$E = \frac{L^3}{P^3 t^4} \sim 3.8 \text{ persona-años}$$

Esto implica que, al extender la fecha final por seis meses, ¡es posible reducir el número de personas de ocho a cuatro! La validez de tales resultados está abierta a debate, pero la implicación es clara: pueden obtenerse beneficios usando menos personal durante un lapso un poco más largo para lograr el mismo objetivo.

27.2.3 Distribución de esfuerzo

Cada una de las técnicas de estimación del proyecto de software que se estudian en el capítulo 26 conducen a estimaciones de unidades de trabajo (por ejemplo, persona-meses) requeridas para completar el desarrollo de software. Una distribución de esfuerzo recomendada a través

del proceso de software con frecuencia se conoce como la *regla 40-20-40*. Cuarenta por ciento de todo el esfuerzo se asigna a análisis frontal y diseño. Un porcentaje similar se aplica a pruebas traseras. De ahí se infiere correctamente que la codificación pierde el énfasis (20 por ciento de esfuerzo).

? ¿Cómo debe distribuirse el esfuerzo a través del flujo de trabajo del proceso de software?

Esta distribución de esfuerzo debe usarse solamente como guía.[6] Las características de cada proyecto dictan la distribución del esfuerzo. El trabajo empleado en la planificación del proyecto rara vez representa más de 2 a 3 por ciento de esfuerzo, a menos que el plan comprometa a una organización a realizar más gastos con alto riesgo. La comunicación con el cliente y el análisis de requerimientos pueden comprender de 10 a 25 por ciento del esfuerzo del proyecto. El esfuerzo que se emplea en análisis o creación de prototipos debe aumentar en proporción directa con el tamaño y la complejidad del proyecto. Por lo general, al diseño de software se aplica un rango de 20 a 25 por ciento del esfuerzo. También debe considerarse el tiempo que se emplea para revisión del diseño y su posterior iteración.

Debido al esfuerzo aplicado al diseño de software, el código debe seguir relativamente con poca dificultad. Es posible lograr un esfuerzo global de 15 a 20 por ciento. Las pruebas y la posterior depuración pueden representar de 30 a 40 por ciento del esfuerzo de desarrollo del software. Lo crucial del software con frecuencia dicta la cantidad de pruebas que se requieren. Si el software se clasifica humanamente (es decir, si errores en el software pueden resultar en pérdida de la vida), incluso son usuales porcentajes más altos.

27.3 Definición de un conjunto de tareas para el proyecto de software

Sin importar el modelo de proceso que se elija, el trabajo que realiza un equipo de software se logra a través de un conjunto de tareas que permiten definir, desarrollar y, a final de cuentas, apoyar el software de computadora. Ningún conjunto de tareas es adecuado para todos los proyectos. Un conjunto de tareas adecuado para un sistema grande y complejo probablemente se percibirá como excesivo para un producto de software pequeño y relativamente simple. En consecuencia, un proceso de software efectivo debe definir una colección de conjuntos de tareas, cada una diseñada para satisfacer las necesidades de diferentes tipos de proyectos.

Como se anotó en el capítulo 2, un conjunto de tareas es una colección de tareas de trabajo de ingeniería del software, hitos, productos operativos y filtros de aseguramiento de la calidad que deben lograrse para completar un proyecto particular. El conjunto de tareas debe proporcionar suficiente disciplina a fin de lograr software de alta calidad. Pero, al mismo tiempo, no debe abrumar al equipo del proyecto con trabajo innecesario.

Con la finalidad de desarrollar un calendario del proyecto, en la línea de tiempo del proyecto debe distribuirse un conjunto de tareas. El conjunto de tareas variará dependiendo del tipo de proyecto y el grado de rigor con el que el equipo de software decide hacer su trabajo. Aunque es difícil desarrollar una taxonomía exhaustiva de los tipos de proyecto de software, la mayoría de las organizaciones de software encuentran los siguientes proyectos:

WebRef

Para auxiliar en la definición de los conjuntos de tareas para varios proyectos de software, se desarrolló un modelo de proceso adaptable (APM, por sus siglas en inglés). En **www.rspa.com/apm** puede encontrar una descripción completa del MPA en inglés.

1. *Proyectos de desarrollo de concepto* que inician para explorar algún concepto empresarial nuevo o la aplicación de alguna nueva tecnología.
2. Los *proyectos de desarrollo de nueva aplicación* que se realizan como consecuencia de la solicitud de un cliente específico.

6 En la actualidad, la regla 40-20-40 está bajo ataque. Algunos creen que más de 40 por ciento de esfuerzo global debe emplearse durante análisis y diseño. Por otro lado, quienes proponen el desarrollo ágil (capítulo 3) argumentan que debe emplearse menos tiempo "frontal" y que un equipo debe avanzar rápidamente hacia la construcción.

3. *Proyectos de mejora de aplicación* que ocurren cuando un software existente experimenta grandes modificaciones a funciones, rendimiento o interfaces que son observables por el usuario final.
4. *Proyectos de mantenimiento de aplicación* que corrigen, adaptan o extienden el software existente de maneras que pueden no ser inmediatamente obvias para el usuario final.
5. *Proyectos de reingeniería* que se llevan a cabo con la intención de reconstruir un sistema existente (heredado) en todo o en parte.

Incluso dentro de un solo tipo de proyecto, muchos factores influyen en el conjunto de tareas por elegir. Los factores incluyen [Pre05]: tamaño del proyecto, número de usuarios potenciales, vitalidad de la misión, longevidad de la aplicación, estabilidad de requerimientos, facilidad de comunicación cliente/desarrollador, madurez de tecnología aplicable, restricciones de rendimiento, características incrustadas y no incrustadas, personal del proyecto y factores de reingeniería. Cuando se combinan, dichos factores proporcionan un indicio del *grado de rigor* con el que debe aplicarse el proceso de software.

27.3.1 Un ejemplo de conjunto de tareas

Los proyectos de desarrollo de concepto inician cuando debe explorarse el potencial para alguna nueva tecnología. No hay certeza de que la tecnología será aplicable, pero un cliente (por ejemplo, mercadotecnia) cree que existen beneficios potenciales. Los proyectos de desarrollo de concepto se abordan al aplicar las siguientes acciones:

1.1 El **ámbito del concepto** determina el ámbito global del proyecto.

1.2 La **planificación preliminar del concepto** establece la habilidad de la organización para llevar a cabo el trabajo implicado por el ámbito del proyecto.

1.3 La **valoración del riesgo tecnológico** evalúa el riesgo asociado con la tecnología que se va a implementar como parte del ámbito del proyecto.

1.4 La **prueba del concepto** demuestra la viabilidad de una nueva tecnología en el contexto del software.

1.5 La **implementación del concepto** constituye la representación del concepto de manera que pueda revisarse por parte de un cliente y se usa con propósitos de "mercadotecnia" cuando debe venderse un concepto a otros clientes o gerentes.

1.6 La **reacción del cliente** al concepto solicita retroalimentación acerca de un nuevo concepto tecnológico y se dirige a aplicaciones de cliente específicas.

Una rápida exploración a dichas acciones debe producir pocas sorpresas. De hecho, el flujo de ingeniería de software para proyectos de desarrollo de concepto (y también para todos los otros tipos de proyectos) no es mucho más que sentido común.

27.3.2 Refinamiento de acciones de ingeniería del software

Las acciones de ingeniería de software descritas en la sección anterior pueden usarse para definir un calendario macroscópico para un proyecto. No obstante, éste debe refinarse para crear un calendario de proyecto detallado. El refinamiento comienza tomando cada acción y descomponiéndola en un conjunto de tareas (con productos operativos e hitos relacionados).

Como ejemplo de descomposición de tarea, considere la acción 1.1, ámbito del concepto. El refinamiento de las tareas puede lograrse usando un formato de bosquejo, pero en este libro se usará un enfoque de lenguaje de diseño de proceso para ilustrar el flujo de la acción de determinación del ámbito del concepto:

Task definition: Acción 1.1 Ámbito del concepto
1.1.1 Identificar necesidad, beneficios y clientes potenciales;
1.1.2 Definir salida/control deseado y eventos de entrada que impulsen la aplicación;
Begin Task 1.1.2
1.1.2.1 RT: Revisar descripción escrita de necesidad[7]
1.1.2.2 Inferir una lista de salidas/entradas visibles para el cliente
1.1.2.3 RT: revisar salidas/entradas con cliente y revisar según se requiera; endtask Task 1.1.2
1.1.3 Definir la funcionalidad/comportamiento para cada función principal;
Begin Task 1.1.3
1.1.3.1 RT: Revisar salida y entrada de objetos de datos inferidos en la tarea 1.1.2;
1.1.3.2 Inferir un modelo de funciones/comportamientos;
1.1.3.3 RT: Revisar funciones/comportamientos con cliente y revisar según se requiera;
endtask Task 1.1.3
1.1.4 Aislar aquellos elementos de la tecnología que se va a implementar en el software;
1.1.5 Investigar disponibilidad de software existente;
1.1.6 Definir factibilidad técnica;
1.1.7 Hacer estimación rápida de tamaño;
1.1.8 Crear una definición de ámbito;
endtask definition: Acción 1.1

Las tareas y subtareas anotadas en el refinamiento del lenguaje de diseño de proceso forman la base de un calendario detallado para la determinación del ámbito del concepto.

27.4 Definición de una red de tareas

PUNTO CLAVE
La red de tareas es un mecanismo útil para mostrar las dependencias intertarea y determinar la ruta crítica.

Las tareas y subtareas individuales tienen interdependencias en función de su secuencia. Además, cuando más de una persona está involucrada en un proyecto de ingeniería del software, es probable que las actividades y tareas de desarrollo se realicen en paralelo. Cuando esto ocurre, las tareas concurrentes deben coordinarse de modo que se completen en el momento en el que las tareas posteriores requieran sus productos operativos.

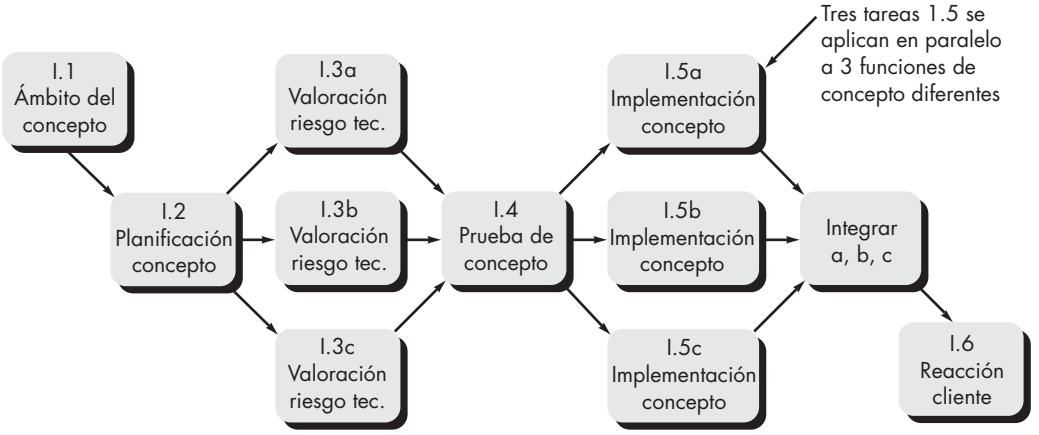
Una *red de tareas*, también llamada *red de actividad*, es una representación gráfica del flujo de tareas para un proyecto. En ocasiones se usa como el mecanismo mediante el cual la secuencia y las dependencias de tareas se integran en una herramienta automatizada de calendarización de proyecto. En su forma más simple (usada cuando se crea un calendario macroscópico), la red de tareas muestra las principales acciones de la ingeniería del software. La figura 27.2 presenta una red de tareas esquemática para un proyecto de desarrollo de concepto.

La naturaleza concurrente de las acciones de ingeniería de software conduce a algunos importantes requerimientos de calendarización. Puesto que las tareas paralelas ocurren de manera asíncrona, debe determinar las dependencias intertarea para asegurar el progreso continuo hacia la conclusión. Además, debe estar al tanto de aquellas tareas que se encuentren en la *ruta crítica*, es decir, aquellas que deben concluirse conforme al calendario si el proyecto como un todo debe completarse de acuerdo con ese calendario. Estos temas se estudian con más detalle más adelante, en este capítulo.

Es importante observar que la red de tareas que se muestra en la figura 27.2 es macroscópica. En una red de tareas detallada (un precursor de un calendario detallado), cada acción que se muestra en la figura se expandirá. Por ejemplo, la tarea 1.1 se expandirá para mostrar todas las tareas detalladas en el refinamiento de las acciones 1.1 que se muestran en la sección 27.3.2.

7 RT indica que debe realizarse una revisión técnica (capítulo 15).

FIGURA 27.2
Red de tareas para desarrollo de concepto



27.5 CALENDARIZACIÓN

Cita:

"Todo lo que debemos decidir es qué hacer con el tiempo que nos dan."

Gandalf en *El señor de los anillos: La comunidad del anillo*

La calendarización de un proyecto de software no difiere enormemente de la de cualquier esfuerzo de ingeniería multitarea. Por tanto, las herramientas y técnicas generalizadas de calendarización de proyecto pueden aplicarse con pocas modificaciones para proyectos de software.

La *evaluación del programa y la técnica de revisión* (PERT, por sus siglas en inglés) y el *método de ruta crítica* (CPM, por sus siglas en inglés) son dos métodos de calendarización de proyecto que pueden aplicarse en el desarrollo de software. Ambas técnicas se impulsan mediante información ya desarrollada en actividades de planificación de proyectos anteriores: estimaciones de esfuerzo, una descomposición de la función del producto, la selección del modelo de proceso y el conjunto de tareas adecuadas, así como la descomposición de las tareas que se seleccionan.

Las interdependencias entre tareas pueden definirse usando una red de tareas. Las tareas, en ocasiones llamadas *estructura de distribución del trabajo* del proyecto (EDT), se definen para el producto como un todo o para funciones individuales.

Tanto la PERT como el CPM proporcionan herramientas cuantitativas que permiten: 1) determinar la ruta crítica (la cadena de tareas que determina la duración del proyecto), 2) establecer estimaciones de tiempo "más probables" para tareas individuales aplicando modelos estadísticos y 3) calcular "tiempos frontera" que definen una "ventana" de tiempo para una tarea particular.

27.5.1 Cronogramas

Cuando se crea un calendario de proyecto de software, se comienza con un conjunto de tareas (la estructura de distribución del trabajo). Si se usan herramientas automatizadas, la distribución del trabajo se ingresa como una red o esbozo de tareas. Luego se ingresan esfuerzo, duración y fecha de inicio para cada tarea. Además, las tareas pueden asignarse a individuos específicos.

PUNTO CLAVE
Un cronograma le permite determinar qué tareas se realizarán en un momento determinado.

Como consecuencia de esta entrada se genera un *cronograma*, también llamado *gráfico de Gantt*. Es posible desarrollar un cronograma para todo el proyecto. Alternativamente, pueden generarse gráficos separados para cada función del proyecto o para cada individuo que trabaje en el proyecto.

La figura 27.3 ilustra el formato de un cronograma. Muestra una parte de un calendario de proyecto de software que enfatiza la tarea de formación del ámbito del concepto para un producto de software procesador de palabras (PP). Todas las tareas del proyecto (para el ámbito del

HERRAMIENTAS DE SOFTWARE

Calendarización del proyecto

Objetivo: El objetivo de las herramientas de calendarización del proyecto es permitir a un gerente de proyecto definir las tareas del trabajo, establecer sus dependencias, asignar recursos humanos a las tareas y desarrollar varios gráficos, cuadros y tablas que ayuden a monitorear y controlar el proyecto de software.

Mecánica: En general, las herramientas de calendarización de proyecto requieren la especificación de una estructura de distribución de trabajo de tareas o la generación de una red de tareas. Una vez definida la distribución o red de tareas (un esbozo), a cada una se le confieren fechas de inicio y de término, recursos humanos, fechas límite y otros datos. Entonces la herramienta genera una variedad de cronogramas y otras tablas que permiten a un gerente valorar el flujo de tareas de un proyecto. Dichos datos pueden actualizarse de manera continua conforme el proyecto avance.

Herramientas representativas:[8]

AMS Realtime, desarrollada por Advanced Management Systems (**www.amsusa.com**), proporciona capacidades de calendarización para proyectos de todos los tamaños y tipos.

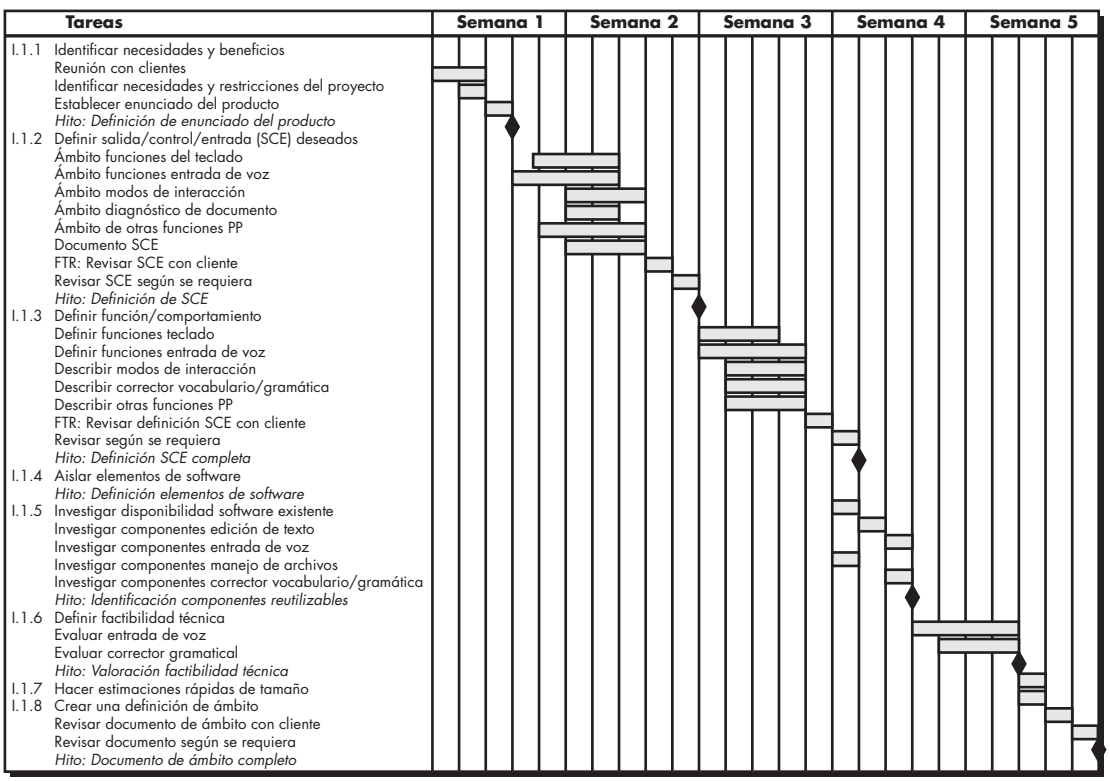
Microsoft Project, desarrollada por Microsoft (**www.microsoft.com**), es la herramienta de calendarización de proyecto basada en PC más ampliamente usada.

4C, desarrollada por 4C Systems (**www.4csys.com**), soporta todos los aspectos de planificación de proyecto, incluida la calendarización.

Una lista exhaustiva de proveedores y productos de software de gestión de proyectos puede encontrarse en **www.infogoal.com/pmc/pmcswr.htm**

concepto) se mencionan en la columna de la izquierda. Las barras horizontales indican la duración de cada tarea. Cuando muchas barras ocurren al mismo tiempo en el calendario, implican la concurrencia de tareas. Los diamantes indican hitos.

FIGURA 27.3
Ejemplo de cronograma



8 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas que hay en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

FIGURA 27.4
Ejemplo de tabla de proyecto

Tareas	Inicio planeado	Inicio real	Conclusión planeada	Conclusión real	Persona asignada	Esfuerzo asignado	Notas
I.1.1 Identificar necesidades y beneficios							Definir el ámbito requerirá más esfuerzo/tiempo
Reunión con clientes	sem1, d1	sem1, d1	sem1, d2	sem1, d2	BLS	2 p-d	
Identificar necesidades y restricciones del proyecto	sem1, d2	sem1, d2	sem1, d2	sem1, d2	JPP	1 p-d	
Establecer enunciado del producto	sem1, d3	sem1, d3	sem1, d3	sem1, d3	BLS/JPP	1 p-d	
Hito: Definición de enunciado del producto	sem1, d3	sem1, d3	sem1, d3	sem1, d3			
I.1.2 Definir salida/control/entrada (SCE) deseados							
Ámbito funciones del teclado	sem1, d4	sem1, d4	sem2, d2		BLS	1.5 p-d	
Ámbito funciones entrada de voz	sem1, d3	sem1, d3	sem2, d2		JPP	2 p-d	
Ámbito modos de interacción	sem2, d1		sem2, d3		MLL	1 p-d	
Ámbito diagnóstico de documento	sem2, d1		sem2, d2		BLS	1.5 p-d	
Ámbito de otras funciones PP	sem1, d4	sem1, d4	sem2, d3		JPP	2 p-d	
Documento SCE	sem2, d1		sem2, d3		MLL	3 p-d	
FTR: Revisar SCE con cliente	sem2, d3		sem2, d3		all	3 p-d	
Revisar SCE según se requiera	sem2, d4		sem2, d4		all	3 p-d	
Hito: Definición de SCE	sem2, d5		sem2, d5				
I.1.3 Definir función/comportamiento							

Una vez ingresada la información necesaria para la generación del cronograma, la mayoría de las herramientas de calendarización de proyecto de software producen *tablas de proyecto*, un listado tabular de todas las tareas del proyecto, sus fechas de inicio y término, planeadas y reales, y otra información relacionada (figura 27.4). Utilizadas en conjunto con el cronograma, las tablas de proyecto permiten monitorear el progreso.

27.5.2 Seguimiento del calendario

Cita:

"La regla básica del reporte de estado del software puede resumirse en una sola frase: 'sin sorpresas'."

Capers Jones

Si se desarrolló de manera adecuada, el calendario del proyecto se convierte en un mapa de caminos que define las tareas e hitos que se van a monitorear y controlar conforme el proyecto avance. El seguimiento puede lograrse en varias formas diferentes:

- Realizar reuniones periódicas del estado del proyecto, en las que cada miembro del equipo reporte avances y problemas
- Evaluar los resultados de todas las revisiones realizadas a través del proceso de ingeniería del software
- Determinar si los hitos formales del proyecto (los diamantes que se muestran en la figura 27.3) se lograron en la fecha prevista
- Comparar la fecha de inicio real con la fecha de inicio planeada para cada tarea de proyecto mencionada en la tabla de recursos (figura 27.4)
- Reunirse informalmente con los profesionales para obtener su valoración subjetiva del avance a la fecha y los problemas en el horizonte
- Usar análisis de valor ganado (sección 27.6) para valorar cuantitativamente el avance

CONSEJO

El mejor indicio de avance es la conclusión y la revisión exitosa de un producto operativo de software definido.

En realidad, todas estas técnicas de seguimiento las usan los gerentes de proyecto experimentados.

El control lo emplea un gerente de proyecto de software para administrar los recursos del proyecto, enfrentar los problemas y dirigir al personal del proyecto. Si las cosas van bien (es decir, si el proyecto avanza conforme el calendario y dentro de presupuesto, las revisiones indican que se realiza progreso real y que se alcanzan los hitos), el control es ligero. Pero cuando ocurren problemas, debe ejercerse el control para reconciliar los elementos discordantes tan rápidamente como sea posible. Después de diagnosticar un problema pueden enfocarse recur-

sos adicionales sobre el área problemática: puede reasignarse al personal o redefinirse el calendario del proyecto.

Cuando se enfrentan a severa presión debido a la fecha límite, los gerentes experimentados en ocasiones usan un calendario de proyecto y técnica de control llamado *time-boxing* (encuadre temporal) [Jal04]. La estrategia *time-boxing* reconoce que el producto completo no puede entregarse en la fecha límite preestablecida. Por tanto, se elige un paradigma de software incremental (capítulo 2) y se establece un calendario para cada entrega incremental.

PUNTO CLAVE

Cuando se alcanza la fecha de conclusión definida de una tarea encuadrada en el tiempo, el trabajo cesa para dicha tarea y comienza la siguiente.

Las tareas asociadas con cada incremento se encuadran en el tiempo. Esto significa que el calendario para cada tarea se ajusta trabajando en reversa desde la fecha de entrega hasta el momento del incremento. Alrededor de cada tarea se pone un "recuadro". Cuando una tarea llega a la frontera de su encuadre temporal (más o menos 10 por ciento), el trabajo se detiene y comienza la siguiente tarea.

Con frecuencia, la reacción inicial ante el enfoque *time-boxing* es negativa: "Si el trabajo no se termina, ¿cómo puedo avanzar?". La respuesta se encuentra en la forma en la que se logra el trabajo. Para cuando se llega a la frontera del encuadre temporal, es probable que 90 por ciento de la tarea esté completa.[9] El restante 10 por ciento, aunque importante, puede 1) demorarse hasta el siguiente incremento o 2) completarse más tarde si se requiere. En lugar de quedarse "atascado" en una tarea, el proyecto avanza hacia la fecha de entrega.

27.5.3 Seguimiento del progreso para un proyecto OO

Aunque un modelo iterativo es el mejor marco conceptual para un proyecto OO, el paralelismo de tareas hace difícil el seguimiento del proyecto. Acaso se tengan dificultades al establecer hitos significativos para un proyecto OO, debido a que algunas cosas diferentes ocurren a la vez. En general, los siguientes hitos importantes pueden considerarse "completos" cuando se satisfacen los criterios anotados.

Hitos técnicos: análisis OO completo

- Definición y revisión de todas las clases y de la jerarquía de clases.
- Definición y revisión de los atributos de clase y de las operaciones asociadas.
- Establecimiento y revisión de las relaciones de clase (capítulo 6).
- Creación y revisión de un modelo de comportamiento (capítulo 7).
- Anotación de las clases reutilizables.

Hitos técnicos: diseño OO completo

- Definición y revisión del conjunto de subsistemas.
- Asignación de clases a subsistemas y su revisión.
- Establecimiento y revisión de la asignación de tareas.
- Identificación de responsabilidades y colaboraciones.
- Diseño y revisión de atributos y operaciones.
- Creación y revisión del modelo de comunicación.

Hitos técnicos: programación OO completa

- Implementación en código de cada nueva clase, a partir del modelo de diseño.
- Implementación de las clases extraídas (a partir de la librería de reutilización).
- Construcción de prototipo o incremento.

9 Un cínico puede recordar el dicho: "el primer 90 por ciento del sistema requiere 90 por ciento del tiempo; el restante 10 por ciento del sistema requiere 90 por ciento del tiempo".

Hitos técnicos: prueba de OO

- Revisión de la exactitud y completitud de los modelos de análisis y diseño OO.
- Desarrollo y revisión de una red clase-responsabilidad-colaboración (capítulo 6).
- Diseño de casos de prueba y realización de pruebas en el nivel de clase (capítulo 19) para cada clase.
- Diseño de casos de prueba, conclusión de pruebas de grupo (capítulo 19) e integración de clases.
- Conclusión de pruebas en el nivel de sistema.

CONSEJO

Depuración y pruebas ocurren en concierto mutuo. El estado de la depuración frecuentemente se valora considerando el tipo y número de errores "abiertos" (bugs).

Recuerde que el modelo de proceso OO es iterativo: cada uno de estos hitos puede revisarse nuevamente conforme diferentes incrementos se entreguen al cliente.

27.5.4 Calendarización para proyectos webapp

La *calendarización de proyectos webapp* distribuye el esfuerzo estimado a través de la línea temporal planeada (duración) para construir cada incremento de la *webapp*. Esto se logra asignando el esfuerzo a tareas específicas. Sin embargo, es importante observar que el calendario *webapp* global evoluciona con el tiempo. Durante la primera iteración se desarrolla un calendario macroscópico. Este tipo de calendario identifica todos los incrementos de la *webapp* y proyecta las fechas en las que se desplegará cada una. Conforme el desarrollo de un incremento está en marcha, la entrada para el incremento en el calendario macroscópico se refina en un calendario detallado. Aquí se identifican y calendarizan tareas de desarrollo específicas (requeridas para lograr una actividad).

Como ejemplo de calendarización macroscópica, considere la *webapp* **CasaSeguraAsegurada.com**. Si recuerda las discusiones anteriores acerca de **CasaSeguraAsegurada.com**, es posible identificar siete incrementos para el componente del proyecto basado en web:

Incremento 1: Información básica de compañía y producto

Incremento 2: Información detallada de producto y descargas

Incremento 3: Citas de producto y procesamiento de pedidos de producto

Incremento 4: Plantilla espacial y diseño de sistema de seguridad

Incremento 5: Servicios de información y solicitud de monitoreo

Incremento 6: Control en línea del equipo de monitoreo

Incremento 7: Acceso a información de cuenta

El equipo consulta y negocia con los participantes y desarrolla un calendario de despliegue *preliminar* para los siete incrementos. En la figura 27.5 se muestra un cronograma para este calendario.

Es importante observar que las fechas de despliegue (representadas mediante diamantes en el cronograma) son preliminares y pueden cambiar conforme ocurran calendarizaciones más detalladas de los incrementos. No obstante, este calendario macroscópico brinda a la administración un indicio acerca de cuándo estará disponible el contenido y la funcionalidad, así como cuándo estará completo todo el proyecto. Como estimación preliminar, el equipo trabajará para desplegar todos los incrementos con un cronograma de 12 semanas. También vale la pena observar que algunos de los incrementos se desarrollarán en paralelo (por ejemplo, los incrementos 3, 4, 6 y 7). Esto supone que el equipo tendrá suficiente personal para hacer este trabajo paralelo.

Una vez desarrollado el calendario macroscópico, el equipo está listo para calendarizar las tareas del trabajo para un incremento específico. Para lograr esto, puede usar un marco conceptual de proceso genérico que sea aplicable a todos los incrementos de la *webapp*. Con las tareas

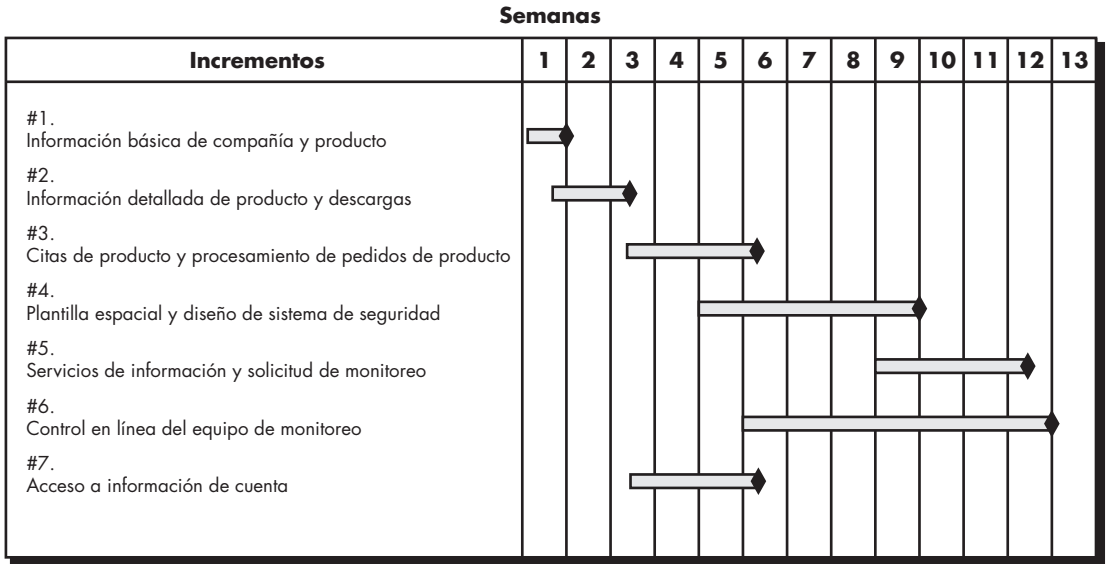


FIGURA 27.5
Cronograma para calendario de proyecto macroscópico

genéricas inferidas como parte del marco conceptual, se crea una *lista de tareas* como punto de partida y luego se adapta considerando el contenido y las funciones que se van a inferir para un incremento específico de la *webapp*.

Cada acción de marco conceptual (y sus tareas relacionadas) pueden adaptarse en una de cuatro formas: 1) una tarea se aplica como es, 2) una tarea se elimina porque no es necesaria para el incremento, 3) se agrega una nueva tarea (a la medida) y 4) una tarea se refina (elabora) en algunas subtareas nominales y cada una se vuelve parte del calendario.

Para ilustrar, considere una acción de *modelado de diseño* genérica para *webapps* que puede lograrse al aplicar alguna o todas las tareas siguientes:

- Diseño de la estética para la *webapp*.
- Diseño de la interfaz.
- Diseño del esquema de navegación.
- Diseño de la arquitectura de la *webapp*.
- Diseño del contenido y la estructura que lo soporta.
- Diseño de componentes funcionales.
- Diseño de mecanismos de seguridad y privacidad adecuados.
- Revisión del diseño.

Como ejemplo, considere la tarea genérica *Diseño de la interfaz* como se aplica al cuarto incremento de **CasaSeguraAsegurada.com**. Recuerde que el cuarto incremento implementa el contenido y la función para describir el espacio habitable o empresarial que se va a asegurar con el sistema de seguridad *CasaSegura*. En la figura 27.5, el cuarto incremento comienza al principio de la quinta semana y termina al final de la novena.

No hay duda de que la tarea *Diseño de la interfaz* debe realizarse. El equipo reconoce que el diseño de la interfaz es crucial para el éxito del incremento y decide refinar (elaborar) la tarea. Las siguientes subtareas se infieren para la tarea *Diseño de la interfaz* para el cuarto incremento:

- Desarrollo de un bosquejo de la plantilla de la página para la página de diseño del espacio.

- Revisión de plantilla con participantes.
- Diseño de mecanismos de navegación en la plantilla espacial.
- Diseño de plantilla "tablero de dibujo".[10]
- Desarrollo de detalles de procedimiento para la función de plantilla de pared gráfica.
- Desarrollo de detalles de procedimiento para el cálculo de longitud de pared y función de despliegue.
- Desarrollo de detalles de procedimiento para la función de plantilla de ventana gráfica.
- Desarrollo de detalles de procedimiento para la función de plantilla de puerta gráfica.
- Diseño de mecanismos para seleccionar componentes del sistema de seguridad (sensores, cámaras, micrófonos, etc.).
- Desarrollo de detalles de procedimiento para la plantilla gráfica de los componentes del sistema de seguridad.
- Realización de un par de recorridos según se requiera.

Estas tareas se vuelven parte del calendario de incrementos para el cuarto incremento de la aplicación Web y se asignan en el calendario de desarrollo de incrementos. Pueden ingresarse a software de calendarización y usarse para seguimiento y control.

CasaSegura

Seguimiento de calendario

La escena: Oficina de Doug Miller antes de iniciar el proyecto de software CasaSegura.

Participantes: Doug Miller (gerente del equipo de ingeniería de software CasaSegura) y Vinod Raman, Jamie Lazar y otros miembros del equipo de ingeniería de software del producto.

La conversación:

Doug (observa una diapositiva powerpoint): El calendario para el primer incremento de CasaSegura parece razonable, pero vamos a tener problemas para monitorear el progreso.

Vinod (con una mirada preocupada): ¿Por qué? Tenemos tareas calendarizadas diariamente, llenas de productos operativos, y nos aseguramos de no asignar demasiados recursos.

Doug: Todo está bien, ¿pero cómo sabemos cuándo está completo el modelo de requerimientos para el primer incremento?

Jamie: Las cosas son iterativas, así que no es difícil.

Doug: Lo entiendo, pero... bueno, por ejemplo, considera la "definición de clases de análisis". Indicaste esto como un hito.

Vinod: Sí.

Doug: ¿Quién determina eso?

Jamie (agravado): Se hace cuando esté lista.

Doug: Eso no es suficientemente bueno, Jamie. Tenemos que calendarizar las RT [revisiones técnicas, capítulo 15] y no lo has hecho. La conclusión exitosa de una revisión en el modelo de análisis, por ejemplo, es un hito razonable. ¿Entendido?

Jamie (frunce el ceño): Está bien, de vuelta al tablero de dibujo.

Doug: No debería tomar más de una hora hacer las correcciones... todos los demás pueden comenzar ahora.

27.6 Análisis de valor ganado

PUNTO CLAVE

El valor ganado proporciona un indicio cuantitativo del progreso.

En la sección 27.5 se estudiaron algunos enfoques cualitativos sobre el seguimiento del proyecto. Cada uno proporciona al gerente del proyecto un indicio del progreso, pero una valoración de la información proporcionada es un poco subjetiva. Es razonable preguntar si existe una técnica cuantitativa para valorar el progreso conforme el equipo de software avanza a través de

10 En esta etapa, el equipo vislumbra la creación del espacio al literalmente dibujar las paredes, ventanas y puertas usando funciones gráficas. Las líneas de pared "encajarán" en puntos de agarre. Las dimensiones de la pared se desplegarán de manera automática. Ventanas y puertas se colocarán gráficamente. El usuario final también podrá seleccionar sensores, cámaras y otros elementos específicos, y colocarlos una vez definido el espacio.

las tareas asignadas en el calendario del proyecto. De hecho, sí existe una técnica para realizar el análisis cuantitativo del progreso. Se llama *análisis de valor ganado* (AVG). Humphrey [Hum95] analiza el valor ganado en los siguientes términos:

> El sistema de valor ganado ofrece una escala de valor común para toda tarea [de proyecto de software], sin importar el tipo de trabajo que se va a realizar. Se estiman las horas totales para hacer todo el proyecto y a cada tarea se le da un valor ganado con base en su porcentaje estimado del total.

Dicho en forma todavía más simple, el valor ganado es una medida de progreso. Le permite valorar el "porcentaje de completitud" de un proyecto usando análisis cuantitativo en lugar de apoyarse en una corazonada. De hecho, Fleming y Koppleman [Fle98] argumentan que el análisis de valor ganado "proporciona lecturas precisas y confiables del rendimiento tan tempranamente como con 15 por ciento del proyecto". Para determinar el valor ganado se realizan los siguientes pasos:

? ¿Cómo se calcula el valor ganado y cómo se le usa para valorar el progreso?

1. El *costo presupuestado del trabajo calendarizado* (CPTC) se determina para cada tarea representada en el calendario. Durante la estimación se planifica el trabajo (en persona-horas o persona-días) de cada tarea de ingeniería del software. Por tanto, $CPTC_i$ es el esfuerzo planificado para la tarea *i*. Para determinar el progreso en un punto determinado a lo largo del calendario del proyecto, el valor de CPTC es la suma de los valores $CPTC_i$ para todas las tareas que deben estar completas en dicho punto en el tiempo señalado en el calendario del proyecto.
2. Los valores CPTC para todas las tareas se suman para inferir el *presupuesto al concluir* (PAC). Por tanto,

   PAC = Σ ($CPTC_k$) para todas las tareas *k*
3. A continuación, se calcula el *costo presupuestado del trabajo realizado* (CPTR). El valor de CPTR es la suma de los valores CPTC para todas las tareas que realmente se concluyeron en un punto en el tiempo sobre el calendario del proyecto.

Wilkens [Wil99] observa que "la distinción entre el CPTC y el CPTR es que el primero representa el presupuesto de las actividades que se planea completar y el último representa el presupuesto de las actividades que realmente se completaron". Determinados los valores para CPTC, PAC y CPTR, es posible calcular importantes indicadores del progreso:

$$\text{Índice de rendimiento en calendario, IRS} = \frac{\text{CPTR}}{\text{CPTC}}$$

$$\text{Variación en calendario, VC} = \text{CPTR} - \text{CPTC}$$

WebRef

En **www.acq.osd.mil/pm/** puede encontrarse una amplia variedad de recursos para análisis de valor ganado.

El IRS es un indicio de la eficiencia con la que el proyecto utiliza los recursos calendarizados. Un valor IRS cercano a 1.0 indica ejecución eficiente del calendario del proyecto. VC es simplemente un indicio absoluto de la variación con respecto al calendario planeado.

$$\text{Porcentaje calendarizado para conclusión} = \frac{\text{CPTC}}{\text{PAC}}$$

proporciona un indicio del porcentaje de trabajo que debe concluirse hacia el tiempo *t*.

$$\text{Porcentaje completo} = \frac{\text{CPTC}}{\text{PAC}}$$

proporciona un indicio cuantitativo del porcentaje de completitud del proyecto en un punto dado en el tiempo *t*.

También es posible calcular el *costo real del trabajo realizado* (CRTR). El valor para CRTR es la suma del esfuerzo realmente utilizado en las tareas que se completaron en un punto en el tiempo en el calendario del proyecto. Entonces es posible calcular

$$\text{Índice de rendimiento de costo, IRC} = \frac{\text{CPTR}}{\text{CRTR}}$$

$$\text{Variación en costo, VC} = \text{CPTR} - \text{CRTR}$$

Un valor IRC cercano a 1.0 proporciona un fuerte indicio de que el proyecto está dentro de su presupuesto definido. La VC es un indicio absoluto de ahorros en costo (contra los costos planificados) o déficits en una etapa particular de un proyecto.

Como otros radares en el horizonte, el análisis de valor ganado ilumina las dificultades en la calendarización antes de que puedan ser aparentes de otro modo. Esto permite tomar acciones correctivas antes de que se desarrolle una crisis en el proyecto.

27.7 Resumen

La calendarización es la culminación de una actividad de planificación que es un componente principal de la administración de proyectos de software. Cuando se combina con los métodos de estimación y análisis de riesgos, establece un mapa de caminos para el gerente del proyecto.

La calendarización comienza con la descomposición del proceso. Las características del proyecto se usan para adaptar un conjunto de tareas adecuado para el trabajo que se va a realizar. Una red de tareas muestra cada tarea de ingeniería, su dependencia de otras tareas y su duración proyectada. La red de tareas se usa para calcular la ruta crítica, un cronograma y otra información del proyecto. Al usar el calendario como guía, puede monitorearse y controlar cada paso en el proceso de software.

Problemas y puntos por evaluar

27.1. Las fechas límite "irracionales" son un hecho de la vida en el negocio del software. ¿Cómo debe proceder si se enfrenta con una?

27.2. ¿Cuál es la diferencia entre un calendario macroscópico y uno detallado? ¿Es posible administrar un proyecto si sólo se desarrolla un calendario macroscópico? Explique su respuesta.

27.3. ¿Puede existir un caso donde un hito de proyecto de software no se ligue a una revisión? Si es así, ofrezca uno o más ejemplos.

27.4. El "gasto en comunicación" puede ocurrir cuando múltiples personas trabajan en un proyecto de software. El tiempo que se emplea para comunicarse con los demás reduce la productividad individual (LOC/mes) y el resultado puede ser menos productividad para el equipo. Ilustre (cuantitativamente) cómo usan los ingenieros versados en las buenas prácticas de ingeniería de software revisiones técnicas que pueden aumentar la tasa de producción de un equipo (cuando se compara con la suma de las tasas de producción individuales) y qué tipo de revisiones técnicas usan. Sugerencia: Puede suponer que las revisiones reducen el reproceso y que el reproceso puede representar de 20 a 40 por ciento del tiempo de una persona.

27.5. Aunque agregar personas a un proyecto de software retrasado puede hacer que se retrase aún más, existen circunstancias en las que esto no es cierto. Descríbalas.

27.6. La relación entre personal y tiempo es enormemente no lineal. Con la ecuación de software de Putnam (descrita en la sección 27.2.2), desarrolle una tabla que relacione el número de personas con la duración del proyecto para un proyecto de software que requiere 50 000 LOC y 15 personas-años de esfuerzo (el parámetro de productividad es 5 000 y $B = 0.37$). Suponga que el software debe entregarse en más o menos 24 meses con una posibilidad de prórroga de 12 meses.

27.7. Suponga el lector que una universidad lo contrata para desarrollar un sistema para registrarse en línea a los cursos (OLCRS, según sus siglas en inglés). Primero, actúe como el cliente (si es estudiante, ¡debe resultarle sencillo!) y especifique las características de un buen sistema. (De manera alternativa, su instructor le proporcionará un conjunto de requerimientos preliminares para el sistema.) Con los métodos de estimación estudiados en el capítulo 26 desarrolle una estimación de esfuerzo y duración para OLCRS. Sugiera cómo:

a) Definiría actividades de trabajo paralelas durante el proyecto OLCRS.

b) Distribuiría el esfuerzo a través del proyecto.

c) Establecería hitos para el proyecto.

27.8. Seleccione un conjunto de tareas adecuado para el proyecto OLCRS.

27.9. Defina una red de tareas para el OLCRS descrito en el problema 27.7 o, alternativamente, para otro proyecto de software que sea de su interés. Asegúrese de mostrar tareas e hitos y de unir estimaciones de esfuerzo y duración a cada tarea. Si es posible, use una herramienta de calendarización automatizada para realizar este trabajo.

27.10. Si está disponible una herramienta de calendarización automatizada determine la ruta crítica para la red definida en el problema 27.9.

27.11. Con la herramienta de calendarización (si está disponible), o con papel y lápiz (si es necesario), desarrolle un cronograma para el proyecto OLCRS.

27.12. Suponga que usted es un gerente de proyecto de software y que se le pide calcular estadísticas de valor ganado para un pequeño proyecto de software. El proyecto tiene 56 tareas planeadas que se estima que requieren 582 personas-días para completarlas. En el momento en el que se le pide hacer el análisis de valor ganado, se han completado 12 tareas. Sin embargo, el calendario del proyecto indica que deberían estar completas 15. Están disponibles los siguientes datos de calendarización (en persona-días):

Tarea	Esfuerzo planeado	Esfuerzo real
1	12.0	12.5
2	15.0	11.0
3	13.0	17.0
4	8.0	9.5
5	9.5	9.0
6	18.0	19.0
7	10.0	10.0
8	4.0	4.5
9	12.0	10.0
10	6.0	6.5
11	5.0	4.0
12	14.0	14.5
13	16.0	—
14	6.0	—
15	8.0	—

Calcule IRS, variación en calendario, porcentaje calendarizado para conclusión, porcentaje completo, IRC y variación en costo para el proyecto.

Lecturas y fuentes de información adicionales

Virtualmente, todo libro escrito acerca de administración de proyectos de software contiene un análisis de la calendarización. Wysoki (*Effective Project Management,* Wiley, 2006), Lewis (*Project Planning Scheduling and Control,* 4a. ed., McGraw-Hill, 2006), Luckey y Phillips (*Software Project Management for Dummies,* For Dummies, 2006), Kerzner (*Project Management: A Systems Approach to Planning, Scheduling, and Controlling,* 9a. ed., Wiley, 2005), Hughes (*Software Project Management,* McGraw-Hill, 2005), The Project Management Institute (*PMBOK Guide,* 3a. ed., PMI, 2004), Lewin (*Better Software Project Management,* Wiley, 2001) y Bennatan (*On Time, Within Budget: Software Project Management Practices and Techniques,* 3a. ed., Wiley, 2000) contienen valiosos análisis sobre la materia. Aunque es específico para aplicación, Harris (*Planning and Scheduling Using Microsoft Office Project 2007*, Eastwood Harris Pty Ltd., 2007) proporciona un útil estudio sobre cómo pueden usarse las herramientas de calendarización para seguimiento y control exitosos de un proyecto de software.

Fleming y Koppelman (*Earned Value Project Management,* 3a ed., Project Management Institute Publications, 2006), Budd (*A Practical Guide to Earned Value Project Management,* Management Concepts, 2005) y

Webb y Wake (*Using Earned Value: A Project Manager's Guide,* Ashgate Publishing, 2003) analizan con detalle considerable el uso de las técnicas de valor ganado para planificación, seguimiento y control de proyectos.

En internet está disponible una gran variedad de fuentes de información acerca de calendarización de proyectos de software. Una lista actualizada de referencias en la World Wide Web que son relevantes para la calendarización de proyectos de software puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

CAPÍTULO

28 ADMINISTRACIÓN DEL RIESGO

En su libro acerca de administración y análisis de riesgos, Robert Charette [Cha89] presenta una definición conceptual de riesgo:

> Primero, el riesgo se preocupa por los acontecimientos futuros. Ayer y hoy están más allá de la preocupación activa, pues ya cosechamos lo que previamente se sembró por nuestras acciones pasadas. La cuestión tiene que ver, por tanto, con si podemos, al cambiar nuestras acciones de hoy, crear una oportunidad para una situación diferente y esperanzadoramente mejor para nosotros en el mañana. Esto significa, segundo, que el riesgo involucra cambio, como en los cambios de mentalidad, de opinión, de acciones o de lugares [...] [Tercero,] el riesgo involucra elección y la incertidumbre que ella conlleva. En consecuencia, paradójicamente, el riesgo, como la muerte y los impuestos, es una de las pocas certezas de la vida.

Cuando se considera el riesgo en el contexto de la ingeniería del software, los tres fundamentos conceptuales de Charette siempre están presentes. El futuro es su preocupación: ¿qué riesgos pueden hacer que el proyecto de software salga defectuoso? El cambio es lo que preocupa: ¿cómo afectan en los cronogramas y en el éxito global los cambios que puede haber en los requisitos del cliente, en las tecnologías de desarrollo, en los entornos meta y en todas las otras entidades conectadas con el proyecto? Por último, se debe lidiar con las opciones: ¿qué métodos y herramientas deben usarse, cuántas personas deben involucrarse, cuánto énfasis es "suficiente" poner en la calidad?

Peter Drucker [Dru75] dijo alguna vez: "aunque sea fútil intentar eliminar el riesgo, y cuestionable intentar minimizarlo, es esencial que los riesgos tomados sean los riesgos correctos".

UNA MIRADA RÁPIDA

¿Qué es? El análisis y la administración del riesgo son acciones que ayudan al equipo de software a entender y manejar la incertidumbre. Muchos problemas pueden plagar un proyecto de software. Un riesgo es un problema potencial: puede ocurrir, puede no ocurrir. Pero, sin importar el resultado, realmente es una buena idea identificarlo, valorar su probabilidad de ocurrencia, estimar su impacto y establecer un plan de contingencia para el caso de que el problema realmente ocurra.

¿Quién lo hace? Todos los involucrados en el proceso de software (gerentes, ingenieros de software y otros interesados) participan en el análisis y la administración del riesgo.

¿Por qué es importante? Piense en la consigna de los boy scouts: "estar preparados". El software es una empresa difícil. Muchas cosas pueden salir mal y, francamente, muchas con frecuencia lo hacen. Por esta razón es que estar preparado, comprender los riesgos y tomar medidas proactivas para evitarlos o manejarlos son elementos clave de una buena administración de proyecto de software.

¿Cuáles son los pasos? Reconocer qué puede salir mal es el primer paso, llamado "identificación de riesgos". A continuación, cada riesgo se analiza para determinar la probabilidad de que ocurra y el daño que causará si ocurre. Una vez establecida esta información se clasifican los riesgos, por probabilidad e impacto. Finalmente, se desarrolla un plan para manejar aquellos que tengan alta probabilidad y alto impacto.

¿Cuál es el producto final? Se produce un plan para mitigar, monitorear y manejar el riesgo (MMMR) o un conjunto de hojas de información de riesgo.

¿Cómo me aseguro de que lo hice bien? Los riesgos que se analizan y manejan deben inferirse a partir de un estudio del personal, el producto, el proceso y el proyecto. El MMMR debe revisarse conforme avance el proyecto para asegurarse de que los riesgos se mantienen actualizados. Los planes de contingencia para administración del riesgo deben ser realistas.

Antes de poder identificar los "riesgos correctos" que se van a tomar durante un proyecto de software, es importante identificar todos los que son obvios para gerentes y profesionales.

28.1 Estrategias reactivas de riesgo frente a estrategias proactivas de riesgo

Cita:

"Si no atacas de manera activa los riesgos, ellos te atacarán de manera activa."

Tom Gilb

Las estrategias *reactivas* de riesgo se han llamado irrisoriamente la "escuela de gestión de riesgo de Indiana Jones" [Tho92]. En las películas que llevan su nombre, Indiana Jones, cuando enfrenta una dificultad abrumadora, invariablemente dice: "No te preocupes, ¡pensaré en algo!". Al nunca preocuparse por los problemas hasta que suceden, Indy reaccionará en alguna forma heroica.

Tristemente, el gerente promedio de proyectos de software no es Indiana Jones y los miembros del equipo del proyecto de software no son sus fieles ayudantes. Sin embargo, la mayoría de los equipos de software se apoyan exclusivamente en estrategias reactivas de riesgo. Cuando mucho, una estrategia reactiva monitorea el proyecto para riesgos probables. Los recursos se hacen a un lado para lidiar con los riesgos, hasta que se convierten en problemas reales. De manera más común, el equipo de software no hace nada acerca de los riesgos hasta que algo sale mal. Entonces el equipo se apresura a entrar en acción con la intención de corregir el problema rápidamente. Con frecuencia esto se llama *modo bombero*. Cuando falla, la "administración de crisis" [Cha92] toma el control y el proyecto está en un peligro real.

Una estrategia considerablemente más inteligente para la administración del riesgo es ser proactivo. Una estrategia *proactiva* comienza mucho antes de iniciar el trabajo técnico. Los riesgos potenciales se identifican, su probabilidad e impacto se valoran y se clasifican por importancia. Luego, el equipo de software establece un plan para gestionar el riesgo. El objetivo principal es evitarlo, pero, dado que no todos los riesgos son evitables, el equipo trabaja para desarrollar un plan de contingencia que le permitirá responder en forma controlada y efectiva. A lo largo del resto de este capítulo se estudia una estrategia proactiva de gestión del riesgo.

28.2 Riesgos de software

Aunque hay un considerable debate acerca de la definición adecuada de riesgo de software, existe un acuerdo general en que los riesgos siempre involucran dos características: *incertidumbre* (el riesgo puede o no ocurrir; es decir, no hay riesgos 100 por ciento probables[1]) y *pérdida* (si el riesgo se vuelve una realidad, ocurrirán consecuencias o pérdidas no deseadas [Hig95]). Cuando se analizan los riesgos es importante cuantificar el nivel de incertidumbre y el grado de pérdidas asociados con cada riesgo. Para lograr esto, se consideran diferentes categorías de riesgos.

? ¿Qué tipos de riesgos es probable encontrar conforme se construye el software?

Los *riesgos del proyecto* amenazan el plan del proyecto, es decir, si los riesgos del proyecto se vuelven reales, es probable que el calendario del proyecto se deslice y que los costos aumenten. Los riesgos del proyecto identifican potenciales problemas de presupuesto, calendario, personal (tanto técnico como en la organización), recursos, participantes y requisitos, así como su impacto sobre un proyecto de software. En el capítulo 26, la complejidad, el tamaño y el grado de incertidumbre estructural del proyecto también se definieron como factores de riesgos para el proyecto (y la estimación).

Los *riesgos técnicos* amenazan la calidad y temporalidad del software que se va a producir. Si un riesgo técnico se vuelve una realidad, la implementación puede volverse difícil o imposible. Los riesgos técnicos identifican potenciales problemas de diseño, implementación, interfaz,

1 Un riesgo que es 100 por ciento probable es una restricción sobre el proyecto de software.

verificación y mantenimiento. Además, la ambigüedad en la especificación, la incertidumbre técnica, la obsolescencia técnica y la tecnología "de punta" también son factores de riesgo. Los riesgos técnicos ocurren porque el problema es más difícil de resolver de lo que se creía.

Los *riesgos empresariales* amenazan la viabilidad del software que se va a construir y con frecuencia ponen en peligro el proyecto o el producto. Los candidatos para los cinco principales riesgos empresariales son: 1) construir un producto o sistema excelente que realmente no se quiere (riesgo de mercado), 2) construir un producto que ya no encaje en la estrategia empresarial global de la compañía (riesgo estratégico), 3) construir un producto que el equipo de ventas no sabe cómo vender (riesgo de ventas), 4) perder el apoyo de los administradores debido a un cambio en el enfoque o en el personal (riesgo administrativo) y 5) perder apoyo presupuestal o de personal (riesgos presupuestales).

Es extremadamente importante observar que la categorización simple de riesgos no siempre funciona. Algunos de ellos son simplemente impredecibles por adelantado.

Otra categorización general de los riesgos es la propuesta por Charette [Cha89]. Los *riesgos conocidos* son aquellos que pueden descubrirse después de una evaluación cuidadosa del plan del proyecto, del entorno empresarial o técnico donde se desarrolla el proyecto y de otras fuentes de información confiables (por ejemplo, fecha de entrega irreal, falta de requisitos documentados o ámbito de software, pobre entorno de desarrollo). Los *riesgos predecibles* se extrapolan de la experiencia en proyectos anteriores (por ejemplo, rotación de personal, pobre comunicación con el cliente, disolución del esfuerzo del personal conforme se atienden las solicitudes de mantenimiento). Los *riesgos impredecibles* son el comodín en la baraja. Pueden ocurrir y lo hacen, pero son extremadamente difíciles de identificar por adelantado.

Cita:

"Los proyectos sin riesgos reales son perdedores. Casi siempre están desprovistos de beneficio; es por esto por lo que no se hicieron años atrás."

Tom DeMarco y Tim Lister

INFORMACIÓN

Siete principios de la administración de riesgos

El Software Engineering Institute (SEI) **(www.sei.cmu.edu)** identifica siete principios que "ofrecen un marco conceptual para lograr una administración de riesgo efectiva". Éstos son:

Mantener una perspectiva global: ver los riesgos del software dentro del contexto de un sistema donde el riesgo es un componente y el problema empresarial que se pretende resolver.

Tomar una visión de previsión: pensar en los riesgos que pueden surgir en el futuro (por ejemplo, debido a cambios en el software); establecer planes de contingencia de modo que los eventos futuros sean manejables.

Alentar la comunicación abierta: si alguien enuncia un riesgo potencial, no lo ignore. Si un riesgo se propone de manera informal, considérelo. Aliente a todos los participantes y usuarios a sugerir riesgos en cualquier momento.

Integrar: una consideración de riesgo debe integrarse en el proceso del software.

Enfatizar un proceso continuo: el equipo debe vigilar a lo largo del proceso de software, modificar los riesgos identificados conforme se conozca más información y agregar unos nuevos conforme se logre mejor comprensión.

Desarrollar una visión de producto compartida: si todos los participantes comparten la misma visión del software, es probable que haya mejor identificación y valoración del riesgo.

Alentar el trabajo en equipo: los talentos, habilidades y conocimientos de todos los participantes deben reunirse cuando se realicen actividades de administración de riesgos.

28.3 IDENTIFICACIÓN DE RIESGOS

La identificación de riesgos es un intento sistemático por especificar amenazas al plan del proyecto (estimaciones, calendario, carga de recursos, etc.). Al identificar los riesgos conocidos y predecibles, el gerente de proyecto da un primer paso para evitarlos cuando es posible y para controlarlos cuando es necesario.

Existen dos tipos distintos de riesgos para cada una de las categorías que se presentaron en la sección 28.2: riesgos genéricos y riesgos específicos del producto. Los *riesgos genéricos* son una amenaza potencial a todo proyecto de software. Los *riesgos específicos del producto* pueden

CONSEJO

Aunque es importante considerar los riesgos genéricos, son los riesgos específicos del producto los que provocan más dolores de cabeza. Asegúrese de emplear tiempo para identificar tantos riesgos específicos del producto como sea posible.

identificarse solamente por quienes tienen clara comprensión de la tecnología, el personal y el entorno específico del software que se construye. Para identificar los riesgos específicos del producto, examine el plan del proyecto y el enunciado de ámbito del software, y desarrolle una respuesta a la siguiente pregunta: ¿qué características especiales de este producto pueden amenazar el plan del proyecto?

Un método para identificar riesgos es crear una lista de verificación de ítem de riesgo. La lista de verificación puede usarse para identificación del riesgo y así enfocarse sobre algún subconjunto de riesgos conocidos y predecibles en las siguientes subcategorías genéricas:

- *Tamaño del producto:* riesgos asociados con el tamaño global del software que se va a construir o a modificar.
- *Impacto empresarial:* riesgos asociados con restricciones impuestas por la administración o por el mercado.
- *Características de los participantes:* riesgos asociados con la sofisticación de los participantes y con la habilidad de los desarrolladores para comunicarse con los participantes en forma oportuna.
- *Definición del proceso:* riesgos asociados con el grado en el que se definió el proceso de software y la manera como se sigue por parte de la organización desarrolladora.
- *Entorno de desarrollo:* riesgos asociados con la disponibilidad y calidad de las herramientas por usar para construir el producto.
- *Tecnología por construir:* riesgos asociados con la complejidad del sistema que se va a construir y con lo "novedoso" de la tecnología que se incluye en el sistema.
- *Tamaño y experiencia del personal:* riesgos asociados con la experiencia técnica y de proyecto global de los ingenieros de software que harán el trabajo.

La lista de verificación de ítem de riesgo puede organizarse en diferentes formas. Las preguntas relevantes en cada uno de los temas pueden responderse para cada proyecto de software. Las respuestas a dichas preguntas permiten estimar el impacto del riesgo. Un formato diferente de lista de verificación de ítem de riesgo simplemente menciona las características que son relevantes en cada subcategoría genérica. Finalmente, se menciona un conjunto de "componentes y promotores de riesgo" [AFC88] junto con sus probabilidades de ocurrencia. Los promotores de desempeño, apoyo, costo y calendario se analizan en respuesta a las preguntas anteriores.

Algunas listas de verificación exhaustivas para riesgo de proyecto de software están disponibles en la red (por ejemplo, [Baa07], [NAS07], [Wor04]). Puede usar dichas listas de verificación para comprender los riesgos genéricos para proyectos de software.

28.3.1 Valoración del riesgo de proyecto global

Las siguientes preguntas se infirieron de los datos de riesgo obtenidos al entrevistar en diferentes partes del mundo a gerentes de proyectos de software experimentados [Kei98]. Las preguntas se ordenan por su importancia relativa para el éxito del proyecto.

? ¿El proyecto de software en el que trabaja está en serio peligro?

1. ¿Los gerentes de software y de cliente se reunieron formalmente para apoyar el proyecto?
2. ¿Los usuarios finales se comprometen de manera entusiasta con el proyecto y con el sistema/producto que se va a construir?
3. ¿El equipo de ingeniería del software y sus clientes entienden por completo los requisitos?
4. ¿Los clientes se involucraron plenamente en la definición de los requisitos?
5. ¿Los usuarios finales tienen expectativas realistas?

6. ¿El ámbito del proyecto es estable?
7. ¿El equipo de ingeniería del software tiene la mezcla correcta de habilidades?
8. ¿Los requisitos del proyecto son estables?
9. ¿El equipo de proyecto tiene experiencia con la tecnología que se va a implementar?
10. ¿El número de personas que hay en el equipo del proyecto es adecuado para hacer el trabajo?
11. ¿Todas las divisiones de cliente/usuario están de acuerdo en la importancia del proyecto y en los requisitos para el sistema/producto que se va a construir?

WebRef

Risk radar es una base de datos y herramientas que ayudan a los gerentes a identificar, clasificar y comunicar riesgos de proyecto. Puede encontrarse en **www.spmn.com**

Si alguna de estas preguntas se responde de manera negativa deben establecerse sin falta pasos de mitigación, monitoreo y gestión. El grado en el que el proyecto está en riesgo es directamente proporcional al número de respuestas negativas a dichas preguntas.

28.3.2 Componentes y promotores de riesgo

La fuerza aérea estadounidense [AFC88] publicó un escrito que contiene excelentes lineamientos para la identificación y reducción de los riesgos de software. El enfoque de la fuerza aérea requiere que el gerente del proyecto identifique los promotores de riesgo que afectan los componentes de riesgo de software: rendimiento, costo, apoyo y calendario. En el contexto de este análisis, los componentes de riesgo se definen en la forma siguiente:

Cita:

"La administración del riesgo es administración de proyecto para adultos."

Tim Lister

- *Riesgo de rendimiento:* grado de incertidumbre de que el producto satisfará sus requisitos y se ajustará al uso pretendido.
- *Riesgo de costo:* grado de incertidumbre de que el presupuesto del proyecto se mantendrá.
- *Riesgo de apoyo:* grado de incertidumbre de que el software resultante será fácil de corregir, adaptar y mejorar.
- *Riesgo de calendario:* grado de incertidumbre de que el calendario del proyecto se mantendrá y de que el producto se entregará a tiempo.

El impacto de cada promotor de riesgo sobre el componente de riesgo se divide en una de cuatro categorías de impacto: despreciable, marginal, crítico o catastrófico. En la figura 28.1 [Boe89] se describe una caracterización de las potenciales consecuencias de errores (hileras con etiqueta 1) o de un fallo para lograr el resultado deseado (hileras con etiqueta 2). La categoría de impacto se elige con base en la caracterización que se ajusta mejor a la descripción en la tabla.

28.4 Proyección del riesgo

La *proyección del riesgo*, también llamada *estimación del riesgo*, intenta calificar cada riesgo en dos formas: 1) la posibilidad o probabilidad de que el riesgo sea real y 2) las consecuencias de los problemas asociados con el riesgo, en caso de que ocurra. Usted trabaja junto con otros gerentes y personal técnico para realizar cuatro pasos de proyección de riesgo:

1. Establecer una escala que refleje la probabilidad percibida de un riesgo.
2. Delinear las consecuencias del riesgo.
3. Estimar el impacto del riesgo sobre el proyecto y el producto.
4. Valorar la precisión global de la proyección del riesgo de modo que no habrá malos entendidos.

FIGURA 28.1
Valoración de impacto.
Fuente: [Boe89].

Componentes / Categoría		Rendimiento	Apoyo	Costo	Calendario
Catastrófico	1	La falla para satisfacer el requisito resultaría en fallo en la misión		La falla da como resultado aumento de costos y demoras en el calendario, con valores esperados en exceso de US$500K	
	2	Degradación significativa para no lograr el rendimiento técnico	Software que no responde o no puede tener apoyo	Significativos recortes financieros, probable agotamiento de presupuesto	IOC inalcanzable
Crítico	1	Falla para satisfacer el requisito degradaría el rendimiento del sistema hasta un punto donde el éxito de la misión sería cuestionable		La falla da como resultado demoras operativas y/o aumento de costos con valor esperado de US$100K a US$500K	
	2	Cierta reducción en rendimiento técnico	Demoras menores en modificaciones de software	Cierto recorte de recursos financieros, posible agotamiento	Posible deterioro en IOC
Marginal	1	Falla para satisfacer los requisitos resultaría en degradación de misión secundaria		Costos, impactos y/o calendario recuperable se deterioran con valor esperado de US$1K a US$100K	
	2	Reducción mínima a pequeña en rendimiento técnico	Apoyo de software receptivo	Suficientes recursos financieros	Calendario realista, alcanzable
Despreciable	1	Falla para satisfacer requisitos crearía inconvenientes o impacto no operativo		Error da como resultado costo menor y/o impacto en calendario con valor esperado de menos de US$1K	
	2	No reducción en rendimiento técnico	Software fácilmente soportable	Posible subejercicio de presupuesto	IOC alcanzable con facilidad

Nota: 1) La consecuencia potencial de errores o fallos de software no detectados.
2) La consecuencia potencial si el resultado deseado no se alcanza.

Piense duro acerca del software que está a punto de construir y pregúntese: ¿qué puede salir mal? Cree su propia lista y pida a otros miembros del equipo que hagan lo mismo.

La intención de estos pasos es considerar los riesgos de manera que conduzcan a una priorización. Ningún equipo de software tiene los recursos para abordar todo riesgo posible con el mismo grado de rigor. Al priorizar los riesgos es posible asignar recursos donde tendrán más impacto.

28.4.1 Elaboración de una tabla de riesgos

Una tabla de riesgos proporciona una técnica simple para proyección de riesgos.[2] Una tabla de muestra de riesgo se ilustra en la figura 28.2.

Comience por elaborar una lista de todos los riesgos (sin importar cuán remotos sean) en la primera columna de la tabla. Esto puede lograrse con la ayuda de las listas de verificación de ítem de riesgo mencionadas en la sección 28.3. Cada riesgo se clasifica en la segunda columna (por ejemplo, TP implica un riesgo de tamaño de proyecto, EMP implica un riesgo empresarial). La probabilidad de ocurrencia de cada riesgo se ingresa en la siguiente columna de la tabla. El valor de probabilidad para cada riesgo puede estimarse individualmente por los miembros del equipo. Una forma de lograr esto es encuestar a todos los miembros del equipo hasta que su valoración colectiva de la probabilidad del riesgo comience a convergir.

A continuación, se valora el impacto de cada riesgo. Cada componente de riesgo se valora usando la caracterización que se presenta en la figura 28.1 y se determina una categoría de

2 La tabla de riesgos puede implementarse como un modelo de hoja de cálculo. Esto permite fácil manipulación y ordenamiento de las entradas.

FIGURA 28.2
Ejemplo de tabla de riesgo previo al ordenamiento

Riesgos	Categoría	Probabilidad	Impacto	RMMM
Estimación de tamaño puede ser significativamente baja	PS	60%	2	
Mayor número de usuarios que el planificado	PS	30%	3	
Menos reuso que el planificado	PS	70%	2	
Usuarios finales que se resisten al sistema	BU	40%	3	
Fecha de entrega será apretada	BU	50%	2	
Pérdida de fondos	CU	40%	1	
Cliente cambiará requisitos	PS	80%	2	
Tecnología no satisfará las expectativas	TE	30%	1	
Falta de capacitación en herramientas	DE	80%	3	
Personal inexperto	ST	30%	2	
Alta rotación de personal	ST	60%	2	
Σ				
Σ				
Σ				

Valores de impacto:
1—catastrófico
2—crítico
3—marginal
4—despreciable

impacto. Las categorías para cada uno de los cuatro componentes de riesgo (rendimiento, apoyo, costo y calendario) se promedian[3] para determinar un valor de impacto global.

Una vez completadas las primeras cuatro columnas de la tabla de riesgos, la tabla se ordena por probabilidad y por impacto. Los riesgos de alta probabilidad y alto impacto se ubican en la parte superior de la tabla y los riesgos de baja probabilidad se ubican en el fondo. Esto logra una priorización de riesgo de primer orden.

Es posible estudiar la tabla ordenada resultante y definir una línea de corte. La *línea de corte* (dibujada horizontalmente en algún punto de la tabla) implica que sólo los riesgos que se encuentran por arriba de la línea recibirán mayor atención. Los riesgos que caen por abajo de la línea se vuelven a valorar para lograr una priorización de segundo orden. En la figura 28.3, el impacto y la probabilidad del riesgo tienen una influencia distinta sobre la preocupación de la administración. Un factor de riesgo que tenga un alto impacto pero una muy baja probabilidad de ocurrencia no debe absorber una cantidad significativa de tiempo administrativo. Sin embargo, los riesgos de alto impacto con probabilidad moderada alta y los riesgos de bajo impacto con alta probabilidad deben someterse a los siguientes pasos del análisis de riesgos.

PUNTO CLAVE
Una tabla de riesgos se ordena por probabilidad e impacto para clasificar riesgos.

Todos los riesgos que se encuentran por arriba de la línea de corte deben manejarse. La columna marcada MMMR contiene un apuntador al *plan de mitigación, monitoreo y manejo de riesgo* o, alternativamente, en una colección de hojas de información de riesgo desarrolladas para todos los riesgos que se encuentran arriba del corte. El plan MMMR y las hojas de información de riesgo se estudian en las secciones 28.5 y 28.6.

Cita:

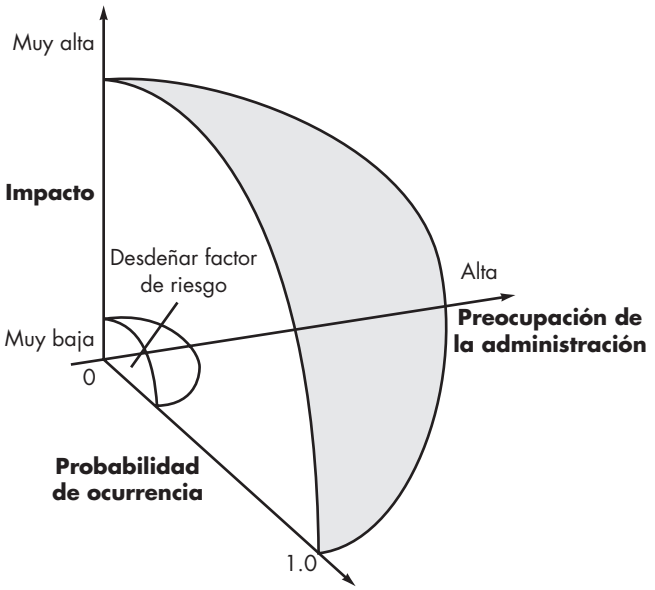
"[Hoy] nadie tiene el lujo de poder conocer una tarea tan bien como para que no contenga sorpresas, y las sorpresas significan riesgo."

Stephen Grey

La probabilidad del riesgo puede determinarse al hacer estimaciones individuales y luego desarrollar un solo valor de consenso. Aunque dicho enfoque es factible, se han desarrollado técnicas más sofisticadas para determinar la probabilidad del riesgo [AFC88]. Los promotores de riesgo pueden valorarse sobre una escala de probabilidad cualitativa que tenga los siguientes valores: imposible, improbable, probable y frecuente. Entonces puede asociarse probabilidad

3 Puede usar un promedio ponderado si un componente de riesgo tiene más significado para un proyecto.

FIGURA 28.3
Riesgo y preocupación de la administración



matemática con cada valor cualitativo (por ejemplo, una probabilidad de 0.7 a 0.99 implica un riesgo enormemente probable).

28.4.2 Valoración de impacto de riesgo

Tres factores afectan las probables consecuencias si ocurre un riesgo: su naturaleza, su ámbito y su temporización. La naturaleza del riesgo indica los problemas probables si ocurre. Por ejemplo, una interfaz externa pobremente definida en el hardware cliente (un riesgo técnico) impedirá el diseño y las pruebas tempranas, y probablemente conducirá más tarde a problemas de integración de sistema en un proyecto. El ámbito de un riesgo combina la severidad (¿cuán serio es?) con su distribución global (¿cuánto del proyecto se afectará o cuántos participantes se dañarán?). Finalmente, la temporización de un riesgo considera cuándo y por cuánto tiempo se sentirá el impacto. En la mayoría de los casos se quiere que las "malas noticias" ocurran tan pronto como sea posible, pero en algunos, mientras más se demoren, mejor.

? ¿Cómo se valoran las consecuencias de un riesgo?

Regrese una vez más al enfoque de análisis de riesgos que propuso la fuerza aérea estadounidense [AFC88]; puede aplicar los siguientes pasos para determinar las consecuencias globales de un riesgo: 1) determine la probabilidad promedio del valor de ocurrencia para cada componente de riesgo; 2) con la figura 28.1, determine el impacto para cada componente con base en los criterios mostrados, y 3) complete la tabla de riesgos y analice los resultados como se describe en las secciones anteriores.

La *exposición al riesgo* global, ER, se determina usando la siguiente relación [Hal98]:

$$\text{ER} = P \times C$$

donde P es la probabilidad de ocurrencia para un riesgo y C es el costo para el proyecto si ocurre el riesgo.

Por ejemplo, suponga que el equipo de software define un riesgo de proyecto en la forma siguiente:

Identificación de riesgo. De hecho, sólo 70 por ciento de los componentes de software calendarizados para reuso se integrarán en la aplicación. La funcionalidad restante tendrá que desarrollarse a la medida.

Probabilidad del riesgo. Un 80 por ciento (probable).

Impacto del riesgo. Se planificaron 60 componentes de software reutilizables. Si sólo puede usarse 70 por ciento, tendrán que desarrollarse 18 componentes desde cero (además de otro software a la medida que se calendarizó para desarrollo). Dado que el componente promedio es de 100 LOC y que los datos locales indican que el costo de la ingeniería del software para cada LOC es US$14.00, el costo global (impacto) para desarrollar los componentes sería $18 \times 100 \times 14 =$ US$25 200.

Exposición al riesgo. ER = $0.80 \times 25\ 200 \sim$ US$20 200.

Compare la ER para todos los riesgos con la estimación de costo para el proyecto. Si ER es mayor a 50 por ciento del costo del proyecto, debe evaluarse la viabilidad de éste.

La exposición al riesgo puede calcularse para cada riesgo en la tabla de riesgo, una vez hecha la estimación del costo del riesgo. La exposición al riesgo total para todos los riesgos (arriba del corte en la tabla de riesgos) puede proporcionar los medios para ajustar la estimación del costo final para un proyecto. También puede usarse para predecir el aumento probable en recursos de personal requeridos en varios puntos durante el calendario del proyecto.

La proyección del riesgo y las técnicas de análisis descritas en las secciones 28.4.1 y 28.4.2 se aplican de manera iterativa conforme avanza el proyecto de software. El equipo del proyecto debe revisar la tabla de riesgos a intervalos regulares, reevaluar cada riesgo para determinar cuándo nuevas circunstancias cambian su probabilidad e impacto. Como consecuencia de esta actividad, acaso sea necesario agregar nuevos riesgos a la tabla, eliminar algunos riesgos que ya no son relevantes e incluso cambiar las posiciones relativas de otros.

CasaSegura

Análisis de riesgos

La escena: Oficina de Doug Miller antes de comenzar el proyecto de software CasaSegura.

Participantes: Doug Miller (gerente del equipo de ingeniería del software CasaSegura) y Vinod Raman, Jamie Lazar y otros miembros del equipo de ingeniería de software del producto.

La conversación:

Doug: Me gustaría usar algo de tiempo en una lluvia de ideas para el proyecto CasaSegura.

Jamie: ¿Acerca de que puede salir mal?

Doug: Sip. Aquí hay algunas categorías donde las cosas pueden salir mal. [Muestra a todos las categorías anotadas en la introducción a la sección 28.3.]

Vinod: Hmmm... quieres que sólo las mencionemos o...

Doug: No. Esto es lo que creo que debemos hacer. Todo mundo haga una lista de riesgos... ahora...

[Transcurren diez minutos, todos escriben].

Doug: Muy bien, deténganse.

Jamie: ¡Pero no he terminado!

Doug: Está bien. Revisaremos la lista de nuevo. Ahora, para cada ítem en su lista, asignen un porcentaje de probabilidad de que ocurrirá el riesgo. Luego, asignen un impacto al proyecto sobre una escala de 1 (menor) a 5 (catastrófico).

Vinod: Si creo que el riesgo es un volado, especifico una probabilidad de 50 por ciento y, si creo que tendrá un impacto de proyecto moderado, especifico un 3, ¿cierto?

Doug: Exactamente.

[Transcurren cinco minutos, todos escriben].

Doug: Muy bien, deténganse. Ahora haremos una lista grupal en el pizarrón. Yo escribiré; cada uno de ustedes dirá una entrada de su lista.

[Transcurren quince minutos; crean la lista].

Jamie (apunta hacia el pizarrón y ríe): Vinod, ese riesgo (apunta hacia una entrada en el pizarrón) es ridículo. Hay una mayor probabilidad de que a todos nos caiga un rayo. Debemos removerlo.

Doug: No, dejémoslo por ahora. Consideremos todos los riesgos, sin importar cuán locos parezcan. Más tarde filtraremos la lista.

Jamie: Pero ya tenemos más de 40 riesgos... ¿cómo vamos a manejarlos todos?

Doug: No podemos. Es por eso por lo que definiremos un corte después de ordenarlos. Yo haré ese corte y nos reuniremos de nuevo mañana. Por ahora, regresen a trabajar... y en su tiempo libre piensen en cualquier riesgo que hayan olvidado.

28.5 REFINAMIENTO DEL RIESGO

Durante las primeras etapas de la planificación del proyecto, un riesgo puede enunciarse de manera muy general. Conforme pasa el tiempo y se aprende más acerca del proyecto y de los riesgos, es posible refinar el riesgo en un conjunto de riesgos más detallados, cada uno un poco más sencillo de mitigar, monitorear y manejar.

¿Cuál es una buena forma de describir un riesgo?

Una forma de hacer esto es representar el riesgo en formato *condición-transición-consecuencia* (CTC) [Glu94]. Es decir, el riesgo se enuncia en la forma siguiente:

> Dado que <condición> entonces hay preocupación porque (posiblemente) <consecuencia>.

Al usar el formato CTC para el riesgo de reutilización anotado en la sección 28.4.2, podría escribir:

> Dado que todos los componentes de software reutilizables deben apegarse a estándares de diseño específicos y dado que algunos no se apegan, entonces existe preocupación de que (posiblemente) sólo 70 por ciento de los módulos reutilizables planeados puedan realmente integrarse en el sistema que se va a construir, lo que da como resultado la necesidad de ingeniería a la medida del restante 30 por ciento de los componentes.

Esta condición general puede refinarse en la forma siguiente:

> **Subcondición 1.** Ciertos componentes reutilizables los desarrolló una tercera persona sin conocimiento de los estándares de diseño internos.
>
> **Subcondición 2.** El estándar de diseño para interfaces de componente todavía no se consolida y puede no apegarse a ciertos componentes reutilizables existentes.
>
> **Subcondición 3.** Ciertos componentes reutilizables se implementaron en un lenguaje que no se soporta en el entorno blanco.

Las consecuencias asociadas con estas subcondiciones refinadas permanecen iguales (es decir, 30 por ciento de componentes de software deben someterse a ingeniería a la medida), pero el refinamiento ayuda a aislar los riesgos subyacentes y puede conducir a análisis y respuestas más sencillos.

28.6 MITIGACIÓN, MONITOREO Y MANEJO DE RIESGO

Cita:

"Si tomo muchas precauciones, es porque no dejo nada al azar."

Napoleón

Todas las actividades de análisis de riesgos presentadas hasta el momento tienen una sola meta: auxiliar al equipo del proyecto a desarrollar una estrategia para lidiar con el riesgo. Una estrategia efectiva debe considerar tres temas: 1) evitar el riesgo, 2) monitorear el riesgo y 3) manejar el riesgo y planificar la contingencia.

Si un equipo de software adopta un enfoque proactivo ante el riesgo, evitarlo siempre es la mejor estrategia. Esto se logra desarrollando un plan para *mitigación del riesgo*. Por ejemplo, suponga que una alta rotación de personal se observa como un riesgo de proyecto r_1. Con base en la historia y la intuición administrativa, la probabilidad l_1 de alta rotación se estima en 0.70 (70 por ciento, más bien alta) y el impacto x_1 se proyecta como crítico, es decir, la alta rotación tendrá un impacto crítico sobre el costo y el calendario del proyecto.

Para mitigar este riesgo se desarrollará una estrategia a fin de reducir la rotación. Entre los posibles pasos por tomar están:

¿Qué puede hacerse para mitigar el riesgo?

- Reunirse con el personal actual para determinar las causas de la rotación (por ejemplo, pobres condiciones laborales, salario bajo, mercado laboral competitivo).
- Mitigar aquellas causas que están bajo su control antes de comenzar el proyecto.

- Una vez iniciado el proyecto, suponer que la rotación ocurrirá y desarrollar técnicas para asegurar la continuidad cuando el personal se vaya.
- Organizar equipos de trabajo de modo que la información acerca de cada actividad de desarrollo se disperse ampliamente.
- Definir estándares de producto operativo y establecer mecanismos para asegurar que todos los modelos y documentos se desarrollen en forma oportuna.
- Realizar revisiones de pares de todo el trabajo (de modo que más de una persona "se ponga al día").
- Asignar un miembro de personal de respaldo para cada técnico crítico.

Conforme avanza el proyecto, comienzan las actividades de *monitoreo de riesgos*. El gerente de proyecto monitorea factores que pueden proporcionar un indicio de si el riesgo se vuelve más o menos probable. En el caso de alta rotación de personal se monitorean: la actitud general de los miembros del equipo con base en presiones del proyecto, el grado en el que el equipo cuaja, relaciones interpersonales entre miembros del equipo, potenciales problemas con la compensación y beneficios, y la disponibilidad de empleos dentro de la compañía y fuera de ella.

Además de monitorear dichos factores, un gerente de proyecto debe dar seguimiento a la efectividad de los pasos de mitigación del riesgo. Por ejemplo, un paso de mitigación del riesgo anotado aquí requiere la definición de estándares de producto operativo y mecanismos para asegurarse de que los productos operativos se desarrollan en forma oportuna. Éste es un mecanismo para asegurar continuidad en caso de que un individuo crucial deje el proyecto. El gerente de proyecto debe monitorear los productos operativos cuidadosamente para asegurarse de que cada uno puede sostenerse por cuenta propia y que imparte información que sería necesaria si un recién llegado fuese forzado a unirse al equipo de software en alguna parte en medio del proyecto.

El *manejo del riesgo* y la *planificación de contingencia* suponen que los esfuerzos de mitigación fracasaron y que el riesgo se convirtió en realidad. Continuando con el ejemplo, el proyecto ya está en marcha y algunas personas anuncian que renunciarán al mismo. Si se siguió la estrategia de mitigación, está disponible el respaldo, la información se documentó y el conocimiento se dispersó a través del equipo. Además, puede cambiar temporalmente el foco de los recursos (y reajustar el calendario del proyecto) hacia aquellas funciones que tengan personal completo, lo que permitirá "ponerse al día" a los recién llegados que deban agregarse al equipo. A los individuos que se retiran se les pide detener todo el trabajo y pasar sus últimas semanas en "modo de transferencia de conocimiento". Esto puede incluir captura de conocimiento en video, desarrollo de "documentos comentados o wikis" y/o reuniones con otros miembros del equipo que permanecerán en el proyecto.

Si la ER para un riesgo específico es menor que el costo de mitigación de riesgo, no intente mitigar el riesgo, sino continuar para monitorearlo.

Es importante anotar que los pasos de mitigación, monitoreo y manejo del riesgo (MMMR) incurren en costos adicionales para el proyecto. Por ejemplo, emplear el tiempo en respaldar a cada técnico crucial cuesta dinero. Por tanto, parte del manejo de riesgos es evaluar cuándo los beneficios acumulativos por los pasos MMMR sobrepasan los costos asociados con su implementación. En esencia, se realiza un análisis clásico costo-beneficio. Si los pasos para evitar el riesgo debido a la alta rotación aumentarán tanto el costo del proyecto como la duración del mismo por un estimado de 15 por ciento, pero el factor de costo predominante es "respaldo", la administración puede decidir no implementar este paso. Por otra parte, si los pasos para evitar el riesgo se proyectan para aumentar los costos en 5 por ciento y la duración sólo en 3 por ciento, la administración probablemente pondrá todo en su lugar.

Para un proyecto grande pueden identificarse 30 o 40 riesgos. Si para cada uno se identifican entre tres y siete pasos de manejo de riesgo, ¡el manejo del riesgo puede convertirse en un proyecto por sí mismo! Por esta razón, debe adaptar al riesgo de software la regla de Pareto de 80-20. La experiencia indica que 80 por ciento del riesgo de proyecto global (es decir, 80 por

ciento del potencial para falla del proyecto) puede explicarse por sólo 20 por ciento de los riesgos identificados. El trabajo realizado durante los primeros pasos del análisis de riesgos ayudará a determinar cuáles de ellos residen en ese 20 por ciento (por ejemplo, riesgos que conducen a la exposición más alta al riesgo). Por esta razón, algunos de los riesgos identificados, valorados y proyectados pueden no llegar al plan MMMR, no se ubican en el crucial 20 por ciento (los riesgos con prioridad de proyecto más alta).

El riesgo no está limitado al proyecto de software en sí. Pueden ocurrir después de que el software se desarrolló exitosamente y de que se entregó al cliente. Dichos riesgos por lo general se asocian con las consecuencias de falla del software en el campo.

La *seguridad del software* y el *análisis de riesgos* (por ejemplo, [Dun02], [Her00], [Lev95]) son las actividades de aseguramiento de la calidad del software (capítulo 16) que se enfocan en la identificación y valoración de los riesgos potenciales que pueden afectar al software negativamente y hacer que falle todo un sistema. Si los riesgos pueden identificarse tempranamente en el proceso de ingeniería del software, pueden especificarse características de diseño del software que eliminarán o controlarán los riesgos potenciales.

28.7 El plan MMMR

En el plan de proyecto del software puede incluirse una estrategia de administración del riesgo, o los pasos de administración del riesgo pueden organizarse en un *plan de mitigación, monitoreo y manejo de riesgo* (MMMR) por separado. El plan MMMR documenta todo el trabajo realizado como parte del análisis de riesgos y el gerente del proyecto lo usa como parte del plan de proyecto global.

Algunos equipos de software no desarrollan un documento MMMR formal. En vez de ello, cada riesgo se documenta individualmente usando una *hoja de información de riesgo* (HIR) [Wil97]. En la mayoría de los casos, la HIR se mantiene con un sistema de base de datos de modo

HERRAMIENTAS DE SOFTWARE

Manejo de riesgo

Objetivo: El objetivo de las herramientas de manejo de riesgo es auxiliar de un equipo de proyecto para definir riesgos, valorar su impacto y probabilidad, y monitorear los riesgos a lo largo de un proyecto de software.

Mecánica: En general, las herramientas de manejo de riesgo auxilian en la identificación de riesgos genéricos al proporcionar una lista de riesgos empresariales y de proyecto usuales, proporcionar listas de verificación u otras técnicas de "entrevista" que auxilien en la identificación de riesgos específicos del proyecto, asignar probabilidad e impacto a cada riesgo, apoyar las estrategias de mitigación de riesgo y generar muchos reportes diferentes relacionados con el riesgo.

Herramientas representativas:[4]

@risk, desarrollada por Palisade Corporation (**www.palisade.com**), es una herramienta de análisis de riesgo genérico que usa simulación Monte Carlo para impulsar su motor analítico.

Riskman, distribuida por ABS Consulting (**www.absconsulting.com/riskmansoftware/index.html**), es un sistema experto de evaluación de riesgos que identifica riesgos relacionados con proyectos.

Risk Radar, desarrollada por SPMN (**www.spmn.com**), ayuda a los gerentes de proyecto a identificar y manejar riesgos de proyecto.

Risk+, desarrollada por Deltek (**www.deltek.com**), se integra con Microsoft Project para cuantificar incertidumbres de costo y calendario.

X:PRIMER, desarrollada por GrafP Technologies (**www.grafp.com**), es una herramienta genérica web que predice qué puede salir mal en un proyecto e identifica las causas raíz para potenciales fallos y contramedidas efectivas.

4 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas que hay en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

FIGURA 28.4
Hoja de información de riesgo.
Fuente: [Wil97].

Hoja de información de riesgo			
Riesgo ID: P02-4-32	Fecha: 5/9/09	Prob: 80%	Impacto: alto
Descripción: De hecho, sólo 70 por ciento de los componentes de software calendarizados para reuso se integrarán en la aplicación. La funcionalidad restante tendrá que desarrollarse a la medida.			
Refinamiento/contexto: Subcondición 1: Ciertos componentes reutilizables se desarrollaron por una tercera persona sin conocimiento de los estándares de diseño internos. Subcondición 2: El estándar de diseño para interfaces de componente no se consolidó y puede ser que no se apegue a ciertos componentes reutilizables existentes. Subcondición 3: Ciertos componentes reutilizables se implementaron en un lenguaje que no es soportado en el entorno meta.			
Mitigación/monitoreo: 1. Contactar tercera persona para determinar conformidad con los estándares de diseño. 2. Presionar por terminación de estándares de interfaz; considerar estructura de componente cuando se decida acerca de protocolo de interfaz. 3. Comprobar para determinar el número de componentes en la categoría de subcondición 3; comprobar para determinar si se puede adquirir soporte de lenguaje.			
Manejo/plan de contingencia/disparador: *ER* calculada en US$20 200. Asignar esta cantidad dentro de los costos de contingencia del proyecto. Desarrollar revisión de calendario y suponer que 18 componentes adicionales tendrán que construirse a la medida; asignar personal en concordancia. Disparador: Pasos de mitigación improductivos al 7/1/09.			
Estado actual: 5/12/09: Pasos de mitigación iniciados.			
Originador: D. Gagne		Asignado: B. Laster	

que la entrada de creación e información, el orden de prioridad, las búsquedas y otros análisis pueden realizarse con facilidad. El formato de la HIR se ilustra en la figura 28.4.

Una vez documentada la MMMR y comenzado el proyecto, inician los pasos de mitigación y monitoreo del riesgo. Como ya se estudió, la mitigación del riesgo es una actividad que busca evitar el problema. El monitoreo del riesgo es una actividad de seguimiento del proyecto con tres objetivos principales: 1) valorar si los riesgos predichos en efecto ocurren, 2) asegurar que los pasos para evitar el riesgo definidos para un riesgo determinado se aplican de manera correcta y 3) recopilar información que pueda usarse para futuros análisis de riesgos. En muchos casos, el problema que ocurre durante un proyecto puede monitorearse en más de un riesgo. Otra actividad del monitoreo de riesgos es intentar asignar orígenes (cuál riesgo causó cuál problema a lo largo del proyecto).

28.8 RESUMEN

Siempre que un colectivo cabalga en un proyecto de software, el sentido común dicta análisis de riesgos. E incluso así, la mayoría de los gerentes de proyectos de software lo hacen de manera informal y superficial, si acaso lo hacen. El tiempo que se emplea en identificar, analizar y manejar el riesgo rinde sus frutos en muchas formas: menos agitación durante el proyecto, una mayor capacidad para monitorear y controlar un proyecto, y la confianza que conlleva la planificación de los problemas antes de que se presenten.

El análisis de riesgos puede absorber una cantidad significativa del esfuerzo de planificación del proyecto. Identificación, proyección, valoración, manejo y monitoreo, todos requieren tiempo. Pero el esfuerzo vale la pena. Para citar a Sun Tzu, el general chino que vivió hace 2 500 años: "si conoces al enemigo y te conoces a ti mismo, no necesitas temer al resultado de cien batallas". Para el gerente de proyecto de software, el enemigo es el riesgo.

Problemas y puntos por evaluar

28.1. Proporcione cinco ejemplos de otros campos que ilustren los problemas asociados con una estrategia de riesgo reactiva.

28.2. Describa la diferencia entre "riesgos conocidos" y "riesgos predecibles".

28.3. Agregue tres preguntas o temas adicionales a cada una de las listas de comprobación de ítem de riesgo que se presentan en el sitio web de esta obra.

28.4. Se le pide construir software para apoyar un sistema de edición de video de bajo costo. El sistema acepta video digital como entrada, almacena el video en disco y luego permite al usuario aplicar una amplia variedad de ediciones al video digitalizado. Después, el resultado puede exhibirse mediante DVD u otros medios. Haga una pequeña cantidad de investigación acerca de sistemas de este tipo y luego elabore una lista de riesgos tecnológicos que enfrentaría mientras comienza un proyecto de este tipo.

28.5. Usted es el gerente de proyecto de una gran compañía de software. Se le pide dirigir un equipo que desarrolle software de procesamiento de palabra de "próxima generación". Cree una tabla de riesgo para el proyecto.

28.6. Describa la diferencia entre componentes de riesgo y promotores de riesgo.

28.7. Desarrolle una estrategia de mitigación de riesgo y actividades específicas de mitigación de riesgo para tres de los riesgos anotados en la figura 28.2.

28.8. Desarrolle una estrategia de monitoreo de riesgo y actividades específicas de monitoreo de riesgo para tres de los riesgos anotados en la figura 28.2. Asegúrese de identificar los factores que monitoreará para determinar si el riesgo se vuelve más o menos probable.

28.9. Desarrolle una estrategia de manejo de riesgo y actividades específicas de manejo de riesgo para tres de los riesgos anotados en la figura 28.2.

28.10. Intente refinar tres de los riesgos anotados en la figura 28.2 y luego cree hojas de información de riesgo para cada uno.

28.11. Represente tres de los riesgos anotados en la figura 28.2 usando un formato CTC.

28.12. Vuelva a calcular la exposición al riesgo que estudió en la sección 28.4.2 cuando costo/LOC es US$16 y la probabilidad es 60 por ciento.

28.13. ¿Puede pensar en una situación en la que un riesgo con alta probabilidad y alto impacto no se considerará como parte de su plan MMMR?

28.14. Describa las cinco áreas de aplicación de software en las que la seguridad del software y el análisis de riesgos serían una preocupación principal.

Lecturas y fuentes de información adicionales

La literatura de gestión del riesgo de software se expandió significativamente en las décadas anteriores. Vun (*Modeling Risk*, Wiley, 2006) presenta un tratamiento matemático detallado del análisis de riesgos que puede aplicarse a proyectos de software. Crohy *et al*. (*The Essentials of Risk Management,* McGraw-Hill, 2006), Mulcahy (*Risk Management, Tricks of the Trade for Project Managers,* RMC Publications, Inc., 2003), Kendrick (*Identifying and Managing Project Risk,* American Management Association, 2003), y Marrison (*The Fundamentals of Risk Measurement,* McGraw-Hill, 2002) presentan métodos y herramientas útiles que puede usar todo gerente de proyecto.

DeMarco y Lister (*Dancing with Bears*, Dorste House, 2003) escribieron un entretenido e inteligente libro que guía a los gerentes y profesionales del software a través de la gestión de riesgos. Moynihan (*Coping with IT/IS Risk Management*, Springer-Verlag, 2002) presenta consejos pragmáticos de gerentes de proyecto que lidian con el riesgo continuamente. Royer (*Project Risk Management*, Management Concepts, 2002) y Smith y Merrit (*Proactive Risk Management*, Productivity Press, 2002) sugieren un proceso proactivo para gestión del riesgo. Karolak (*Software Engineering Risk Management*, Wiley, 2002) escribió un manual que introduce un modelo de análisis de riesgo fácil de usar, con valiosas listas de comprobación y cuestionarios apoyados por un paquete de software.

Capers Jones (*Assesment and Control of Software Risks*, Prentice Hall, 1994) presenta un detallado análisis de los riesgos de software, que incluye datos recopilados de cientos de proyectos de software. Jones define 60 factores de riesgo que pueden afectar el resultado de los proyectos de software. Boehm [Boe89] sugiere excelentes cuestionarios y formatos de lista de verificación que pueden resultar invaluables en la identificación del riesgo. Charette [Cha89] presenta un tratamiento detallado de la mecánica del análisis de riesgos, y se apoya en teoría de probabilidad y técnicas estadísticas para analizar los riesgos. En otro volumen, Charette (*Application Strategies for Risk Analysis*, McGraw-Hill, 1990) analiza el riesgo en el contexto de la ingeniería de sistemas y de software, y sugiere estrategias pragmáticas para la gestión del riesgo. Gilb (*Principles of Software Engineering Management*, Addison-Wesley, 1988) presenta un conjunto de "principios" (que con frecuencia son entretenidos y en ocasiones profundos) que pueden servir como una valiosa guía para la gestión del riesgo.

Ewusi-Mensah (*Software Development Failures: Anatomy of Abandoned Projects*, MIT Press, 2003) y Yourdon (*Death March*, Prentice Hall, 1997) estudian lo que ocurre cuando los riesgos abruman a un equipo de proyecto de software. Bernstein (*Against the Gods*, Wiley, 1998) presenta una entretenida historia del riesgo, que se remonta a tiempos antiguos.

El Software Engineering Institute publicó muchos reportes detallados y manuales acerca del análisis y la gestión del riesgo. El panfleto AFSCP 800-45 del Air Force Systems Command [AFC88] describe la identificación del riesgo y técnicas para su reducción. Cada tema del *ACM Software Engineering Notes* tiene una sección titulada "Riesgos para el público" (editor, P. G. Neumann). Si quiere las más recientes y mejores historias de horror del software, éste es el lugar al que debe ir.

En internet está disponible una gran variedad de fuentes de información acerca de la gestión del riesgo de software. Una lista actualizada de referencias en la World Wide Web que son relevantes para la gestión del riesgo puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

CAPÍTULO

29

MANTENIMIENTO Y REINGENIERÍA

CONCEPTOS CLAVE

Sin importar su dominio de aplicación, su tamaño o su complejidad, el software de computadora evolucionará con el tiempo. El cambio impulsa este proceso. Para el software de computadora, el cambio ocurre cuando se corrigen los errores, cuando el software se adapta a un nuevo entorno, cuando el cliente solicita nuevas características o funciones y cuando la aplicación se somete a reingeniería para ofrecer beneficio en un contexto moderno. Durante los pasados 30 años, Manny Lehman [por ejemplo, Leh97a] y sus colaboradores realizaron análisis detallados de software de grado industrial y de sistemas con la intención de desarrollar una *teoría unificada para evolución del software*. Los detalles de este trabajo están más allá del ámbito de este libro, pero vale la pena destacar las leyes subyacentes derivadas de ella [Leh97b]:

Ley de cambio continuo (1974): El software que se implementó en un contexto de cómputo del mundo real y que, por tanto, evolucionará con el tiempo (llamados *sistemas tipo E*) debe adaptarse continuamente o de otro modo se volverá progresivamente menos satisfactorio.

Ley de complejidad creciente (1974): Conforme un sistema tipo E evoluciona, su complejidad aumenta, a menos que se haga trabajo para mantenerlo o reducirlo.

Ley de autorregulación (1974): El proceso de evolución del sistema tipo E es autorregulable con medidas de distribución de producto y de proceso cercanas a lo normal.

UNA MIRADA RÁPIDA

¿Qué es? Considere cualquier producto tecnológico que le haya funcionado bien. Lo usa con regularidad, pero está envejeciendo. Se descompone con frecuencia, se tarda más tiempo en reparar del que quisiera y ya no representa a la más reciente tecnología. ¿Qué hacer? Durante algún tiempo intenta repararlo, parcharlo, incluso extiende su funcionalidad. A esto se le llama mantenimiento. Pero éste se vuelve cada vez más difícil conforme pasan los años. Llega un momento en el que necesitará reconstruirlo. Creará un producto con funcionalidad agregada, mejor desempeño y confiabilidad, así como mantenibilidad mejorada. A eso se le llama reingeniería.

¿Quién lo hace? En el nivel de la organización, el mantenimiento lo realiza el personal de apoyo que es parte de la organización de ingeniería de software. La reingeniería la realizan especialistas en negocios (con frecuencia compañías consultoras). En el nivel de software, la reingeniería la realizan ingenieros de software.

¿Por qué es importante? Al vivir en un mundo que cambia rápidamente, las demandas sobre las funciones empresariales y la tecnología de la información que las apoyan cambian a un paso que pone enorme presión competitiva sobre toda organización comercial. Por esto, el software debe mantenerse continuamente y, en el momento adecuado, someterse a reingeniería para sostener el paso.

¿Cuáles son los pasos? El mantenimiento corrige los defectos, adapta el software para satisfacer un entorno cambiante y mejorar la funcionalidad a fin de cubrir las necesidades evolutivas de los clientes. Estratégicamente, la reingeniería de procesos de empresa (RPE) define las metas empresariales, identifica y evalúa los procesos empresariales existentes y crea procesos empresariales revisados que satisfacen mejor las metas del momento. La reingeniería de software abarca análisis de inventarios, restructuración de documentos, ingeniería inversa, reestructuración de programas y datos e ingeniería hacia adelante. La intención de dichas actividades es crear versiones de los programas existentes que muestren mayor calidad y mejor mantenibilidad.

¿Cuál es el producto final? Se producen varios productos operativos de mantenimiento y reingeniería (por ejemplo, casos de uso, modelos de análisis y diseño, procedimientos de prueba). El resultado final es actualización de software.

¿Cómo me aseguro de que lo hice bien? Con el uso de las mismas prácticas SQA que se aplican en todo proceso de ingeniería de software: revisiones técnicas para valorar los modelos de análisis y diseño; revisiones especializadas para considerar aplicabilidad y compatibilidad empresarial; y aplicación de pruebas para descubrir errores en contenido, funcionalidad e interoperabilidad.

Ley de conservación de estabilidad organizativa (1980): La tasa de actividad global efectiva promedio en un sistema tipo E en evolución no varía durante el tiempo de vida del producto.

Ley de conservación de familiaridad (1980): Conforme un sistema tipo E evoluciona, todo lo asociado con él: desarrolladores, personal de ventas, usuarios, etc., deben mantener el dominio de su contenido y comportamiento para lograr evolución satisfactoria. El crecimiento excesivo disminuye dicho dominio. Por tanto, el crecimiento incremental promedio permanece sin variación conforme el sistema evoluciona.

¿Cómo evolucionan los sistemas heredados conforme pasa el tiempo?

Ley de crecimiento continuo (1980): El contenido funcional de los sistemas tipo E debe aumentar continuamente para mantener la satisfacción del usuario durante su tiempo de vida.

Ley de declive de la calidad (1996): La calidad de los sistemas tipo E declinará, a menos que se mantengan y adapten rigurosamente a los cambios del entorno operativo.

Ley de realimentación del sistema (1996): Los procesos evolutivos tipo E constituyen sistemas de realimentación multinivel, multibucle y multiagente, y deben tratarse como tales para lograr mejora significativa sobre cualquier base razonable.

Las leyes que definieron Lehman y sus colegas son parte inherente de una realidad de la ingeniería de software. En este capítulo se estudia el reto del mantenimiento del software y las actividades de reingeniería que se requieren para extender la vida efectiva de los sistemas heredados.

29.1 Mantenimiento de software

Éste comienza casi de inmediato. El software se libera a los usuarios finales y, en cuestión de días, los reportes de errores se filtran de vuelta hacia la organización de ingeniería de software. En semanas, una clase de usuarios indica que el software debe cambiarse de modo que pueda ajustarse a las necesidades especiales de su entorno. Y en meses, otro grupo corporativo, que no quería saber nada del software cuando se liberó, ahora reconoce que puede ofrecerle beneficios inesperados. Necesitará algunas mejoras para hacer que funcione en su mundo.

El reto del mantenimiento del software comienza. Uno se enfrenta con una creciente lista de corrección de errores, peticiones de adaptación y mejoras categóricas que deben planearse, calendarizarse y, a final de cuentas, lograrse. Mucho antes, la fila creció bastante y el trabajo que implica amenaza con abrumar los recursos disponibles. Conforme pasa el tiempo, la organización descubre que emplea más dinero y tiempo en mantener los programas existentes que en someter a ingeniería nuevas aplicaciones. De hecho, no es raro que una organización de software emplee entre 60 y 70 por ciento de todos sus recursos en mantenimiento del software.

Acaso el lector pregunte por qué se requiere tanto mantenimiento y por qué se emplea tanto esfuerzo. Osborne y Chikofsky [Osb90] proporcionan una respuesta parcial:

> Mucho del software del que dependemos en la actualidad tiene en promedio una antigüedad de 10 a 15 años. Aun cuando dichos programas se crearon usando las mejores técnicas de diseño y codificación conocidas en la época [y muchas no lo fueron], se produjeron cuando el tamaño del programa y el espacio de almacenamiento eran las preocupaciones principales. Luego migraron a nuevas plataformas, se ajustaron para cambios en máquina y tecnología de sistema operativo, y aumentaron para satisfacer las necesidades de los nuevos usuarios, todo sin suficiente preocupación por la arquitectura global. El resultado es estructuras pobremente diseñadas, pobre codificación, pobre lógica y pobre documentación de los sistemas de software que ahora debemos seguir usando...

Otra razón del problema del mantenimiento del software es la movilidad del personal. Es probable que el equipo (o la persona) de software que hizo el trabajo original ya no esté más por ahí. Peor aún, otras generaciones de personal de software modificaron el sistema y se mudaron.

Y puede ser que ya no quede alguien que tenga algún conocimiento directo del sistema heredado.

Como se anotó en el capítulo 22, la naturaleza ubicua del cambio subyace a todo el trabajo del software. El cambio es inevitable cuando se construyen sistemas basados en computadoras; por tanto, deben desarrollarse mecanismos para evaluar, controlar y realizar modificaciones.

Cita:

"Mantenibilidad y comprensión de un programa son conceptos paralelos: mientras más difícil sea entender un programa, más difícil será darle mantenimiento."

Gerald Berns

A lo largo de este libro se enfatiza la importancia de entender el problema (análisis) y de desarrollar una solución bien estructurada (diseño). De hecho, la parte 2 del libro se dedica a la mecánica de tales acciones de ingeniería de software y la 3 se enfoca en las técnicas requeridas para asegurarse de que se hicieron correctamente. Análisis y diseño conducen a una importante característica del software que se llamará mantenibilidad. En esencia, la *mantenibilidad* es un indicio cualitativo[1] de la facilidad con la que el software existente puede corregirse, adaptarse o aumentarse. Gran parte de lo que trata la ingeniería de software es acerca de la construcción de sistemas que muestren alta mantenibilidad.

¿Pero qué es mantenibilidad? El software mantenible muestra modularidad efectiva (capítulo 8). Usa patrones de diseño (capítulo 12) que permiten facilidad de comprensión. Se construyó con estándares y convenciones de codificación bien definidos, que conducen a código fuente autodocumentable y comprensible. Experimentó varias técnicas de aseguramiento de calidad (parte 3 de este libro) que descubrieron potenciales problemas de mantenimiento antes de que el software se liberara. Fue creado por ingenieros de software que reconocen que acaso ya no estén presentes cuando deban realizarse cambios. En consecuencia, el diseño y la implementación del software debe "auxiliar" a la persona que realice el cambio.

29.2 Soportabilidad del software

Con la finalidad de dar soporte efectivo al software de grado industrial, su organización (o su encargado) deben poder realizar las correcciones, adaptaciones y mejoras que son parte de la actividad de mantenimiento. Pero, además, la organización debe proporcionar otras importantes actividades de soporte que incluyen soporte operativo en marcha, soporte al usuario final y actividades de reingeniería durante el ciclo de vida completo del software. Una definición razonable de *soportabilidad del software* es

> ... la capacidad de dar soporte a un sistema de software durante toda la vida del producto. Esto implica satisfacer cualquier necesidad o requisito, pero también provisión de equipo, infraestructura de soporte, software adicional, instalaciones, mano de obra o cualquier otro recurso requerido para mantener el software operativo y capaz de satisfacer su función [SSO08].

En esencia, la soportabilidad es uno de los muchos factores de calidad que deben considerarse durante las acciones de análisis y diseño que son parte del proceso de software. Deben abordarse como parte del modelo (o especificación) de requisitos y considerarse conforme el diseño evoluciona y comienza la construcción.

WebRef

En **www.software-supportability.org/Downloads.html**, puede encontrar una amplia variedad de documentos descargables acerca de soportabilidad del software.

Por ejemplo, la necesidad de software "antierrores" en el nivel de componente y código se estudió anteriormente en este libro. El software debe contener facilidades para auxiliar al personal de apoyo cuando se encuentre un defecto en el entorno operativo (y de no cometer equívocos, se *encontrarán* los defectos). Además, el personal de apoyo debe tener acceso a una base de datos que contenga registros de todos los defectos que ya se encontraron: sus características, causas y cura. Esto permitirá al personal de apoyo examinar defectos "similares" y poder brindar un medio para un diagnóstico y corrección más rápidos.

1 Existen algunas medidas cuantitativas que proporcionan un indicio indirecto de la mantenibilidad (ver, por ejemplo, [Sch99], [SEI02]).

Aunque los defectos encontrados en una aplicación son un tema de soporte crucial, la soportabilidad también demanda que se proporcionen recursos para dar soporte diario a los conflictos del usuario final. La labor del personal de soporte al usuario final es responder las consultas del usuario acerca de instalación, operación y uso de la aplicación.

29.3 Reingeniería

En un artículo fundamental escrito para *Harvard Business Review*, Michael Hammer [Ham90] sienta las bases para una revolución en el pensamiento administrativo acerca de los procesos empresariales y la computación:

> Es momento de dejar de pavimentar el camino de las vacas. En lugar de incrustar procesos caducos en silicio y software, debemos eliminarlos y comenzar de nuevo. Debemos "reingeniar" nuestras empresas: usar el poder de la moderna tecnología de la información para rediseñar radicalmente nuestros procesos empresariales con la finalidad de lograr mejoras dramáticas en su rendimiento.
>
> Toda compañía opera de acuerdo con un gran número de reglas desarticuladas [...] La reingeniería lucha por liberarse de las antiguas reglas acerca de cómo organizarnos y dirigir nuestros negocios.

Cita:

"Enfrentar el mañana con el pensamiento puesto en usar los métodos de ayer es vislumbrar la vida paralizado."

James Bell

Como toda revolución, el llamado a las armas de Hammer resultó en cambios tanto positivos como negativos. Durante los años de 1990, algunas compañías hicieron un esfuerzo legítimo por someterse a reingeniería y los resultados condujeron a mejora competitiva. Otros se apoyaron exclusivamente en reducción y subcontratación (en lugar de en reingeniería) para mejorar su línea de referencia. Con frecuencia resultaron organizaciones "medias" con poco potencial para crecimiento futuro [DeM95a].

Hacia finales de la primera década del siglo XXI, la publicidad relacionada con la reingeniería disminuyó, pero el proceso en sí continúa en las compañías grandes y pequeñas. El nexo entre reingeniería empresarial e ingeniería de software yace en una "visión de sistema".

Con frecuencia, el software es la realización de las reglas empresariales que Hammer analiza. En la actualidad, las principales compañías tienen decenas de miles de programas de cómputo que soportan las "antiguas reglas empresariales". Conforme los administradores trabajan para modificar las reglas a fin de lograr mayor efectividad y competitividad, el software debe seguirles el paso. En algunos casos, esto significa la creación de grandes y novedosos sistemas basados en cómputo.[2] Pero en muchos otros, significa la modificación o reconstrucción de las aplicaciones existentes.

En las secciones que siguen, se examina la reingeniería en una forma descendente, comenzando con un breve panorama de la reingeniería de los procesos de empresas y avanzando hacia un análisis más detallado de las actividades técnicas que ocurren cuando el software se somete a reingeniería.

29.4 Reingeniería de procesos de empresa

PUNTO CLAVE

La RPE con frecuencia da como resultado nueva funcionalidad de software, mientras que la reingeniería de software trabaja para sustituir la funcionalidad de software existente con software mejor y más mantenible.

La *reingeniería de procesos de empresa* (RPE) se extiende más allá del ámbito de las tecnologías de la información y de la ingeniería de software. Entre las muchas definiciones (la mayoría un tanto abstractas) que se han sugerido para la RPE, está una publicada en *Fortune Magazine* [Ste93]: "la búsqueda, e implementación, de cambios radicales en los procesos de las empresas para lograr resultados innovadores". ¿Pero cómo se realiza la búsqueda y cómo se logra la implementación? Más importante, ¿cómo puede asegurarse que el "cambio radical" sugerido conducirá realmente a "resultados innovadores" en lugar de a caos organizacional?

2 La explosión de las aplicaciones y sistemas basados en web es indicativo de esta tendencia.

29.4.1 Procesos empresariales

Un proceso empresarial es "un conjunto de tareas lógicamente relacionadas, que se realizan para lograr un resultado empresarial definido" [Dav90]. Dentro del proceso empresarial, personal, equipo, recursos materiales y procedimientos empresariales se combinan para producir un resultado específico. Los ejemplos de procesos empresariales incluyen diseñar un nuevo producto, comprar servicios y suministros, contratar un nuevo empleado y pagar a proveedores. Cada uno demanda un conjunto de tareas y usa diversos recursos dentro de la empresa.

Como ingeniero de software, su trabajo se realiza en el fondo de esta jerarquía. Sin embargo, asegúrese de que alguien considera seriamente los niveles superiores. Si esto no se ha hecho, su trabajo está en riesgo.

Todo proceso empresarial tiene un cliente definido: una persona o grupo que recibe el resultado (por ejemplo, una idea, un reporte, un diseño, un servicio, un producto). Además, los procesos empresariales atraviesan las fronteras de la organización. Requieren que diferentes grupos organizativos participen en las "tareas lógicamente relacionadas" que definen el proceso.

Todo sistema es en realidad una jerarquía de subsistemas. Una empresa no es la excepción. Toda la empresa está segmentada en la forma siguiente:

Empresa → sistemas empresariales → procesos empresariales → subprocesos empresariales

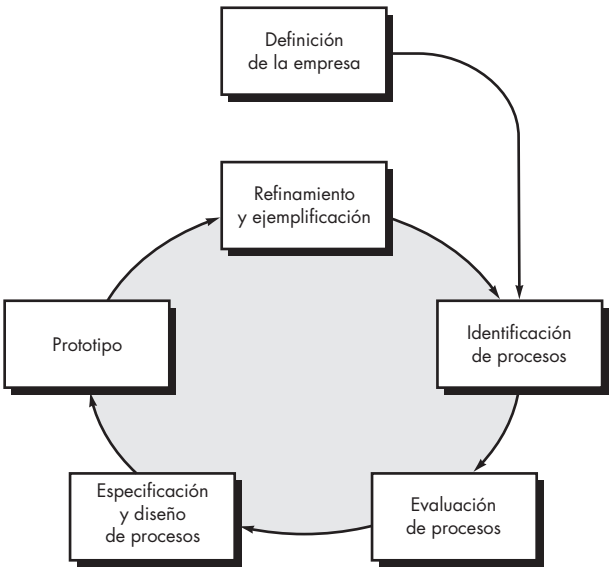
Cada sistema empresarial (también llamado *función empresarial*) está compuesto de uno o más procesos empresariales, y cada proceso empresarial se define mediante un conjunto de subprocesos.

La RPE puede aplicarse en cualquier nivel de la jerarquía, pero conforme se ensancha su ámbito (es decir, conforme se avanza hacia arriba en la jerarquía), los riesgos asociados con la RPE crecen de manera dramática. Por esta razón, la mayoría de los esfuerzos RPE se enfocan en procesos o subprocesos individuales.

29.4.2 Un modelo RPE

Como la mayoría de las actividades de ingeniería, la reingeniería de procesos de empresa es iterativa. Las metas de la empresa y los procesos que los logran deben adaptarse a un entorno empresarial cambiante. Por esta razón, no hay inicio ni fin de la RPE: es un proceso evolutivo. En la figura 29.1 se muestra un modelo para reingeniería de proceso de empresa. El modelo define seis actividades:

FIGURA 29.1
Un modelo RPE



Definición de la empresa. Las metas de la empresa se identifican dentro del contexto de cuatro motores clave: reducción de costo, reducción de tiempo, mejora de la calidad y desarrollo y fortalecimiento del personal. Las metas pueden definirse para toda la empresa o para un componente específico de ella.

Identificación de procesos. Se identifican los procesos que son cruciales para lograr las metas definidas en la definición de la empresa. Luego pueden clasificarse por importancia, por necesidad para cambiar o en cualquier otra forma que sea adecuada para la actividad de reingeniería.

Evaluación de procesos. Los procesos existentes se analizan y miden ampliamente. Se identifican las tareas de proceso, se anotan los costos y el tiempo consumido por ellas y se aíslan los problemas de calidad/desempeño.

Especificación y diseño de procesos. Con base en la información obtenida durante las primeras tres actividades RPE, se preparan casos de uso (capítulos 5 y 6) para cada proceso que deba someterse a reingeniería. Dentro del contexto de la RPE, los casos de uso identifican un escenario que entrega algún resultado a un cliente. Con el caso de uso como la especificación del proceso, se diseña un nuevo conjunto de tareas para el proceso.

Cita:

"Tan pronto como algo viejo se nos muestra en forma novedosa, nos pacificamos."

F. W. Nietzsche

Prototipo. Un proceso empresarial rediseñado debe convertirse en prototipo antes de que se integre plenamente en la empresa. Esta actividad "pone a prueba" el proceso, de modo que puedan hacerse refinamientos.

Refinamiento y ejemplificación. Con base en la realimentación del prototipo, el proceso empresarial se refina y luego se ejemplifica dentro de un sistema empresarial.

En ocasiones, dichas actividades RPE se usan en conjunto con herramientas de análisis de flujo de trabajo. La intención de dichas herramientas es construir un modelo del flujo de trabajo existente con la intención de analizar mejor los procesos actuales.

HERRAMIENTAS DE SOFTWARE

Reingeniería de procesos de empresa (RPE)

Objetivo: El objetivo de las herramientas RPE es dar soporte al análisis y valoración de los procesos empresariales existentes, y a la especificación y diseño de nuevos.

Mecánica: La mecánica de las herramientas varía. En general, las herramientas RPE permiten a un analista empresarial modelar los procesos empresariales existentes con la intención de valorar las ineficiencias del flujo de trabajo o problemas funcionales. Una vez identificados los problemas existentes, las herramientas permiten el análisis para crear prototipos y/o simular procesos empresariales revisados.

Herramientas representativas:[3]

Extend, desarrollada por ImagineThat, Inc. (**www.imaginethatinc.com**), es una herramienta de simulación para modelar procesos existentes y explorar nuevos. Extend proporciona abundante capacidad "y si…", que permite a un analista empresarial explorar diferentes escenarios de proceso.

e-Work, desarrollada por Metastorm (**www.metastorm.com**), proporciona soporte de gestión de procesos empresariales para procesos tanto manuales como automatizados.

IceTools, desarrollada por Blue Ice (**www.blueice.com**), es una colección de plantillas RPE para Microsoft Office y Microsoft Project.

SpeeDev, desarrollada por SpeeDev Inc. (**www.speedev.com**), es una de muchas herramientas que permiten a una organización modelar flujo de trabajo de procesos (en este caso, flujo de trabajo IT).

Workflow tool suite, desarrollada por MetaSoftware (**www.metasoftware.com**), incorpora una suite de herramientas para modelado, simulación y calendarización de flujo de trabajo.

Una útil lista de ligas a herramientas RPE puede encontrarse en **www.opfro.org/index.html?Components/Producers/Tools/BusinessProcessReengineeringTools.html~Contents**

3 Las herramientas que se mencionan aquí no representan un respaldo, sino que son una muestra de las herramientas que hay en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

29.5 Reingeniería de software

El escenario es demasiado común: una aplicación que atendió las necesidades empresariales de una compañía durante 10 o 15 años. Durante ese tiempo se corrigió, adaptó y mejoró muchas veces. Las personas realizaban esta tarea con las mejores intenciones, pero las buenas prácticas de ingeniería siempre se hicieron a un lado (debido a la presión de otros asuntos). Ahora la aplicación es inestable. Todavía funciona, pero cada vez que se intenta un cambio, ocurren inesperados y serios efectos colaterales. Aun así, la aplicación debe seguir evolucionando. ¿Qué hacer?

El software sin mantenimiento no es un problema nuevo. De hecho, el énfasis ampliado acerca de la reingeniería de software se produjo por los problemas de mantenimiento de software que se acumularon durante más de cuatro décadas.

29.5.1 Un modelo de proceso de reingeniería de software

La reingeniería toma tiempo, cuesta cantidades significativas de dinero y absorbe recursos que de otro modo pueden ocuparse en preocupaciones inmediatas. Por todas estas razones, la reingeniería no se logra en pocos meses o incluso en algunos años. La reingeniería de los sistemas de información es una actividad que absorberá recursos de tecnología de la información durante muchos años. Por esto, toda organización necesita una estrategia pragmática para la reingeniería de software.

WebRef

Una excelente fuente de información acerca de la reingeniería de software puede encontrarse en **reengineer.org**

Una estrategia factible se contempla en un modelo de proceso de reingeniería. Más tarde, en esta sección, se estudiará el modelo, pero primero se presentan algunos principios básicos.

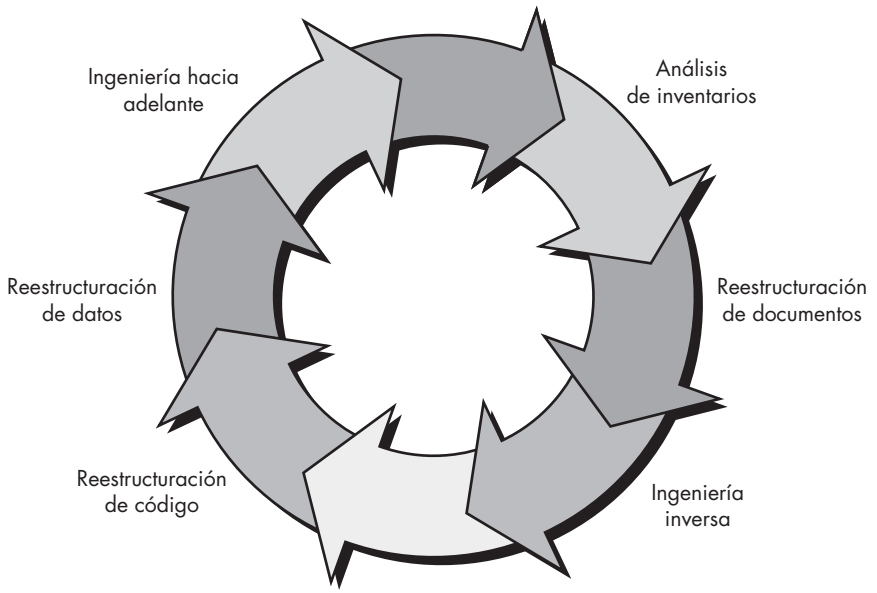
La reingeniería es una actividad de reconstrucción. Para entenderla mejor, piense en una actividad análoga: la reconstrucción de una casa. Considere la siguiente situación. Usted compra una casa en otro estado. En realidad nunca ha visto la propiedad, pero la adquirió a un precio sorprendentemente bajo, con la advertencia de que es posible que deba reconstruirla por completo. ¿Cómo procedería?

- Antes de comenzar a reconstruir, parecería razonable inspeccionar la casa. Para determinar si necesita reconstruirse, usted (o un inspector profesional) crea una lista de criterios, de modo que su inspección sea sistemática.
- Antes de demoler y reconstruir toda la casa, asegúrese de que la estructura es débil. Si la casa es estructuralmente sólida, acaso sea posible "remodelar" sin reconstruir (a un costo mucho más bajo y en mucho menos tiempo).
- Antes de comenzar a reconstruir, asegúrese de entender cómo se construyó la original. Eche un vistazo detrás de las paredes. Entienda cómo están el alambrado, la plomería y la estructura interna. Incluso si tira todo a la basura, la comprensión que obtenga le servirá cuando comience la construcción.
- Si comienza a reconstruir, use solamente los materiales más modernos y más duraderos. Esto puede costar un poco más ahora, pero le ayudará a evitar costos y tardados mantenimientos posteriores.
- Si decide reconstruir, sea disciplinado en ello. Use prácticas que resultarán en alta calidad, hoy y en el futuro.

Aunque estos principios se enfocan en la reconstrucción de una casa se aplican igualmente bien a la reingeniería de los sistemas y aplicaciones basados en cómputo.

Para implementar estos principios puede usar un modelo de proceso de reingeniería de software que defina seis actividades, como se muestra en la figura 29.2. En algunos casos, dichas actividades ocurren en secuencia lineal, aunque no siempre. Por ejemplo, es posible que se tenga que recurrir a la ingeniería inversa (comprender el funcionamiento interno de un programa) antes de que pueda comenzar la reestructuración de documentos.

FIGURA 29.2
Modelo de proceso de reingeniería de software



29.5.2 Actividades de reingeniería de software

El paradigma de reingeniería que se muestra en la figura 29.2 es un modelo cíclico. Esto significa que cada una de las actividades presentadas como parte del paradigma puede revisarse. Para algún ciclo particular, el proceso puede terminar después de cualquiera de estas actividades.

CONSEJO

Si el tiempo y los recursos son escasos, puede considerar la aplicación del principio de Pareto al software que se someterá a reingeniería. Aplique el proceso de reingeniería al 20 por ciento del software que represente el 80 por ciento de los problemas.

Análisis de inventarios. Toda organización de software debe tener un inventario de todas las aplicaciones. El inventario puede ser nada más que un modelo de hojas de cálculo que contenga información que ofrezca una descripción detallada (por ejemplo, tamaño, edad, importancia empresarial) de cada aplicación activa. Al ordenar esta información de acuerdo con importancia empresarial, longevidad, mantenibilidad actual, soportabilidad y otros importantes criterios locales, aparecen los candidatos para reingeniería. Entonces pueden asignarse recursos a esas aplicaciones.

Es importante observar que el inventario debe revisarse con regularidad. El estado de las aplicaciones (por ejemplo, importancia empresarial) puede cambiar con el tiempo y, como resultado, cambiarán las prioridades para aplicar la reingeniería.

CONSEJO

Cree sólo tanta documentación como necesite para entender el software, ni una página más.

Reestructuración de documentos. La documentación débil es el distintivo de muchos sistemas heredados. Pero, ¿qué puede hacer con ella? ¿Cuáles son sus opciones?

1. *La creación de documentación consume demasiado tiempo*. Si el sistema funciona puede elegir vivir con lo que tiene. En algunos casos, éste es el enfoque correcto. No es posible volver a crear documentación para cientos de programas de cómputo. Si un programa es relativamente estático, se aproxima al final de su vida útil y es improbable que experimente cambio significativo, ¡déjelo así!
2. *La documentación debe actualizarse, pero su organización tiene recursos limitados.* Use un enfoque "documente cuando toque". Acaso no sea necesario volver a documentar por completo una aplicación. En vez de ello, aquellas porciones del sistema que en el momento experimenten cambio se documentan por completo. Con el tiempo, evolucionará una colección de documentación útil y relevante.
3. *El sistema tiene importancia empresarial y debe volver a documentarse por completo.* Incluso en este caso, un enfoque inteligente es recortar la documentación a un mínimo esencial.

Cada una de estas opciones es viable. Su organización de software debe elegir aquella que sea más adecuada para cada caso.

Ingeniería inversa. El término *ingeniería inversa* tiene su origen en el mundo del hardware. Una compañía desensambla un producto de hardware de otra empresa con la intención de entender los "secretos" de diseño y fabricación de su competidor. Dichos secretos podrían entenderse fácilmente si se obtuvieran las especificaciones de diseño y fabricación. Pero esos documentos son propiedad de la empresa competidora y no están disponibles para la compañía que hace la ingeniería inversa. En esencia, la ingeniería inversa exitosa deriva en una o más especificaciones de diseño y fabricación para un producto al examinar especímenes reales del mismo.

WebRef

En **www.comp.lancs.ac.uk/projects/RenaissanceWeb/**, puede encontrar varios recursos para la comunidad de reingeniería.

La ingeniería inversa para el software es muy similar. No obstante, en la mayoría de los casos, el programa que se va a someter a ingeniería inversa no es de un competidor: es el propio trabajo de la compañía (con frecuencia, elaborado muchos años atrás). Los "secretos" por entender son oscuros porque jamás se desarrollaron especificaciones. Por tanto, la ingeniería inversa para software es el proceso de analizar un programa con la intención de crear una representación del mismo en un nivel superior de abstracción que el código fuente. La ingeniería inversa es un proceso de *recuperación de diseño*. Las herramientas de ingeniería inversa extraen información de diseño de datos, arquitectónico y procedimental de un programa existente.

Reestructuración de código. El tipo más común de reingeniería (en realidad, en este caso es cuestionable el uso del término reingeniería) es la *reestructuración de código*.[4] Algunos sistemas heredados tienen una arquitectura de programa relativamente sólida, pero los módulos individuales fueron codificados en una forma que los hace difíciles de entender, poner a prueba y mantener. En tales casos, el código dentro de los módulos sospechosos puede reestructurarse.

Para realizar esta actividad se analiza el código fuente con una herramienta de reestructuración. Las violaciones a los constructos de programación estructurada se anotan y luego el código se reestructura (esto puede hacerse automáticamente) o incluso se reescribe en un lenguaje de programación más moderno. El código reestructurado resultante se revisa y pone a prueba para garantizar que no se introdujeron anomalías. La documentación de código interna se actualiza.

Reestructuración de datos. Un programa con arquitectura de datos débil será difícil de adaptar y mejorar. De hecho, para muchas aplicaciones, la arquitectura de información tiene más que ver con la viabilidad a largo plazo de un programa que con el código fuente en sí.

A diferencia de la reestructuración de código, que ocurre en un nivel de abstracción relativamente bajo, la reestructuración de datos es una actividad de reingeniería a gran escala. En la mayoría de los casos, la reestructuración de los datos comienza con una actividad de ingeniería inversa. La arquitectura de datos existente se diseca y se definen modelos de datos necesarios (capítulos 6 y 9). Se identifican los objetos y atributos de datos, y se revisa la calidad de las estructuras de datos existentes.

Cuando la estructura de datos es débil (por ejemplo, si se implementan archivos planos, cuando un enfoque relacional simplificaría enormemente el procesamiento), los datos se someten a reingeniería.

Puesto que la arquitectura de datos tiene una fuerte influencia sobre la arquitectura del programa y sobre los algoritmos que los pueblan, los cambios a los datos invariablemente resultarán en cambios arquitectónicos o en el nivel de código.

4 La reestructuración de código tiene algunos de los elementos de la "refactorización", un concepto de rediseño introducido en el capítulo 8 y estudiado en otras partes de este libro.

Ingeniería hacia adelante. En un mundo ideal, las aplicaciones se reconstruirían usando un "motor de reingeniería" automático. El programa antiguo se alimentaría en el motor, se analizaría, se reestructuraría y luego se regeneraría de manera que mostrara los mejores aspectos de la calidad del software. A corto plazo es improbable que tal "motor" aparezca, pero los proveedores introdujeron herramientas que proporcionan un subconjunto limitado de dichas capacidades y que abordan dominios de aplicación específicos (por ejemplo, aplicaciones que se implementan usando un sistema de base de datos específico). Más importante, dichas herramientas de reingeniería se vuelven cada vez más sofisticadas.

La ingeniería hacia adelante no sólo recupera información de diseño del software existente, sino que también usa esta información para alterar o reconstituir el sistema existente con la intención de mejorar su calidad global. En la mayoría de los casos, el software sometido a reingeniería vuelve a implementar la función del sistema existente y también añade nuevas funciones y/o mejora el rendimiento global.

29.6 Ingeniería inversa

La ingeniería inversa conjura una imagen de la "rendija mágica". Usted alimenta en la rendija un archivo fuente sin documentar, diseñado de manera fortuita, y del otro lado sale una descripción y documentación completas del diseño para el programa de cómputo. Por desgracia, la rendija mágica no existe. La ingeniería inversa puede extraer información de diseño a partir del código fuente, pero el nivel de abstracción, la completitud de la documentación, el grado en el que las herramientas y un analista humano trabajan en conjunto, y la direccionalidad del proceso son enormemente variables.

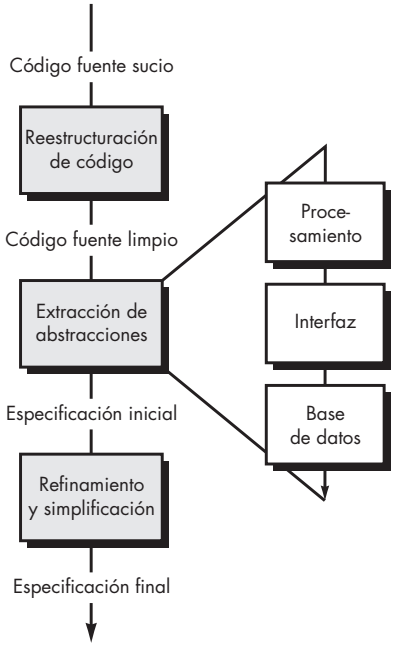
El *nivel de abstracción* de un proceso de ingeniería inversa y las herramientas usadas para efectuarla tienen que ver con la sofisticación de la información de diseño que puede extraerse del código fuente. De manera ideal, el nivel de abstracción debe ser tan alto como sea posible, es decir, el proceso de ingeniería inversa debe ser capaz de inferir representaciones de diseño procedimental (una abstracción de bajo nivel), información de estructura de programa y datos (un nivel de abstracción un poco más alto), modelos de objeto, modelos de datos y/o flujo de control (un nivel de abstracción relativamente alto) y modelos de relación de entidad (un nivel de abstracción alto). Conforme aumenta el nivel de abstracción se proporciona información que permitirá facilitar la comprensión del programa.

La *completitud* de un proceso de ingeniería inversa se refiere al nivel de detalle que se proporciona en un nivel de abstracción. En la mayoría de los casos, la completitud disminuye conforme aumenta el nivel de abstracción. Por ejemplo, dada una lista de código fuente, es relativamente sencillo desarrollar una representación de diseño procedimental completa. También pueden inferirse representaciones de diseño arquitectónico simples, pero es mucho más difícil desarrollar un conjunto completo de diagramas o modelos UML.

La completitud mejora en proporción directa a la cantidad de análisis realizado por la persona que efectúa la ingeniería inversa. La *interactividad* tiene que ver con el grado en el que el ser humano se "integra" con las herramientas automatizadas para crear un proceso de ingeniería inversa efectivo. En la mayoría de los casos, conforme aumenta el nivel de abstracción, la interactividad debe aumentar o decaerá la completitud.

Si la *direccionalidad* del proceso de ingeniería inversa es de una vía, toda la información extraída del código fuente se proporciona al ingeniero de software que luego puede usarla, durante cualquier actividad de mantenimiento. Si la direccionalidad es de dos vías, la información se alimenta a una herramienta de reingeniería que intenta reestructurar o regenerar el programa antiguo.

FIGURA 29.3
El proceso de ingeniería inversa



WebRef
Útiles recursos para "recuperación de diseño y comprensión de programa" pueden encontrarse en **wwwsel.iit.nrc.ca/projects/dr/dr.html**

El proceso de ingeniería inversa se representa en la figura 29.3. Antes de poder comenzar las actividades de ingeniería inversa, el código fuente no estructurado ("sucio") se reestructura (sección 29.5.1) de modo que sólo contenga los constructos de programación estructurados.[5] Esto hace que el código fuente sea más fácil de leer y que proporcione la base para todas las actividades de ingeniería inversa posteriores.

El núcleo de la ingeniería inversa radica en una actividad llamada *extracción de abstracciones*. Debe evaluar el programa antiguo y, a partir del código fuente (con frecuencia no documentado), desarrollar una especificación significativa del procesamiento que se realiza, de la interfaz de usuario que se aplica y de las estructuras de datos del programa o de la base de datos que se usa.

29.6.1 Ingeniería inversa para comprender datos

En algunos casos, la primera actividad de reingeniería intenta construir un diagrama de clase UML.

El enfoque de la ingeniería inversa para datos para software convencional sigue una ruta análoga: 1) construir un modelo de datos, 2) identificar atributos de objetos de datos y 3) definir relaciones.

La ingeniería inversa de datos ocurre en diferentes niveles de abstracción y con frecuencia es la primera tarea de reingeniería. En el nivel del programa, las estructuras de datos internas del programa con frecuencia deben someterse a ingeniería inversa como parte de un esfuerzo de reingeniería global. En el nivel del sistema, las estructuras de datos globales (por ejemplo, archivos, bases de datos) con frecuencia se someten a reingeniería para acomodar nuevos paradigmas de administración de base de datos (por ejemplo, moverse de un archivo plano a sistemas de bases de datos relacionales u orientadas a objetos). La ingeniería inversa de las estructuras de datos globales actuales monta el escenario para la introducción de una nueva base de datos en todo el sistema.

Estructuras de datos internas. Las técnicas de ingeniería inversa para datos internos del programa se enfocan en la definición de clases de objetos. Esto se logra al examinar el código del programa con la intención de agrupar variables del programa relacionadas. En muchos ca-

5 El código puede reestructurarse usando un *motor de reestructuración*, una herramienta que reestructura código fuente.

sos, la organización de datos dentro del código identifica tipos de datos abstractos. Por ejemplo, el registro de estructuras, archivos, listas y otras estructuras de datos con frecuencia proporciona un indicador inicial de clases.

Estructura de la base de datos. Sin importar su organización lógica y su estructura física, una base de datos permite la definición de objetos de datos y soporta algún método para establecer relaciones entre los objetos. Por tanto, la reingeniería de un esquema de base de datos en otro nuevo requiere comprender los objetos existentes y sus relaciones.

Puede usar los siguientes pasos [Pre94] para definir el modelo de extracción de datos como precursor de la reingeniería de un nuevo modelo de base de datos: 1) construir un modelo de objeto inicial, 2) determinar claves candidatas (los atributos se examinan para determinar si se usan para apuntar hacia otro registro o tabla; los que funcionan como punteros se convierten en clases candidatas), 3) refinar las clases tentativas, 4) definir generalizaciones y 5) descubrir asociaciones usando técnicas que sean análogas al enfoque CRC. Una vez que se conoce la información definida en los pasos anteriores, puede aplicarse una serie de transformaciones [Pre94] para mapear la antigua estructura de la base de datos en una nueva estructura.

29.6.2 Ingeniería inversa para entender el procesamiento

Cita:

"Existe una pasión por la comprensión, así como existe una pasión por la música. Dicha pasión es más bien común en los niños, pero se pierde en la mayoría de la gente tiempo después."

Albert Einstein

La ingeniería inversa para entender el procesamiento comienza con un intento por comprender y luego extraer abstracciones procedimentales representadas mediante el código fuente. Para comprender las abstracciones procedimentales se analiza el código en varios niveles de abstracción: sistema, programa, componente, patrón y enunciado.

La funcionalidad global de todo el sistema de aplicación debe entenderse antes de que ocurra trabajo de ingeniería inversa más detallado. Esto establece el contexto para un mayor análisis y proporciona comprensión acerca de los conflictos de interoperabilidad entre aplicaciones dentro del sistema. Cada uno de los programas que constituyen el sistema de aplicación representa una abstracción funcional en un nivel alto de detalle. Se crea un diagrama de bloques, que representa la interacción entre dichas abstracciones funcionales. Cada componente realiza alguna subfunción y representa una abstracción procedimental definida. Se desarrolla una narrativa de procesamiento para cada componente. En algunas situaciones, ya existen especificaciones de sistema, programa y componente. Cuando éste es el caso, las especificaciones se revisan para conformarse con el código existente.[6]

Las cosas se vuelven más complejas cuando se considera el código dentro de un componente. Debe buscar secciones de código que representen patrones procedimentales genéricos. En casi todo componente, una sección de código prepara los datos para procesamiento (dentro del módulo), una sección diferente del código realiza el procesamiento y otra prepara los resultados del procesamiento para exportarlos desde el componente. Dentro de cada una de esas secciones, pueden encontrarse patrones más pequeños; por ejemplo, validación de datos y comprobación de enlaces que con frecuencia ocurren dentro de la sección de código que prepara los datos para procesamiento.

Para sistemas grandes, la ingeniería inversa por lo general se logra usando un enfoque semiautomatizado. Es posible usar herramientas automatizadas para auxiliarse en la comprensión de la semántica del código existente. La salida de este proceso pasa entonces a reestructuración de herramientas de ingeniería hacia adelante a fin de completar el proceso de reingeniería.

6 Con frecuencia, las especificaciones escritas anteriormente, en la historia de vida del programa, nunca se actualizan. Conforme se realizan cambios, el código ya no es congruente con la especificación.

29.6.3 Ingeniería inversa de interfaces de usuario

Las GUI (interfaces de usuario gráficas) sofisticadas se han vuelto obligatorias para productos y sistemas basados en computadora de todo tipo. Por tanto, el redesarrollo de las interfaces de usuario se ha convertido en uno de los tipos más comunes de actividad de reingeniería. Pero antes de poder reconstruir una interfaz de usuario, debe realizarse ingeniería inversa.

Para comprender completamente una interfaz de usuario existente, deben especificarse la estructura y el comportamiento de la interfaz. Merlo *et al.* [Mer93] sugieren tres preguntas básicas que deben responderse conforme comienza la ingeniería inversa de la UI.

? ¿Cómo comprender el funcionamiento de una interfaz de usuario existente?

- ¿Cuáles son las acciones básicas (por ejemplo, golpes de tecla y clics de ratón) que debe procesar la interfaz?
- ¿Cuál es la descripción compacta de la respuesta de comportamiento del sistema a dichas acciones?
- ¿Qué se entiende por "reemplazo" o, más precisamente, qué concepto de equivalencia de interfaces es relevante aquí?

La notación de modelado de comportamiento (capítulo 7) puede proporcionar un medio para desarrollar respuestas a las primeras dos preguntas. Mucha de la información necesaria para crear un modelo de comportamiento puede obtenerse al observar la manifestación externa de la interfaz existente. Pero información adicional necesaria para crear el modelo de comportamiento debe extraerse del código.

Es importante observar que un reemplazo de GUI puede no reflejar con exactitud la antigua interfaz (de hecho, puede ser radicalmente diferente). Con frecuencia, vale la pena desarrollar una nueva metáfora de interacción. Por ejemplo, una antigua UI solicita que un usuario proporcione un factor de escala (que va de 1 a 10) para encoger o ampliar una imagen gráfica. Una GUI sometida a reingeniería puede usar una barra de desplazamiento y ratón para lograr la misma función.

HERRAMIENTAS DE SOFTWARE

Ingeniería inversa

Objetivo: Auxiliar a los ingenieros de software a comprender la estructura interna de diseño de programas complejos.

Mecánica: En la mayoría de los casos, las herramientas de ingeniería inversa aceptan código fuente como entrada y producen varias representaciones de diseño estructural, procedimental, de datos y de comportamiento.

Herramientas representativas:[7]

Imagix 4D, desarrollada por Imagix (**www.imagix.com**), "ayuda a los desarrolladores de software a comprender software C y C++ complejo o heredado" mediante ingeniería inversa y documentación de código fuente.

Understand, desarrollado por Scientific Toolworks, Inc. (**www.scitools.com**), analiza gramaticalmente Ada, Fortran, C, C++ y Java "para ingeniería inversa, documenta automáticamente, calcula métricas de código y le ayuda a comprender, navegar y mantener código fuente".

Una lista exhaustiva de herramientas de ingeniería puede encontrarse en **http://scgwiki.iam.unibe.ch:8080/SCG/370**

7 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas que hay en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

29.7 Reestructuración

La *reestructuración de software* modifica el código fuente y/o los datos con la intención de hacerlos sensibles a cambios futuros. En general, la reestructuración no modifica la arquitectura global del programa. Tiende a enfocarse sobre detalles de diseño de módulos individuales y sobre estructuras de datos locales definidas dentro de módulos. Si el esfuerzo de reestructuración se extiende más allá de las fronteras del módulo y abarca la arquitectura del software, la reestructuración se convierte en ingeniería hacia adelante (sección 29.7).

Aunque la reestructuración de código puede aliviar los problemas inmediatos asociados con la depuración o con pequeños cambios, no es reingeniería. El beneficio real se logra sólo cuando se reestructuran los datos y la arquitectura.

La reestructuración ocurre cuando la arquitectura básica de una aplicación es sólida, aun cuando el interior técnico necesite trabajarse. Se inicia cuando grandes partes del software son aprovechables y sólo un subconjunto de todos los módulos y datos necesitan modificación extensa.[8]

29.7.1 Reestructuración de código

La *reestructuración de código* se realiza para producir un diseño que produzca la misma función pero con mayor calidad que el programa original. En general, las técnicas de reestructuración de código (por ejemplo, las técnicas de simplificación lógica de Warnier [War74]) modelan la lógica del programa usando álgebra booleana y luego aplican una serie de reglas de transformación que producen lógica reestructurada. El objetivo es tomar una "ensalada" de código y derivar un diseño procedimental que se conforme con la filosofía de programación estructurada (capítulo 10).

También se han propuesto otras técnicas de reestructuración para su uso con herramientas de reingeniería. Un diagrama de intercambio de recursos mapea cada módulo de programa y los recursos (tipos de datos, procedimientos y variables) que se intercambiarán entre él y otros módulos. Al crear representaciones de flujo de recursos, la arquitectura del programa puede reestructurarse para lograr un mínimo acoplamiento entre módulos.

29.7.2 Reestructuración de datos

Antes de que pueda comenzar la reestructuración de datos debe realizarse una actividad de ingeniería inversa llamada *análisis de código fuente*. Se evalúan todos los enunciados de lenguaje de programación que contienen definiciones de datos, descripciones de archivo I/O y descripciones de interfaz. La intención es extraer ítems de datos y objetos, obtener información acerca del flujo de datos y entender las estructuras de datos existentes que se implementaron. Esta actividad en ocasiones se llama *análisis de datos*.

Una vez completado el análisis de datos, comienza el *rediseño de datos*. En su forma más simple, un paso de *estandarización de registro de datos* clarifica las definiciones de los datos para lograr consistencia entre nombres de ítem de datos o formatos de registro físico dentro de una estructura de datos existente o dentro de un formato de archivo. Otra forma de rediseño, llamada *racionalización de nombre de datos*, garantiza que todas las convenciones de nomenclatura de datos se establezcan de acuerdo con estándares locales y que los sobrenombres se eliminen conforme los datos fluyen a través del sistema.

Cuando la reestructuración avanza más allá de la estandarización y la racionalización, se realizan modificaciones físicas a las estructuras de datos existentes para hacer que el diseño de datos sea más efectivo. Esto puede significar una traducción de un formato de archivo a otro o, en algunos casos, traducción de un tipo de base de datos a otra.

8 En ocasiones es difícil hacer una distinción entre reestructuración extensa y redesarrollo. Ambos son reingeniería.

HERRAMIENTAS DE SOFTWARE

Reestructuración de software

Objetivo: El objetivo de las herramientas de reestructuración es transformar el software de computadora no estructurado más antiguo en lenguajes de programación y estructuras de diseño modernos.

Mecánica: En general, se ingresa el código fuente y se transforma en un programa mejor estructurado. En algunos casos, la transformación ocurre dentro del mismo lenguaje de programación. En otros, un lenguaje de programación más antiguo se transforma en un lenguaje más moderno.

Herramientas representativas:[9]

DMS Software Reengineering Toolkit, desarrollada por Semantic Design (**www.semdesigns.com**), proporciona varias capacidades de reestructuración para COBOL, C/C++, Java, Fortran 90 y VHDL.

Clone Doctor, desarrollada por Semantic Designs, Inc. (**www.semdesigns.com**), analiza y transforma programas escritos en C, C++, Java o COBOL o en cualquier otro lenguaje de computadora basado en texto.

plusFORT, desarrollada por Polyheddron (**www.polyhedron.com**), es una suite de herramientas FORTRAN que contiene capacidades para reestructurar programas FORTRAN pobremente diseñados en el estándar moderno FORTRAN o C.

Indicadores hacia varias herramientas de reingeniería e ingeniería inversa pueden encontrarse en **www.csse.monash.edu/~ipeake/reeng/free-swre-tools.html** y en **www.cs.ualberta.ca/~kenw/toolsdir/all.html**

29.8 INGENIERÍA HACIA ADELANTE

¿Qué opciones existen cuando uno se enfrenta con un programa diseñado e implementado pobremente?

Un programa con flujo de control que sea el equivalente gráfico de una olla de espagueti, con "módulos" que tienen 2 000 enunciados de largo, con pocas líneas de comentarios significativos en 290 000 enunciados fuente y sin otra documentación, debe modificarse para alojar los cambiantes requisitos de usuario. Se tienen las siguientes opciones:

1. Para implementar los cambios necesarios puede luchar a través de modificación tras modificación, combatir al diseño *ad hoc* y el código fuente enredado.
2. Puede intentar comprender los funcionamientos interiores más amplios del programa con la intención de hacer modificaciones de manera más efectiva.
3. Puede rediseñar, recodificar y poner a prueba aquellas porciones del software que requieran modificación y aplicar un enfoque de ingeniería de software a todos los segmentos revisados.
4. Puede rediseñar, recodificar y poner a prueba completamente el programa, y usar herramientas de reingeniería para ayudar a comprender el diseño actual.

La reingeniería es un poco como limpiar sus dientes. Puede pensar en miles de razones para demorarla, y la evitará con desidia durante un rato. Pero finalmente, sus tácticas de demora regresarán para causarle dolor.

No hay una sola opción "correcta". Las circunstancias pueden dictar la primera opción incluso si las otras son más deseables.

En lugar de esperar hasta recibir una solicitud de mantenimiento, la organización de desarrollo o soporte usa los resultados de un análisis de inventario para seleccionar un programa: 1) que permanecerá en uso durante un número preseleccionado de años, 2) que en el momento se use con éxito y 3) que tenga probabilidad de experimentar grandes modificaciones o aumentos en el futuro cercano. Entonces se aplican las opciones 2, 3 o 4.

A primera vista, la sugerencia de que redesarrolle un programa grande cuando ya existe una versión operativa puede parecer muy extravagante. Antes de juzgar, considere los siguientes puntos:

9 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas que hay en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

1. El costo de mantener una línea de código fuente puede ser 20 a 40 veces el costo del desarrollo inicial de dicha línea.
2. El rediseño de la arquitectura del software (programa y/o estructura de datos), con el uso de modernos conceptos de diseño, puede facilitar enormemente el mantenimiento futuro.
3. Puesto que ya existe un prototipo del software, la productividad de desarrollo debe ser mucho más alta que el promedio.
4. Ahora el usuario experimenta con el software. Por tanto, pueden averiguarse con mayor facilidad los nuevos requisitos y la dirección del cambio.
5. Las herramientas automatizadas para reingeniería facilitarán algunas partes de la labor.
6. Existirá una configuración de software completa (documentos, programas y datos) al completar el mantenimiento preventivo.

Un gran desarrollador interno de software (por ejemplo, un grupo de desarrollo de sistemas de software empresarial para una gran compañía de productos al consumidor) puede tener una producción de 500 a 2 000 programas dentro de su dominio de responsabilidad. Dichos programas pueden clasificarse por importancia y luego revisarse como candidatos para ingeniería hacia adelante.

El proceso de ingeniería hacia adelante aplica los principios, conceptos y métodos de la ingeniería de software para volver a crear una aplicación existente. En la mayoría de los casos, la ingeniería hacia adelante no crea simplemente un equivalente moderno de un programa más antiguo. En vez de ello, en el esfuerzo de reingeniería se integran nuevos requisitos de usuario y tecnología. El programa redesarrollado extiende las capacidades de la aplicación más antigua.

29.8.1 Ingeniería hacia adelante para arquitecturas cliente-servidor

Durante las décadas pasadas, muchas aplicaciones de mainframe se sometieron a reingeniería para alojar arquitecturas cliente-servidor (incluidas las *webapps*). En esencia, los recursos de cómputo centralizados (incluido el software) se distribuyen entre muchas plataformas cliente. Aunque pueden diseñarse varios entornos distribuidos, la aplicación mainframe típica que se somete a reingeniería en una arquitectura cliente-servidor tiene las siguientes características:

- La funcionalidad de la aplicación migra a cada computadora cliente.
- Se implementan nuevas interfaces GUI en los sitios cliente.
- Las funciones de base de datos se ubican en el servidor.
- La funcionalidad especializada (por ejemplo, análisis con uso intenso de computadora) puede permanecer en el sitio servidor.
- Deben establecerse nuevos requisitos de comunicaciones, seguridad, archivado y control en los sitios cliente y servidor.

En algunos casos, la migración a una arquitectura cliente-servidor debe abordarse no como reingeniería, sino como un nuevo esfuerzo de desarrollo. La reingeniería entra al cuadro solamente cuando deba integrarse funcionalidad específica del sistema antiguo en la nueva arquitectura.

Es importante observar que la migración de mainframe a cómputo cliente-servidor requiere reingeniería tanto de la empresa como del software. Además, debe establecerse una "infraestructura de red empresarial" [Jay94].

La reingeniería para aplicaciones cliente-servidor comienza con un profundo análisis del entorno empresarial que abarca el mainframe existente. Pueden identificarse tres capas de abstracción. La *base de datos* que se asienta en los cimientos de una arquitectura cliente-servidor gestiona las transacciones y consultas de aplicaciones del servidor. Aunque dichas transacciones y consultas deben controlarse dentro del contexto de un conjunto de reglas empresariales (definidas por un proceso empresarial existente o de reingeniería), las aplicaciones cliente proporcionan funcionalidad dirigida a la comunidad usuaria.

Las funciones del sistema de gestión de base de datos existente y la arquitectura de datos de la base de datos existente deben someterse a ingeniería inversa como precursor del rediseño de la capa cimiento de la base de datos. En algunos casos, se crea un nuevo modelo de datos (capítulo 6). En todo caso, la base de datos cliente-servidor se somete a reingeniería para garantizar que las transacciones se ejecutan en forma consistente, que todas las actualizaciones se realizan sólo por usuarios autorizados, que las reglas empresariales núcleo se refuerzan (por ejemplo, antes de borrar el registro de un vendedor, el servidor se asegura de que no existan cuentas por pagar, contratos o comunicaciones para dicho proveedor), que las consultas puedan acomodarse de manera eficiente y que se establece la capacidad completa de archivado.

Las capas de reglas empresariales representan software residente tanto en el cliente como en el servidor. Este software realiza tareas de control y coordinación para garantizar que las transacciones y consultas entre la aplicación cliente y la base de datos se conforman con el proceso empresarial establecido.

Las capas de aplicaciones cliente implementan funciones empresariales que requieren grupos específicos de usuarios finales. En muchas instancias, una aplicación mainframe se segmenta en algunas aplicaciones de escritorio más pequeñas sometidas a reingeniería. La comunicación entre aplicaciones de escritorio (cuando sea necesario) se controla mediante la capa de reglas empresariales.

Un análisis profundo del diseño y reingeniería de software cliente-servidor se deja para libros dedicados a la materia. Si tiene más interés, consulte [Van02], [Cou00] u [Orf99].

29.8.2 Ingeniería hacia adelante para arquitecturas orientadas a objetos

La ingeniería de software orientada a objetos se ha convertido en el paradigma de desarrollo elegido por muchas organizaciones de software. Pero, ¿qué hay de las aplicaciones existentes que se desarrollaron usando métodos convencionales? En algunos casos, la respuesta es dejar tales aplicaciones "como están". En otras, las aplicaciones antiguas deben someterse a reingeniería, de modo que puedan integrarse con facilidad en sistemas grandes orientados a objetos.

La reingeniería de software convencional en una implementación orientada a objeto usa muchas de las mismas técnicas estudiadas en la parte 2 de este libro. Primero, el software existente se somete a ingeniería inversa para que puedan crearse modelos adecuados de datos, funciones y comportamientos. Si el sistema sometido a reingeniería extiende la funcionalidad o comportamiento de la aplicación original, se crean casos de uso (capítulos 5 y 6). Los modelos de datos creados durante la ingeniería inversa se usan entonces en conjunción con el modelado CRC (capítulo 6) para establecer la base para la definición de clases. Se definen entonces las jerarquías de clase, modelos objeto-relacional, modelos objeto- comportamiento y subsistemas, y se comienza el diseño orientado a objetos.

Conforme avanza la ingeniería hacia adelante orientada a objetos desde el análisis hacia el diseño, puede invocarse un modelo de proceso ISBC (capítulo 10). Si la aplicación existente reside dentro de un dominio que ya está poblado con muchas aplicaciones orientadas a objetos, es probable que exista una robusta librería de componentes y que pueda usarse durante la ingeniería hacia adelante.

Para aquellas clases que deban someterse a ingeniería desde cero, es posible reutilizar algoritmos y estructuras de datos de la aplicación convencional existente. No obstante, las mismas deben rediseñarse para estar de acuerdo con la arquitectura orientada a objetos.

29.9 Economía de la reingeniería

En un mundo perfecto, todo programa no mantenible se retiraría de inmediato para sustituirlo con aplicaciones sometidas a reingeniería de alta calidad desarrolladas mediante modernas

Cita:

"Puede pagar un poco ahora o puede pagar mucho tiempo después."

Anuncio en una empresa de venta de autos que sugiere una afinación.

prácticas de ingeniería de software. Pero se vive en un mundo de recursos limitados. La reingeniería gasta recursos que pueden usarse para otros propósitos empresariales. Por tanto, antes de que una organización intente someter a reingeniería una aplicación existente, debe realizar un análisis costo-beneficio.

Sneed [Sne95] propone un modelo de análisis costo-beneficio para reingeniería, definiendo nueve parámetros:

P_1 = costo de mantenimiento anual actual para una aplicación
P_2 = costo de operaciones anuales actuales para una aplicación
P_3 = valor empresarial anual actual de una aplicación
P_4 = costo de mantenimiento anual predicho después de reingeniería
P_5 = costo de operaciones anuales predichas después de reingeniería
P_6 = valor empresarial anual predicho después de reingeniería
P_7 = costo de reingeniería estimado
P_8 = tiempo calendario de reingeniería estimado
P_9 = factor de riesgo de reingeniería ($P_9 = 1.0$ es nominal)
L = vida esperada del sistema

El costo asociado con el mantenimiento continuo de una aplicación candidata (es decir, la reingeniería no se realiza) puede definirse como

$$C_{\text{mant}} = [P_3 - (P_1 + P_2)] \times \text{L} \tag{29.1}$$

El costo asociado con reingeniería se define usando la siguiente relación:

$$C_{\text{reing}} = P_6 - (P_4 + P_5) \times (L - P_8) - (P_7 \times P_9) \tag{29.2}$$

Con los costos presentados en las ecuaciones 29.1 y 29.2, el beneficio global de la reingeniería puede calcularse como

$$\text{Beneficio en costo} = C_{\text{reing}} - C_{\text{mant}} \tag{29.3}$$

El análisis costo-beneficio que se presenta en estas ecuaciones puede realizarse para todas las aplicaciones de alta prioridad identificadas durante el análisis de inventario (sección 29.4.2). Aquellas aplicaciones que muestren el mayor costo-beneficio pueden marcarse para reingeniería, mientras que el trabajo sobre otras puede posponerse hasta que haya recursos disponibles.

29.10 Resumen

El mantenimiento y soporte del software son actividades en marcha que ocurren a lo largo de todo el ciclo de vida de una aplicación. Durante dichas actividades se corrigen defectos, se adaptan aplicaciones a un entorno operativo o empresarial cambiante, se implementan mejoras a petición de los participantes y se da soporte a los usuarios conforme integran una aplicación en su flujo de trabajo personal o empresarial.

La reingeniería ocurre en dos niveles de abstracción diferentes. En el nivel empresarial, la reingeniería se enfoca en el proceso empresarial con la intención de realizar cambios para mejorar la competitividad en alguna área de la empresa. En el nivel de software, la reingeniería examina los sistemas y aplicaciones de información con la intención de reestructurar o reconstruirlos de modo que muestren mayor calidad.

La reingeniería de procesos empresarial define las metas de la empresa, identifica y evalúa los procesos empresariales existentes (en el contexto de metas definidas), especifica y diseña procesos revisados y crea prototipos, los refina y ejemplifica dentro de una empresa. La RPE representa un enfoque que se extiende más allá del software. El resultado de la RPE con frecuen-

cia es la definición de formas en las que las tecnologías de la información pueden apoyar mejor a la empresa.

La reingeniería de software abarca una serie de actividades que incluyen análisis de inventario, reestructuración de documentos, ingeniería inversa, reestructuración de programa y datos e ingeniería hacia adelante. La intención de dichas actividades es crear versiones de programas existentes que muestren mayor calidad y mejor mantenibilidad, mismos que serán viables bien entrado el siglo XXI.

El costo-beneficio de la reingeniería puede determinarse de manera cuantitativa. El costo del *status quo*, es decir, el costo asociado con el soporte y mantenimiento actuales de una aplicación existente, se compara con los costos de proyectos de la reingeniería y con la reducción resultante en costos de mantenimiento y soporte. En casi todos los casos en los que un programa tenga una vida larga y en el momento muestre pobre mantenibilidad o soportabilidad, la reingeniería representa una estrategia empresarial efectiva en costo.

PROBLEMAS Y PUNTOS POR EVALUAR

29.1. Considere cualquier trabajo que el lector haya tenido durante los cinco años anteriores. Describa el proceso empresarial en el que participó. Use el modelo RPE descrito en la sección 29.4.2 para recomendar cambios al proceso con la intención de hacerlo más eficiente.

29.2. Realice algo de investigación acerca de la eficacia de la reingeniería de procesos de empresa. Presente argumentos a favor y en contra para este enfoque.

29.3. Su instructor seleccionará uno de los programas que hayan desarrollado en la clase durante este curso. Intercambie su programa al azar con alguien más en la clase. No explique o repase el programa. Ahora, implemente una mejora (especificada por su instructor) en el programa que haya recibido.

a) Realice todas las tareas de ingeniería de software, incluida una breve explicación (mas no con el autor del programa).
b) Siga cuidadosamente la pista a todos los errores que encuentre durante las pruebas.
c) Analice en clase sus experiencias.

29.4. Explore la lista de verificación del análisis de inventario que se presenta en el sitio web del libro, e intente desarrollar un sistema de calificación de software cuantitativo que pueda aplicar a programas existentes con la intención de elegir programas candidato para reingeniería. Su sistema debe extenderse más allá del análisis económico que se presentó en la sección 29.9.

29.5. Sugiera alternativas a la documentación impresa o electrónica convencional que puedan servir como base para la reestructuración de documentos. (Sugerencia: piense en nuevas tecnologías descriptivas que puedan usarse para comunicar la intención del software.)

29.6. Algunas personas creen que la tecnología de inteligencia artificial aumentará el nivel de abstracción del proceso de ingeniería inverso. Realice investigación acerca de este tema (es decir, el uso de IA para ingeniería inversa) y escriba un breve ensayo que adopte una postura acerca de este punto.

29.7. ¿Por qué es difícil lograr la completitud conforme aumenta el nivel de abstracción?

29.8. ¿Por qué debe aumentar la interactividad si aumenta la completitud?

29.9. Con información obtenida en la web, presente a su clase las características de tres herramientas de ingeniería inversa.

29.10. Existe una sutil diferencia entre reestructuración e ingeniería hacia adelante. ¿Cuál es?

29.11. Investigue la literatura y/o fuentes en internet para encontrar uno o más artículos que analizan estudios de caso de reingeniería de mainframe a cliente-servidor. Presente un resumen.

29.12. ¿Cómo determinaría de P_4 a P_7 en el modelo costo-beneficio que se presentó en la sección 29.9?

Lecturas y fuentes de información adicionales

Es irónico que el mantenimiento y el soporte de software representen las actividades más costosas en la vida de una aplicación y, sin embargo, se hayan escrito menos libros acerca de mantenimiento y soporte que de cualquier otro tema importante de la ingeniería de software. Entre las adiciones recientes a la literatura están los libros de Jarzabek (*Effective Software Maintenance and Evolution,* Auerbach, 2007), Grubb y Takang *(Software Maintenance: Concepts and Practice,* World Scientific Publishing Co., 2a. ed., 2003), y Pigoski (*Practical Software Maintenance,* Wiley, 1996). Éstos cubren las prácticas básicas de mantenimiento y soporte, y presentan una guía administrativa útil. Las técnicas de mantenimiento que se enfocan en entornos cliente-servidor se estudian en Schneberger (*Client/Server Software Maintenance*, McGraw-Hill, 1997). La investigación actual en "evolución del software" se presenta en una antología editada por Mens y Demeyer (*Software Evolution*, Springer, 2008).

Como muchos temas calientes en la comunidad empresarial, el alboroto que rodea a la reingeniería de procesos de empresa dio lugar a una visión más pragmática de la materia. Hammer y Champy (*Reengineering the Corporation*, HarperBusiness, edición revisada, 2003) precipitaron un interés temprano con su libro tan solicitado. Otros libros de Smith y Fingar [*Business Process Management (BPM): The Third Wave,* Meghan-Kiffer Press, 2003], Jacka y Keller (*Business Process Mapping: Improving Customer Satisfaction,* Wiley, 2001), Sharp y McDermott (*Workflow Modeling,* Artech House, 2001), Andersen (*Business Process Improvement Toolbox,* American Society for Quality, 1999), y Harrington *et al*. (*Business Process Improvement Workbook,* McGraw-Hill, 1997) presentan estudios de caso y lineamientos detallados para RPE.

Fong (*Information Systems Reengineering and Integration*, Springer, 2006) describe técnicas de conversión de bases de datos, ingeniería inversa e ingeniería hacia adelante para grandes sistemas de información. Demeyer *et al.* (*Object Oriented Reengineering Patterns*, Morgan Kaufmann, 2002) ofrecen una visión basada en patrones sobre cómo refactorizar y/o someter a reingeniería sistemas OO. Secord *et al.* (*Modernizing Legacy Systems,* Addison-Wesley, 2003), Ulrich (*Legacy Systems: Transformation Strategies,* Prentice Hall, 2002), Valenti (*Successful Software Reengineering,* IRM Press, 2002), y Rada (*Reengineering Software: How to Reuse Programming to Build New, State-of-the-Art Software,* Fitzroy Dearborn Publishers, 1999) se enfocan en estrategias y prácticas para reingeniería en un nivel técnico. Miller (*Reengineering Legacy Software Systems*, Digital Press, 1998) "ofrece un marco conceptual para mantener los sistemas de aplicación en sincronía con las estrategias empresariales y los cambios tecnológicos".

Cameron (*Reengineering Business for Success in the Internet Age,* Computer Technology Research, 2000) y Umar [*Application (Re)Engineering: Building Web-Based Applications and Dealing with Legacies,* Prentice Hall, 1997] proporcionan valiosos lineamientos para organizaciones que quieren transformar sistemas heredados en un entorno basado en web. Cook (*Building Enterprise Information Architectures: Reengineering Information Systems*, Prentice Hall, 1996) analiza el puente entre RPE y tecnología de la información. Aiken (*Data Reverse Engineering*, McGraw-Hill, 1996) analiza cómo recuperar, reorganizar y reutilizar datos organizacionales. Arnold (*Software Reengineering*, IEEE Computer Society Press, 1993) reunió una excelente antología de los primeros ensayos que se enfocan en las tecnologías de reingeniería de software.

En internet, está disponible una gran variedad de fuentes de información acerca de la reingeniería de software. Una lista actualizada de referencias en la World Wide Web que son relevantes para el mantenimiento y la reingeniería de software puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

PARTE

Cinco

TEMAS AVANZADOS

En esta parte de *Ingeniería del software. Un enfoque práctico*, se considerarán algunos temas avanzados que extenderán su comprensión de la ingeniería del software. En los siguientes capítulos se abordan las siguientes preguntas:

- ¿Qué es el mejoramiento del proceso de software y cómo puede usarse para optimizar el estado de la práctica de la ingeniería del software?
- ¿Qué tendencias emergentes es posible que tengan una influencia significativa sobre la práctica de la ingeniería del software en la siguiente década?
- ¿Cuál es el camino por venir para los ingenieros del software?

Una vez respondidas dichas preguntas, comprenderá los temas que pueden tener un profundo impacto sobre la ingeniería del software en los años por venir.

CAPÍTULO

30 MEJORAMIENTO DEL PROCESO DE SOFTWARE

Mucho antes de que se usara ampliamente la frase "mejoramiento del proceso de software", el autor trabajó con grandes corporaciones con la intención de mejorar el estado de sus prácticas de ingeniería del software. Como consecuencia de sus experiencias, el autor escribió un libro titulado *Making Software Engineering Happen* [Pre88]. En el prefacio de dicho libro hizo el siguiente comentario:

> Durante los pasados diez años, he tenido la oportunidad de ayudar a algunas compañías grandes a implementar prácticas de ingeniería del software. La tarea es difícil y rara vez pasa tan suavemente como uno quisiera; pero cuando triunfa, los resultados son profundos: los proyectos de software tienen más probabilidad de completarse con el tiempo, mejora la comunicación entre todos los constituyentes involucrados en el desarrollo de software, el nivel de confusión y caos que con frecuencia prevalece para grandes proyectos de software se reduce de manera sustancial, el número de errores que encuentra el cliente disminuye sustancialmente, la credibilidad de la organización de software aumenta y la administración tiene un problema menos por el cual preocuparse.
>
> Pero no todo es dulzura y luz. Muchas compañías intentan implementar prácticas de ingeniería del software y caen en la frustración. Otras llegan a medio camino y nunca ven los beneficios anotados anteriormente. Otras más lo hacen en forma tan ruda que da como resultado rebelión abierta entre el personal técnico y los administradores, con la posterior pérdida de moral.

Aunque tales palabras se escribieron hace más de 20 años, siguen siendo igualmente ciertas el día de hoy.

Conforme se avanza hacia la segunda década del siglo XXI, la mayoría de las principales organizaciones de ingeniería del software intentaron "hacer posible la ingeniería del software". Algunas implementaron prácticas individuales que ayudaron a mejorar la calidad del producto que construían y la oportunidad de su entrega. Otras establecieron un proceso de software "maduro" que guía las actividades técnicas y administrativas del proyecto. Pero otras continúan luchando. Sus prácticas son acierto y error, y su proceso es *ad hoc*. Ocasionalmente, su trabajo

UNA MIRADA RÁPIDA

¿Qué es? El mejoramiento del proceso de software abarca un conjunto de actividades que conducirán a un mejor proceso de software y, en consecuencia, a software de mayor calidad y a su entrega en forma más oportuna.

¿Quién lo hace? El personal que impulsa el MPS (mejoramiento de proceso de software) proviene de tres grupos: gerentes técnicos, ingenieros de software e individuos que tienen responsabilidad en el aseguramiento de la calidad.

¿Por qué es importante? Algunas organizaciones de software tienen poco más que un proceso de software *ad hoc*. Conforme trabajan para mejorar sus prácticas de ingeniería del software, deben abordar las debilidades en sus procesos existentes e intentar mejorar su enfoque para el trabajo de software.

¿Cuáles son los pasos? El enfoque del MPS es iterativo y continuo, pero puede verse en cinco pasos: 1) valoración del proceso de software actual, 2) educación y capacitación de profesionales y gerentes, 3) selección y justificación de elementos de proceso, métodos de ingeniería del software y herramientas, 4) implementación del plan MPS y 5) evaluación y afinación con base en los resultados del plan.

¿Cuál es el producto final? Aunque existen muchos productos operativos MPS intermedios, el resultado final es un proceso de software mejorado que conduce a software de mayor calidad.

¿Cómo me aseguro de que lo hice bien? El software que produce su organización se entregará con menos defectos, se reducirá la repetición del trabajo en cada etapa del proceso de software y la entrega oportuna será mucho más probable.

es espectacular, pero, en promedio, cada proyecto es una aventura, y nadie sabe si terminará mal o bien.

Así que, ¿cuál de estas cohortes necesita mejoramiento del proceso de software? La respuesta (que puede sorprenderle) es *ambas*. Las que triunfaron en hacer posible la ingeniería del software no pueden volverse complacientes. Deben trabajar de manera continua para mejorar su enfoque de la ingeniería del software. Y las que luchan deben comenzar su viaje por el camino hacia el mejoramiento.

30.1 ¿Qué es MPS?

MPS implica un proceso de software definido, un enfoque organizacional y una estrategia para el mejoramiento.

El término *mejoramiento del proceso de software* (MPS) implica muchas cosas. Primero, que los elementos de un proceso de software efectivo pueden definirse en forma efectiva; segundo, que un enfoque organizacional existente sobre el desarrollo del software puede valorarse en contraste con dichos elementos; y tercero, que es posible definir una estrategia de mejoramiento significativa. La estrategia MPS transforma el enfoque existente sobre el desarrollo del software en algo que es más enfocado, más repetible y más confiable (en términos de la calidad del producto producido y de la oportunidad de la entrega).

Puesto que el MPS no es gratuito, debe entregar un rendimiento sobre la inversión. El esfuerzo y el tiempo que se requieren para implementar una estrategia MPS deben pagar por sí mismos en alguna forma mensurable. Para hacer esto, los resultados del proceso y la práctica mejorados deben conducir a una reducción en los "problemas" del software que cuestan tiempo y dinero. Debe reducir el número de defectos que se entregan a los usuarios finales, la cantidad de repetición de proceso debida a problemas de calidad, los costos asociados con el mantenimiento y el soporte del software (capítulo 29) y los costos indirectos que ocurren cuando el software se entrega tarde.

30.1.1 Enfoques del MPS

Cita:

"Mucha de la crisis del software es autoinfligida, como cuando un presidente del área de informática en ejercicio dice: 'Prefiero que salga mal antes que entregarlo tarde. Siempre podremos repararlo después'."

Mark Paulk

Aunque una organización puede elegir un enfoque relativamente informal del MPS, la gran mayoría elige uno de los marcos conceptuales MPS. Un *marco conceptual MPS* define: 1) un conjunto de características que deben presentarse si quiere lograrse un proceso de software efectivo, 2) un método para valorar si dichas características están presentes, 3) un mecanismo para resumir los resultados de cualquier valoración y 4) una estrategia para auxiliar a una organización de software a implementar aquellas características del proceso que sean débiles o que hagan falta.

Un marco conceptual MPS valora la "madurez" del proceso de una organización y proporciona un indicio cualitativo de su nivel de madurez. De hecho, con frecuencia se aplica el término "modelo de madurez" (sección 30.1.2). En esencia, el marco conceptual MPS abarca un modelo de madurez que a su vez incorpora un conjunto de indicadores de calidad de proceso que ofrecen una medida global de la calidad del proceso que llevará a la calidad del producto.

La figura 30.1 proporciona un panorama de un marco conceptual MPS típico. Se muestran los elementos clave del marco conceptual y su relación mutua.

¿Qué grupos impulsan un esfuerzo MPS?

Debe observarse que no existe un marco conceptual MPS universal. De hecho, el marco conceptual MPS que elige una organización refleja las áreas que impulsan el esfuerzo MPS. Conradi [Con96] define seis diferentes grupos de apoyo al MPS:

Certificadores de calidad. Los esfuerzos de mejoramiento de proceso que impulsa este grupo se enfocan en la siguiente relación:

Calidad (*Proceso*) ⇒ **Calidad** (*Producto*)

Su enfoque consiste en enfatizar los métodos de valoración y examinar un conjunto bien definido de características que le permiten determinar si el proceso muestra calidad. Es

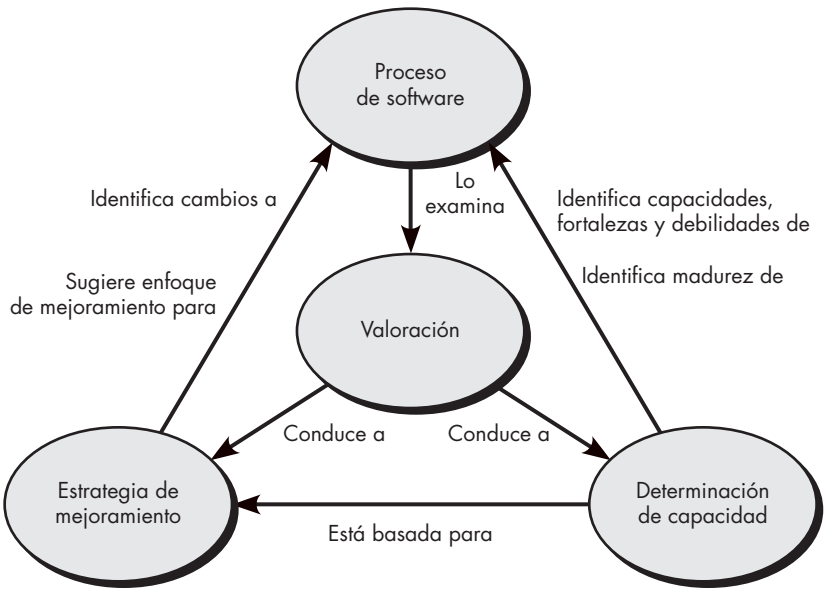


FIGURA 30.1
Elementos de un marco conceptual MPS
Fuente: Adaptado de [Rou02].

más probable que adopten un marco conceptual de proceso, como CMM, SPICE, TickIT o Bootstrap.[1]

Formalistas. Este grupo quiere entender (y cuando es posible, optimizar) el flujo de trabajo del proceso. Para lograrlo, usa lenguajes de modelado de proceso (PML) a fin de crear un modelo del proceso existente y luego diseñar extensiones o modificaciones que harán más efectivo el proceso.

¿Cuáles son los diferentes grupos que apoyan el MPS?

Defensores de las herramientas. Este grupo insiste en un enfoque del MPS asistido por herramientas que modelan el flujo de trabajo y otras características del proceso de manera que pueda analizarse para su mejoramiento.

Profesionales. Este grupo usa un enfoque pragmático, "que enfatiza la administración tradicional de proyecto, calidad y producto, y aplica planificación y métricas en el nivel de proyecto, pero con poco modelado de proceso formal o pronunciamiento de apoyo" [Con96].

Reformadores. La meta de este grupo es el cambio organizacional que pueda conducir a un mejor proceso de software. Tienden a enfocarse más en los temas humanos (sección 30.5) y enfatizan medidas de capacidad humana y estructura.

Ideólogos. Este grupo se enfoca en lo adecuado de un modelo de proceso particular para un dominio de aplicación específico o estructura organizativa. En lugar de los modelos de proceso de software típicos (por ejemplo, modelos iterativos), los ideólogos tendrían mayor interés en un proceso que, por ejemplo, apoyara el reuso o la reingeniería.

Conforme se aplica un marco conceptual MPS, el grupo impulsor (sin importar su enfoque global) debe establecer mecanismos para: 1) apoyar la transición tecnológica, 2) determinar el grado en el que una organización está lista para absorber los cambios de proceso que se propongan y 3) medir el grado en el que se adoptaron los cambios.

1 Cada uno de estos marcos conceptuales MPS se estudia más adelante, en este capítulo.

30.1.2 Modelos de madurez

Un *modelo de madurez* se aplica dentro del contexto de un marco conceptual MPS. La intención del modelo de madurez es proporcionar un indicio global de la "madurez del proceso" que muestra una organización de software, es decir, un indicio de la calidad del proceso de software, el grado en el que los profesionales entienden y aplican el proceso, y el estado general de la práctica de ingeniería del software. Esto se logra usando algún tipo de escala ordinal.

Por ejemplo, el *modelo de madurez de capacidad* (sección 30.4) del Software Engineering Institute sugiere cinco niveles de madurez [Sch96]:

PUNTO CLAVE

Un modelo de madurez define niveles de competencia e implementación de proceso de software.

Nivel 5, optimizado. La organización tiene sistemas de realimentación cuantitativa en su lugar para identificar las debilidades del proceso y fortalecer esos puntos de manera proactiva. Los equipos de proyecto analizan defectos para determinar sus causas; los procesos de software se evalúan y actualizan para evitar que recurran tipos conocidos de defectos.

Nivel 4, gestionado. Métricas de proceso de software y de calidad de producto detalladas establecen el cimiento de evaluación cuantitativa. Las variaciones significativas en el desempeño del proceso pueden distinguirse del ruido aleatorio, y pueden predecirse las tendencias en las cualidades del proceso y el producto.

Nivel 3, definido. Los procesos para administración e ingeniería se documentan, estandarizan e integran en un proceso de software estándar para la organización. Todos los proyectos usan una versión aprobada y a la medida del proceso de software estándar de la organización para desarrollo de software.

Nivel 2, repetible. Se establecen procesos de administración de proyecto básicos para rastrear costo, calendario y funcionalidad. La planificación y administración de nuevos productos se basa en la experiencia con proyectos similares.

Nivel 1, inicial. Pocos procesos definidos, y el éxito depende más del esfuerzo heroico individual que de seguir un proceso y usar un esfuerzo sinérgico de equipo.

La escala de madurez CMM va más allá, pero la experiencia indica que muchas organizaciones muestran niveles de "inmadurez de proceso" [Sch96] que minan cualquier intento racional por mejorar las prácticas de ingeniería de software. Schorsch [Sch06] sugiere cuatro niveles de inmadurez que se encuentran frecuentemente en el mundo real de las organizaciones de desarrollo del software:

? ¿Cómo se reconoce a una organización que resistirá los esfuerzos MPS?

Nivel 0, negligente. Fracaso para permitir que tenga éxito un proceso de desarrollo exitoso. Todos los problemas se perciben como problemas técnicos. Las actividades administrativas y de aseguramiento de la calidad están condenadas a situarse por encima y ser superfluas en relación con las tareas del proceso de desarrollo del software. Se confía en las balas de plata.

Nivel 1, obstructivo. Se imponen procesos contraproducentes. Los procesos se definen rígidamente y se adhieren a la forma que subrayan. Abundan las ceremonias rituales. La administración colectiva impide la asignación de responsabilidad. *Status quo über alles* (sobre todo).

Nivel 2, despreciador. No se preocupa por la buena ingeniería de software institucionalizada. Hay desunión completa entre actividades de desarrollo de software y actividades de mejoramiento del proceso de software y falta completa de programas de capacitación.

Nivel 3, socavación. Desprecio total por la propia organización, descrédito consciente de los esfuerzos de mejoramiento del proceso de software de los pares de la organización. Recompensa al fracaso y al pobre desempeño.

Los niveles de inmadurez de Schorsch son tóxicos para cualquier organización de software. Si usted encuentra alguno de ellos, los intentos por MPS están condenados al fracaso.

La pregunta decisiva es si las escalas de madurez, como las que se proponen como parte del CMM, proporcionan algún beneficio real. El autor cree que lo tienen. Una escala de madurez

proporciona una instantánea fácilmente comprensible de la calidad del proceso que pueden emplear los profesionales y administradores como hito desde el cual puedan planificar estrategias de mejoramiento.

30.1.3 ¿El MPS es para todos?

Durante muchos años, el MPS se vio como una actividad "corporativa", un eufemismo para algo que sólo realizan las grandes compañías. Pero hoy, un porcentaje significativo de todo el desarrollo de software lo realizan compañías que emplean menos de 100 personas. ¿Una compañía pequeña puede iniciar actividades MPS y realizarlas con éxito?

Si un modelo de proceso específico o enfoque MPS se percibe como excesivo en su organización, probablemente lo es.

Existen sustanciales diferencias culturales entre grandes y pequeñas organizaciones de desarrollo de software. No debe sorprender que las organizaciones pequeñas sean más informales, apliquen menos prácticas estándar y tiendan a la autoorganización. También tienden a enorgullecerse por la "creatividad" de los miembros individuales de la organización de software, e inicialmente ven un marco conceptual MPS como excesivamente burocrático y pesado. Sin embargo, el mejoramiento de los procesos es tan importante para una organización pequeña como para una grande.

Dentro de las organizaciones pequeñas, la implementación de un marco conceptual MPS requiere recursos que pueden tener un suministro reducido. Los administradores deben asignar personal y dinero para lograr que ocurra la ingeniería del software. Por tanto, sin importar el tamaño de la organización del software, es razonable considerar la motivación empresarial para el MPS.

Éste se aprobará e implementará sólo después de que sus proponentes demuestren apalancamiento financiero [Bir98], que se demuestra al examinar los beneficios técnicos (por ejemplo, menos defectos entregados al campo, reelaboración reducida, menores costos de mantenimiento o tiempo de llegada al mercado más rápido) y traducirlos en dinero. En esencia, deben demostrar un rendimiento realista sobre la inversión (sección 30.7) para los costos MPS.

30.2 El proceso MPS

La parte dura del MPS no es establecer las características que definen un proceso de software de alta calidad o la creación de un modelo de madurez de proceso. Ésas son relativamente sencillas. La parte dura es establecer un consenso para iniciar MPS y definir una estrategia continua a fin de implementarla a través de una organización de software.

El Software Engineering Institute desarrolló IDEAL, "un modelo de mejoramiento organizacional que funciona como mapa de caminos para iniciar, planificar e implementar acciones de mejoramiento" [SEI08]. IDEAL es representativo de muchos modelos de proceso para MPS y define cinco actividades distintas: inicio, diagnóstico, establecimiento, acción y aprendizaje, que guían a una organización a través de las actividades MPS.

En este libro se presenta un mapa de caminos un tanto diferente para MPS, con base en el modelo de proceso para MPS originalmente propuesto en [Pre88]. Aplica una filosofía de sentido común que requiere que una organización 1) se observe en el espejo, 2) se vuelva más astuta para tomar elecciones inteligentes, 3) seleccione el modelo de proceso (y los elementos tecnológicos relacionados) que satisfagan mejor sus necesidades, 4) ejemplifique el modelo en su entorno operativo y su cultura y 5) evalúe lo que se hizo. Estas cinco actividades (analizadas en las subsecciones[2] que siguen) se aplican en forma iterativa (cíclica) con la intención de fomentar el mejoramiento de proceso continuo.

2 Algunos de los contenidos de estas secciones han sido adaptados de [Pre88] con autorización.

30.2.1 Valoración y análisis de la desviación

Cualquier intento por mejorar un proceso de software existente sin primero valorar la eficacia de las actividades del marco conceptual y las prácticas de ingeniería del software asociadas sería como iniciar un largo viaje hacia una nueva localidad sin idea de dónde se comienza. Se parte con gran movimiento y se vaga por ahí intentando conseguir un rumbo, se emplea mucha energía y se padece de grandes dosis de frustración y, probablemente, se decide que en realidad no se quiere viajar de cualquier forma. Dicho de manera simple, antes de comenzar cualquier viaje, es buena idea saber precisamente dónde se está.

Asegúrese de entender sus fortalezas así como sus debilidades. Si es inteligente, construirá sobre la base de las fortalezas.

La primera actividad del mapa de caminos, llamada *valoración*, le permite adquirir rumbo. La intención de la valoración es descubrir las fortalezas y las debilidades en la forma en la que su organización aplica el proceso de software existente y las prácticas de ingeniería del software que pueblan el proceso.

La valoración examina un amplio rango de acciones y tareas que conducirán a un proceso de alta calidad. Por ejemplo, sin importar el modelo de proceso que elija, la organización de software debe establecer mecanismos genéricos como: enfoques definidos para comunicación con el cliente; establecimiento de métodos para representar requisitos de usuarios; definición de un marco conceptual de gestión del proyecto que incluya definición del ámbito, estimación, calendarización y rastreo del proyecto; métodos de análisis de riesgos; cambio de procedimientos administrativos; actividades de aseguramiento y control de la calidad que incluyan revisiones y muchas otras. Cada una se considera dentro del contexto del marco conceptual y las actividades sombrilla (capítulo 2) que se establecieron y valoraron para determinar si se responden las siguientes preguntas:

- ¿El objetivo de la acción está claramente definido?
- ¿Los productos operativos requeridos como entrada y producidos como salida se identifican y describen?
- ¿Las tareas de trabajo por realizar se describen claramente?
- ¿Las personas que deben realizar la acción se identifican por rol?
- ¿Se establecieron criterios de entrada y salida?
- ¿Se establecieron métricas para la acción?
- ¿Hay herramientas disponibles para apoyar la acción?
- ¿Existe algún programa de capacitación explícito que aborde la acción?
- ¿La acción se realiza de manera uniforme para todos los proyectos?

Aunque las preguntas anotadas implican una respuesta de *sí* o *no*, el papel de la valoración es mirar detrás de la respuesta para determinar si la acción en cuestión se realiza con las mejores prácticas.

Conforme se realiza el proceso de valoración, usted (o quienes se contraten para realizar la valoración) también deben enfocarse en los siguientes atributos:

¿Qué atributos genéricos se observan durante la valoración?

Consistencia. ¿Las actividades, acciones y tareas importantes se aplican de manera consistente a través de todos los proyectos de software y por todos los equipos de software?

Sofisticación. ¿Las acciones administrativas y técnicas se realizan con un nivel de sofisticación que implica una comprensión profunda de las mejores prácticas?

Aceptación. ¿El proceso de software y la práctica de ingeniería del software se aceptan ampliamente por parte del personal administrativo y técnico?

Compromiso. ¿La administración comprometió los recursos requeridos para lograr consistencia, sofisticación y aceptación?

La diferencia entre aplicación local y mejores prácticas representa una "brecha" que ofrece oportunidades para el mejoramiento. El grado en el cual se logren consistencia, sofisticación, aceptación y compromiso indica la cantidad de cambio cultural que se requerirá para lograr mejoría significativa.

30.2.2 Educación y capacitación

Aunque pocas personas de software cuestionan los beneficios de un proceso de software organizado y ágil o las prácticas sólidas de ingeniería del software, muchos profesionales y administradores no conocen lo suficiente acerca de alguno de los temas.[3] Como consecuencia, percepciones imprecisas de los procesos y de las prácticas conducen a decisiones inadecuadas cuando se introduce un marco conceptual MPS. Se concluye entonces que un elemento clave de cualquier estrategia MPS es la educación y capacitación de los profesionales, gerentes técnicos y gerentes ejecutivos que tengan contacto directo con la organización de software. Con ese fin, deben realizarse tres tipos de educación y capacitación:

Intente proporcionar capacitación "justo a tiempo" dirigida a las necesidades reales de un equipo de software.

Conceptos y métodos genéricos. Dirigida tanto a gerentes como a profesionales, esta categoría subraya tanto procesos como práctica. La intención es proporcionar a los profesionales las herramientas intelectuales necesarias para aplicar el proceso de software de manera efectiva y la toma de decisiones racionales acerca de las mejorías del proceso.

Tecnología y herramientas específicas. Dirigida principalmente a profesionales, esta categoría subraya las tecnologías y herramientas que se adoptaron para el uso local. Por ejemplo, si se eligió UML para modelado de análisis y diseño, se establece un programa de estudios de capacitación para ingeniería del software usando UML.

Comunicación empresarial y temas relacionados con la calidad. Dirigida a todos los participantes, esta categoría se enfoca en los temas "blandos" que ayudan a una mejor comunicación entre los participantes y fomentan un mayor foco de calidad.

En un contexto moderno, educación y capacitación pueden entregarse en varias formas distintas. Todo puede ofrecerse como parte de una estrategia MPS, desde podcast hasta capacitación basada en internet (por ejemplo, [QAI08]), pasando por DVD y cursos en aulas.

30.2.3 Selección y justificación

Conforme elija, asegúrese de considerar la cultura de su organización y el nivel de aceptación que cada elección probablemente provocará.

Una vez completada la actividad de valoración inicial[4] e iniciada la educación, una organización de software debe comenzar a elegir, lo que ocurre durante una *actividad de selección y justificación* en la que se eligen características del proceso y se especifican métodos de ingeniería del software determinados para poblar el proceso de software.

Primero, debe elegir el modelo de proceso (capítulos 2 y 3) que se ajuste mejor a su organización, a sus participantes y al software que construirán. Debe decidir cuáles de las actividades del conjunto del marco conceptual se aplicarán, los principales productos operativos que se producirán y los puntos de verificación de aseguramiento de la calidad que permitirán a su equipo valorar el progreso. Si la actividad de valoración MPS indica debilidades específicas (por ejemplo, no hay funciones SQA formales), debe enfocar su atención en las características del proceso que abordarán directamente dichas debilidades.

A continuación, desarrolle un trabajo innovador para cada actividad de marco conceptual (por ejemplo, modelado) y defina el conjunto de tareas que se aplicarán para un proyecto típico. También debe considerar los métodos de ingeniería del software que pueden aplicarse para

3 Si ha invertido tiempo en la lectura de este libro, ¡no será uno de ellos!

4 En realidad, la valoración es una actividad continua. Se realiza de manera periódica con la intención de determinar si la estrategia MPS logró sus metas inmediatas y si se establece el escenario para una futura mejoría.

lograr dichas tareas. Conforme se hagan estas elecciones, educación y capacitación deben coordinarse para garantizar el reforzamiento de la comprensión.

De manera ideal, todos trabajan en conjunto para seleccionar varios procesos y elementos tecnológicos y para moverse suavemente hacia la actividad de instalación o migración (sección 30.2.4). En realidad, la selección puede ser un camino empedrado. Con frecuencia es difícil lograr consenso entre diferentes grupos. Si un comité establece los criterios para la selección, las personas pueden discutir sin fin acerca de si los criterios son adecuados y si una elección realmente satisface los criterios que se establecieron.

Es cierto que una mala elección puede hacer más daño que bien, pero "parálisis por análisis" significa que ocurre poco progreso, tal vez, y que persisten los problemas del proceso. En tanto las características del proceso o los elementos tecnológicos tengan buena posibilidad de satisfacer las necesidades de una organización, a veces es mejor jalar el gatillo y hacer la selección, en lugar de esperar la solución óptima.

Una vez hecha la elección, deben emplearse tiempo y dinero para demostrarla dentro de una organización, y dichos gastos de recursos deben justificarse. En la sección 30.7 se presenta un análisis acerca de la justificación del costo y el rendimiento sobre la inversión para el MPS.

30.2.4 Instalación/migración

La *instalación* es el primer punto donde una organización de software siente los efectos de los cambios implementados como consecuencia del mapa de caminos MPS. En algunos casos, se recomienda un proceso completamente nuevo para una organización. Las actividades de marco conceptual, acciones de ingeniería del software y tareas de trabajo individuales deben definirse e instalarse como parte de una nueva cultura de ingeniería del software. Tales cambios representan una transición organizativa y tecnológica sustancial, y deben administrarse con mucho cuidado.

En otros casos, los cambios asociados con MPS son relativamente menores, lo que representa pequeñas modificaciones, pero significativas, a un modelo de proceso existente. A tales cambios con frecuencia se les conoce como *migración de proceso*. En la actualidad, muchas organizaciones de software tienen un "proceso" en su lugar. El problema es que no funciona de forma efectiva. Por tanto, una estrategia más efectiva es una *migración* incremental de un proceso (que no funciona tan bien como se desea) a otro proceso.

Instalación y migración en realidad son actividades de *rediseño de proceso de software* (RPS). Scacchi [Sca00] afirma que "el RPS se preocupa por la identificación, aplicación y refinamiento de nuevas formas de mejorar dramáticamente y de transformar los procesos de software". Cuando se inicia un proceso formal de RPS se consideran tres modelos de proceso diferentes: 1) el proceso existente ("como es"), 2) un proceso transicional ("de aquí a allá") y 3) el proceso meta ("por ser"). Si este último es significativamente diferente del proceso existente, el único enfoque racional de la instalación es una estrategia incremental en la que el proceso transicional se implemente en pasos. El proceso transicional ofrece una serie de puntos que permiten que la cultura de la organización de software se adapte a pequeños cambios a lo largo de un periodo.

30.2.5 Evaluación

Aunque se menciona como la última actividad en el mapa de caminos MPS, la *evaluación* ocurre a lo largo del MPS. La actividad de evaluación valora el grado en el cual los cambios se demostraron y adoptaron, el grado en el que tales cambios dan como resultado mejor calidad de software u otros beneficios tangibles de proceso y el estado global del proceso y de la cultura de la organización conforme avanzan las actividades MPS.

Durante la actividad de evaluación se consideran tanto factores cualitativos como métricas cuantitativas. Desde un punto de vista cualitativo, las actitudes anteriores de administradores y

profesionales acerca del proceso de software pueden compararse con las que tienen después de la instalación de los cambios del proceso. Las métricas cuantitativas (capítulo 25) se recopilan de proyectos que usaron el proceso transicional o del proceso "por ser" y se comparan con métricas similares que se recopilan para proyectos que se realizaron bajo el proceso "como es".

30.2.6 Gestión del riesgo para MPS

PUNTO CLAVE

El MPS con frecuencia fracasa porque los riesgos no se consideraron adecuadamente y no se tuvo un plan de contingencia.

El MPS es una empresa riesgosa. De hecho, más de la mitad de todos los esfuerzos MPS terminan en fracaso. Las razones del fracaso varían enormemente y son específicas de la organización. Entre los riesgos más comunes están: falta de apoyo administrativo, resistencia cultural por parte del personal técnico, una estrategia MPS pobremente planeada, un enfoque excesivamente formal del MPS, selección de un proceso inadecuado, falta de "adquisición" por parte de participantes clave, un presupuesto inadecuado, falta de capacitación de personal, inestabilidad de la organización, entre muchos otros factores. El papel de quienes tienen la responsabilidad del MPS es analizar los riesgos probables y desarrollar una estrategia interna para mitigarlos.

Una organización de software debe gestionar el riesgo en tres puntos clave en el proceso MPS [Sta97b]: antes del inicio del mapa de caminos MPS, durante la ejecución de las actividades MPS (valoración, educación, selección, instalación) y durante la actividad de evaluación que sigue a la ejemplificación de algunas características del proceso. En general, pueden identificarse las siguientes categorías [Sta97b] para factores de riesgo MPS: presupuesto y costo, contenido y entregables, cultura, mantenimiento de entregables MPS, misión y metas, administración de la organización, estabilidad de la organización, participantes en el proceso, calendario de desarrollo MPS, entorno de desarrollo MPS, proceso de desarrollo MPS, administración del proyecto MPS y personal MPS.

Dentro de cada categoría pueden identificarse algunos factores de riesgo genéricos. Por ejemplo, la cultura organizacional tiene un fuerte vínculo con el riesgo. Para la categoría cultura, pueden definirse los siguientes factores de riesgo genéricos[5] [Sta97b]:

- Actitud hacia el cambio, con base en esfuerzos previos por cambiar
- Experiencia con programas de calidad, nivel de éxito
- Orientación de la acción para resolver problemas frente a luchas políticas
- Uso de hechos para gestionar la organización y los negocios
- Paciencia con el cambio; habilidad para pasar tiempo socializando
- Orientación de las herramientas: esperanza de que las herramientas puedan resolver los problemas
- Nivel de "planificación": habilidad de la organización para planificar
- Habilidad de los miembros de la organización para participar abiertamente en las reuniones con varios niveles de la organización
- Habilidad de los miembros de la organización para administrar las reuniones de manera eficaz
- Nivel de experiencia en la organización con procesos definidos

Al usar los factores de riesgo y los atributos genéricos como guía, es posible calcular la exposición al riesgo en la forma siguiente:

$$\text{Exposición} = (\text{probabilidad de riesgo}) \times (\text{pérdida estimada})$$

Puede desarrollar una tabla de riesgo (capítulo 28) para aislar aquellos riesgos que garanticen mayor atención de la administración.

5 Factores de riesgo para cada una de las categorías de riesgo anotadas en esta sección pueden encontrarse en [Sta97b].

30.2.7 Factores de éxito cruciales

En la sección 30.2.6 se observó que el MPS es una empresa riesgosa y que la tasa de fracaso para las compañías que intentan mejorar su proceso es angustiosamente elevada. Los riesgos de la organización, del personal y de la administración de proyecto presentan retos para quienes dirigen cualquier esfuerzo MPS. Aunque la gestión del riesgo es importante, lo es igualmente reconocer aquellos factores cruciales que conducen al éxito.

Después de examinar 56 organizaciones de software y sus esfuerzos MPS, Stelzer y Mellis [Ste99] identifican un conjunto de factores de éxito cruciales (FEC) que deben presentarse si ha de triunfar el MPS. En esta sección se presentan los cinco principales FEC.

? ¿Qué factores de éxito cruciales son vitales para el éxito del MPS?

Compromiso y apoyo de la administración. Como la mayoría de las actividades que precipitan el cambio organizativo y cultural, el MPS triunfará sólo si la administración se involucra de manera activa. Los gerentes ejecutivos deben reconocer la importancia del software para sus compañías y ser patrocinadores activos del esfuerzo MPS. Los gerentes técnicos deben involucrarse enormemente en el desarrollo de la estrategia MPS local. Como anotan los autores del estudio: "el mejoramiento del proceso de software no es factible sin investigar tiempo, dinero y esfuerzo" [Ste99]. El compromiso y el apoyo administrativo son esenciales para sostener dicha inversión.

Involucramiento del personal. El MPS no puede imponerse de manera descendente ni desde el exterior. Si los esfuerzos MPS han de triunfar, el mejoramiento debe ser orgánico, patrocinado por gerentes técnicos y técnicos ejecutivos, y adoptado por profesionales locales.

Integración y comprensión del proceso. El proceso de software no existe en un vacío organizativo. Debe integrarse con otros procesos y requisitos empresariales. Para lograr esto, los responsables del esfuerzo MPS deben tener un conocimiento íntimo de los otros procesos empresariales. Además, deben entender el proceso de software "como es" y valorar cuánto cambio transicional es tolerable dentro de la cultura local.

Una estrategia MPS a la medida. No hay una receta para la estrategia MPS. Como se anotó anteriormente, en este capítulo, el mapa de caminos MPS debe adaptarse al entorno local: deben considerarse cultura de equipo, mezcla de producto, y fortalezas y debilidades locales.

Administración sólida del proyecto MPS. El MPS es un proyecto como cualquier otro. Involucra coordinación, calendarización, tareas paralelas, productos entregables, adaptación (cuando el riesgo se convierte en realidad), políticas, control presupuestal y mucho más. Sin administración activa y efectiva, un proyecto MPS está condenado al fracaso.

30.3 El CMMI

WebRef

Para obtener información más completa acerca del CMMI, dirigirse a **www.sei.cmu.edu/cmmi/**

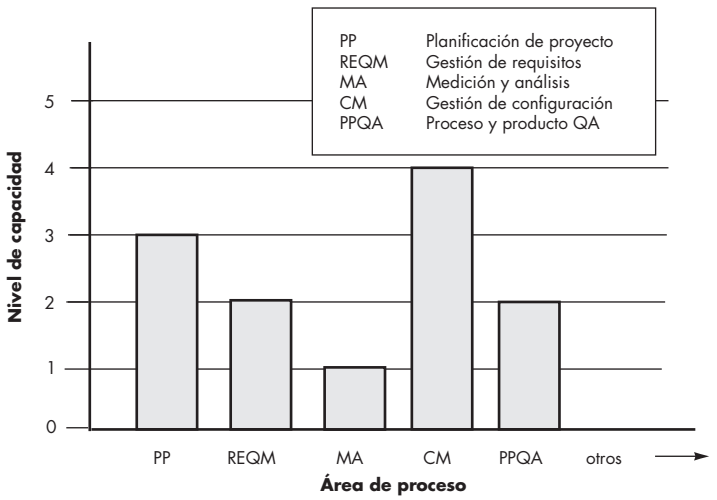
El CMM original se desarrolló y actualizó por parte del Software Engineering Institute a lo largo de los años de 1990 como un marco conceptual MPS completo. Más adelante, evolucionó en la *Integración del Modelo de Madurez de Capacidades* (CMMI) [CMM07], un metamodelo de proceso exhaustivo que se impulsa en un conjunto de sistemas y capacidades de ingeniería del software que deben presentarse conforme las organizaciones alcanzan diferentes niveles de capacidad y madurez del proceso.

El CMMI representa un metamodelo de proceso en dos formas diferentes: 1) como un modelo "continuo" y 2) como un modelo "en etapas". El metamodelo CMMI continuo describe un proceso en dos dimensiones, como se ilustra en la figura 30.2. Cada área de proceso (por ejemplo, planificación de proyecto o gestión de requisitos) se valora formalmente contra metas y prácticas específicas y se clasifica de acuerdo con los siguientes niveles de capacidad:

FIGURA 30.2

Perfil de capacidad de área de proceso CMMI

Fuente: [Phi02].



Nivel 0: *Incompleto*: el área del proceso (por ejemplo, gestión de requisitos) no se realiza o no logra todas las metas y objetivos definidos por la CMMI para la capacidad nivel 1 del área de proceso.

Nivel 1: *Realizado*: todas las metas específicas del área de proceso (como se define mediante la CMMI) están satisfechas. Se realizan las tareas de trabajo requeridas para producir productos operativos definidos.

Nivel 2: *Administrado*: se satisfacen todos los criterios del nivel 1 de capacidad. Además, todo el trabajo asociado con el área de proceso se encuentra acorde con una política definida de manera organizacional; todo el personal que realiza el trabajo tiene acceso a recursos adecuados para tener listo el trabajo; los participantes se involucran de manera activa en el área de proceso según se requiera; todas las tareas del trabajo y los productos operativos se "monitorean, controlan y revisan, y se evalúan para su adhesión a la descripción del proceso" [CMM07].

CONSEJO

Cada organización debe esforzarse por lograr el objetivo de la CMMI. Sin embargo, la aplicación de todos los aspectos del modelo puede ser exagerado.

Nivel 3: *Definido*: se logran todos los criterios del nivel 2 de capacidad. Además, el proceso se "hace a la medida, a partir del conjunto de procesos estándar y de acuerdo con los lineamientos de producción de la organización; contribuye con productos operativos, medidas y otra información para mejorar los procesos activos del proceso organizacional" [CMM07].

Nivel 4: *Administrado cuantitativamente*: se logran todos los criterios del nivel 3 de capacidad. Además, el área de proceso se controla y mejora, usando medición y valoración cuantitativa. "Los objetivos cuantitativos para el rendimiento cualitativo y de proceso se establecen y usan como criterios para gestionar el proceso" [CMM07].

Nivel 5: *Optimizado*: se logran todos los criterios del nivel 4 de capacidad. Además, el área de proceso se adapta y optimiza, usando medios cuantitativos (estadísticos) para satisfacer las necesidades cambiantes del cliente y para mejorar continuamente la eficacia del área de proceso bajo consideración.

La CMMI define cada área de proceso en términos de "metas específicas", y de "prácticas específicas" requeridas para lograr dichas metas. Las *metas específicas* establecen las características que deben existir si las actividades implicadas por un área de proceso han de ser efectivas. Las *prácticas específicas* desglosan una meta en un conjunto de actividades relacionadas con el proceso.

Por ejemplo, la **planificación del proyecto** es una de las ocho áreas de proceso definidas por la CMMI para la categoría "gestión de proyecto".[6] Las metas específicas (ME) y las prácticas específicas (PE) asociadas definidas para **planificación de proyecto** son [CMM07]:

WebRef

En **www.sei.cmu.edu/CMMI/** puede obtenerse información completa además de una versión descargable de la CMMI

ME 1 Establecimiento de estimaciones

PE 1.1-1 Estimación del ámbito del proyecto

PE 1.2-1 Establecimiento de estimaciones de producto operativo y atributos de tarea

PE 1.3-1 Definición de ciclo de vida del proyecto

PE 1.4-1 Determinación de estimaciones de esfuerzo y costo

ME 2 Desarrollo de un plan de proyecto

PE 2.1-1 Establecimiento del presupuesto y calendario

PE 2.2-1 Identificación de riesgos del proyecto

PE 2.3-1 Plan para gestión de datos

PE 2.4-1 Plan para recursos del proyecto

PE 2.5-1 Plan para conocimiento y habilidades necesarias

PE 2.6-1 Plan de involucramiento de participantes

PE 2.7-1 Establecimiento del plan del proyecto

ME 3 Obtención de compromiso del plan

PE 3.1-1 Revisión de planes que afectan el proyecto

PE 3.2-1 Reconciliación de niveles de trabajo y recursos

PE 3.3-1 Obtención de compromiso del plan

Además de las metas y prácticas específicas, la CMMI también define un conjunto de cinco metas genéricas y prácticas relacionadas para cada área de proceso. Cada una de las cinco metas genéricas corresponde a uno de los cinco niveles de capacidad. Por tanto, para lograr un nivel de capacidad particular deben lograrse la meta genérica para dicho nivel y las prácticas genéricas que correspondan a dicha meta. Para ilustrar, las metas genéricas (MG) y las prácticas genéricas (PG) para el área de proceso **planificación del proyecto** son [CMM07]:

MG 1 Logro de metas específicas

PG 1.1 Realización de prácticas base

MG 2 Institucionalización de un proceso administrado

PG 2.1 Establecimiento de una política organizacional

PG 2.2 Plan del proceso

PG 2.3 Provisión de recursos

PG 2.4 Asignación de responsabilidad

PG 2.5 Capacitación de personal

PG 2.6 Gestión de configuraciones

PG 2.7 Identificación e involucramiento de participantes relevantes

PG 2.8 Monitoreo y control del proceso

PG 2.9 Evaluación objetiva de la adhesión

PG 2.10 Revisión de estatus con administración de nivel superior

6 Otras áreas de proceso definidas para "gestión de proyecto" incluyen: monitoreo y control del proyecto, administración de acuerdo con proveedores, administración de proyecto integrado para IPPD, gestión del riesgo, formación de equipo integrado, gestión de proveedor integrado y gestión de proyecto cuantitativo.

MG 3 Institucionalización de un proceso definido

PG 3.1 Establecimiento de un proceso definido

PG 3.2 Recopilación de información de mejoría

MG 4 Institucionalización de un proceso administrado cuantitativamente

PG 4.1 Establecimiento de objetivos cuantitativos para el proceso

PG 4.2 Estabilización de desempeño de subprocesos

MG 5 Institucionalización de un proceso de optimización

PG 5.1 Aseguramiento del mejoramiento de proceso continuo

PG 5.2 Corrección de causas originales de problemas

El modelo CMMI por etapas define las mismas áreas de proceso, metas y prácticas que el modelo continuo. La diferencia principal es que el modelo por etapas define cinco niveles de madurez, en lugar de cinco niveles de capacidad. Para lograr un nivel de madurez deben lograrse las metas y prácticas específicas asociadas con un conjunto de áreas de proceso. La relación entre niveles de madurez y áreas de proceso se muestra en la figura 30.3.

INFORMACIÓN

La CMMI: ¿Se debe o no se debe?

La CMMI es un metamodelo de proceso. Define (en más de 700 páginas) las características de proceso que deben existir si una organización quiere establecer un proceso de software que sea completo. La pregunta que se ha debatido durante más de una década es: ¿es excesiva la CMMI? Como la mayoría de las cosas en la vida (y en el software), la respuesta no es un simple "sí" o "no".

El espíritu de la CMMI siempre debe adoptarse. Con riesgo de sobresimplifiación, se argumenta que el desarrollo del software debe tomarse con seriedad: planificarse a profundidad, controlarse de manera uniforme, rastrearse con precisión y llevarse a cabo con profesionalismo. Debe enfocarse en las necesidades de los participantes en el proyecto, en las habilidades de los ingenieros del software y en la calidad del producto final. Nadie estaría en desacuerdo con estas ideas.

Los requisitos detallados de la CMMI deben considerarse seriamente si una organización construye grandes sistemas complejos que involucran decenas o cientos de personas a lo largo de varios meses o años. Puede ser que la CMMI sea "correcta" en tales situaciones si la cultura de la organización es sensible a modelos de proceso estándar y si la administración se compromete para convertirlo en éxito. Sin embargo, en otras situaciones, la CMMI simplemente puede ser demasiado para que una organización lo asimile con éxito. ¿Esto significa que la CMMI es "mala" o "excesivamente burocrática" o "anticuada"? No..., no lo es. Simplemente significa que lo que es correcto para una cultura organizacional puede no serlo para otra.

La CMMI es un logro significativo en ingeniería del software. Proporciona un análisis amplio de las actividades y acciones que deben presentarse cuando una organización construye software de computadora. Incluso si una organización de software elige no adoptar sus detalles, todo equipo de software debe abrazar su espíritu y ganar comprensión de su análisis del proceso y de la práctica de la ingeniería del software.

30.4 El CMM de personal

El CMM de personal sugiere prácticas que mejoran la competencia y cultura de la fuerza laboral.

Un proceso de software, sin importar cuán bien se conciba, no triunfará sin personal de software talentoso y motivado. El *Modelo de Madurez de Capacidad de Personal* "es un mapa de caminos para implementar prácticas que mejoran de manera continua la capacidad de la fuerza de trabajo de una organización" [Cur02]. Desarrollado a mediado de los años de 1990 y refinado durante los años siguientes, la meta del CMM de personal es alentar el mejoramiento continuo del conocimiento de la fuerza laboral genérica (llamadas "competencias centrales"), de las habilidades específicas de los ingenieros del software y de la administración del proyecto (llamadas "competencias de la fuerza laboral") y de las habilidades relacionadas con el proceso.

Como el CMM, la CMMI y los marcos conceptuales MPS relacionados, el CMM de personal define un conjunto de cinco niveles de madurez organizativa que proporcionan un indicio de la sofisticación relativa de las prácticas y procesos de la fuerza laboral. Dichos niveles de madurez

FIGURA 30.3
Áreas de proceso requeridas para lograr un nivel de madurez
Fuente: [Phi02].

Nivel	Enfoque	Áreas de proceso
Optimización	***Mejora de proceso continua***	**Innovación y despliegue organizacional** **Análisis causal y resolución**
Administrado cuantitativamente	***Administración cuantitativa***	**Desempeño de proceso organizacional** **Administración de proyecto cuantitativa**
Definido	***Estandarización de proceso***	**Desarrollo de requisitos** **Solución técnica** **Integración de producto** **Verificación** **Validación** **Enfoque en proceso organizacional** **Definición de proceso organizacional** **Capacitación organizacional** **Administración de proyecto integrada** **Administración de proveedor integrada** **Gestión del riesgo** **Análisis de decisión y resolución** **Entorno organizacional para integración** **Formación de equipo integrado**
Administrado	***Administración básica de proyecto***	**Gestión de requisitos** **Planificación de proyecto** **Monitoreo y control del proyecto** **Administración de acuerdo con proveedor** **Medición y análisis** **Aseguramiento de calidad de proceso y producto** **Administración de configuración**
Realizado		

[CMM08] se ligan a la existencia (dentro de una organización) de un conjunto de áreas de proceso clave (APC). En la figura 30.4 se muestra un panorama de los niveles organizativos y las APC relacionadas.

El CMM de personal complementa cualquier marco conceptual MPS al alentar a una organización a nutrir y mejorar su activo más importante: su personal. Tan importante, que establece una atmósfera de fuerza de trabajo que permite a una organización de software "atraer, desarrollar y conservar talento sobresaliente" [CMM08].

30.5 OTROS MARCOS CONCEPTUALES MPS

Aunque los CMM y CMMI del SEI son los marcos conceptuales MPS de mayor aplicación, se han propuesto algunas alternativas[7] que están en uso. Entre las más ampliamente utilizadas se encuentran:

- **SPICE:** una iniciativa internacional para dar apoyo a la valoración de proceso ISO y a estándares de proceso de ciclo de vida [SPI99]
- **ISO/IEC 15504** para Valoración de Proceso (de Software) [ISO08]
- **Bootstrap:** un marco conceptual MPS para organizaciones pequeñas y medianas que se adecua a SPICE [Boo06]

7 Es razonable argumentar que algunos de estos marcos conceptuales no son tanto "alternativas" como enfoques complementarios del MPS. Una tabla exhaustiva de muchos más marcos conceptuales MPS puede encontrarse en **www.geocities.com/lbu_measure/spi/spi.htm#p2**

FIGURA 30.4
Áreas de proceso para el CMM de personal

Nivel	Enfoque	Áreas de proceso
Optimizado	***Mejoramiento continuo***	**Innovación continua de la fuerza laboral** **Alineación del desempeño organizativo** **Mejoramiento de capacidad continuo**
Predecible	***Cuantifica y gestiona conocimiento, capacidades y habilidades***	**Tutelaje** **Administración de capacidad organizativa** **Administración de desempeño cuantitativo** **Activos basados en competencia** **Grupos de trabajo fortalecidos** **Integración de competencia**
Definido	***Identifica y desarrolla conocimiento, capacidades y habilidades***	**Cultura de participación** **Desarrollo de grupos de trabajo** **Prácticas basadas en competencia** **Desarrollo profesional** **Desarrollo de competencias** **Planificación de fuerza de trabajo** **Análisis de competencia**
Administrado	***Prácticas de administración de personal básicas, repetibles***	**Compensación** **Capacitación y desarrollo** **Administración del desempeño** **Entorno laboral** **Comunicación y coordinación** **Dotación de personal**
Inicial	***Prácticas inconsistentes***	

- **PSP y TSP:** marcos conceptuales MPS individuales y específicos de equipo ([Hum97], [Hum00]) que se enfocan en procesos micro, un enfoque más riguroso del desarrollo de software acoplado con medición
- **TickIT:** método de auditoría [Tic05] que valora el cumplimiento de una organización al Estándar ISO 9001:2000

En los siguientes párrafos se presenta un breve panorama de cada uno de estos marcos conceptuales MPS. Si tiene más interés está disponible una gran variedad de recursos tanto impresos como en la web.

? Además del CMM, ¿existen otros marcos conceptuales MPS que puedan considerarse?

SPICE. El modelo SPICE (*Software Process Improvement and Capability dEtermination*: determinación de mejoramiento y capacidad del proceso de software) proporciona un marco conceptual de valoración MPS que cumple con ISO 15504:2003 e ISO 12207. La suite de documentos SPICE [SDS08] presenta un marco conceptual MPS completo, que incluye un modelo para gestión de proceso, lineamientos para realizar una valoración y clasificación del proceso bajo consideración, construcción, selección y uso de instrumentos y herramientas de valoración y capacitación para asesores.

Bootstrap. El marco conceptual MPS *Bootstrap* "se desarrolló para asegurar conformidad con el estándar ISO emergente para valoración y mejoramiento del proceso de software (SPICE) y para alinear la metodología con ISO 12207" [Boo06]. El objetivo de Bootstrap es evaluar un proceso de software, usando un conjunto de mejores prácticas de ingeniería del software como base para la valoración. Como el CMMI, Bootstrap proporciona un nivel de madurez de proceso, empleando los resultados de cuestionarios que recopilan información acerca del proceso de

software "como es" y proyectos de software. Los lineamientos MPS se basan en nivel de madurez y metas organizativas.

PSP y TSP. Aunque MPS generalmente se caracteriza como una actividad organizativa, no hay razón por la que el mejoramiento del proceso no pueda realizarse en un nivel individual o de equipo. Tanto PSP como TSP (capítulo 2) enfatizan la necesidad de recopilar datos continuamente acerca del trabajo que se realiza y de usar dichos datos para desarrollar estrategias para su mejoramiento. Watts Humphrey [Hum97], el desarrollador de ambos métodos, comenta:

> El PSP [y TSP] le mostrará cómo planificar y rastrear su trabajo y cómo producir software de alta calidad de manera consistente. Al usar PSP [y TSP], contará con los datos que muestran la efectividad de su trabajo y que identifican sus fortalezas y debilidades [...] Para tener una carrera exitosa y gratificante, necesita conocer sus capacidades y habilidades, luchar por mejorarlas y capitalizar sus talentos únicos en el trabajo que realice.

Cita:

"Las organizaciones de software muestran limitaciones significativas en su habilidad para capitalizar las experiencias obtenidas de los proyectos completados."

NASA

TickIT. El método de auditoría Ticket garantiza cumplimiento con ISO 9001:200 para software: un estándar genérico que se aplica a cualquier organización que quiera mejorar la calidad global de los productos, sistemas o servicios que ofrece. Por tanto, el estándar es directamente aplicable a organizaciones y compañías de software.

La estrategia subyacente sugerida por ISO 9001:2000 se describe de la forma siguiente [ISO01]:

> ISO 9001:2000 subraya la importancia para una organización de identificar, implementar, gestionar y mejorar continuamente la efectividad de los procesos que son necesarios para el sistema de gestión de la calidad, y de administrar las interacciones de dichos procesos con la finalidad de lograr los objetivos de la organización [...] La efectividad y eficiencia del proceso pueden valorarse a través de procesos de revisión interna o externa y evaluarse sobre una escala de madurez.

WebRef

Un excelente resumen de ISO 9001:2000 puede encontrarse en **http://praxiom.com/iso-9001.htm**

ISO 9001:2000 adoptó un ciclo de "planificar-hacer-verificar-actuar" que se aplica a los elementos de gestión de la calidad de un proyecto de software. Dentro de un contexto de software, "planificar" establece los objetivos, actividades y tareas del proceso, necesarios para lograr software de alta calidad y la resultante satisfacción del cliente. "Hacer" implementa el proceso de software (incluidas tanto actividades de marco conceptual como sombrilla). La "verificación" monitorea y mide el proceso para asegurar que se lograron todos los requisitos establecidos para gestión de la calidad. Al "actuar", se inician las actividades de mejoramiento del proceso de software que trabajan continuamente para mejorar el proceso. TickIt puede usarse a través del ciclo "planificar-hacer-verificar-actuar" a fin de garantizar que el proceso MPS avanza. Los auditores TickIT valoran la aplicación del ciclo como un precursor de la certificación ISO 9001:2000. Para un análisis detallado de ISO 9001:2000 y TickIT debe examinar [Ant06], [Tri05] o [Sch03].

30.6 Rendimiento sobre inversión de MPS

El MPS representa un trabajo duro y requiere inversión sustancial de dinero y de personal. Los administradores que aprueben el presupuesto y los recursos para MPS invariablemente plantearán la pregunta: ¿cómo sé que lograremos un rendimiento razonable por el dinero que gastemos?

Cualitativamente, quienes impulsan MPS arguyen que un proceso de software mejorado conducirá a calidad de software mejorada. Afirman que el proceso mejorado dará como resultado la implementación de mejores filtros de calidad (lo que arrojará menos defectos propagados), mejor control de cambio (que da como resultado menos caos de proyecto) y menos reelaboración técnica (lo que desemboca en menor costo y mejor tiempo de llegada al mercado).

¿Pero estos beneficios cualitativos pueden traducirse en resultados cuantitativos? La ecuación clásica de rendimiento sobre inversión (RSI) es:

$$\text{RSI} = \left[\frac{\Sigma(\textit{beneficios}) - \Sigma(\textit{costos})}{\Sigma(\textit{costos})}\right] \times 100\%$$

donde

Los *beneficios* incluyen los ahorros en costo asociados con productos de mayor calidad (menos defectos), menos reelaboración, esfuerzo reducido asociado con cambios e ingreso que se acumula por menor tiempo de llegada al mercado.

Los *costos* incluyen tanto costos MPS directos (por ejemplo, capacitación, medición) como costos indirectos asociados con un mayor énfasis en el control de la calidad y en las actividades de gestión del cambio, así como por la aplicación más rigurosa de los métodos de ingeniería del software (por ejemplo, la creación de un modelo de diseño).

En el mundo real, dichas cantidades de beneficios y costos en ocasiones son difíciles de medir con precisión y están abiertos a interpretación. Pero ello no significa que una organización de software debe realizar un programa MPS sin análisis cuidadoso de los costos y beneficios que acumula. Un tratamiento amplio de RSI para MPS puede encontrarse en un libro inigualable de David Rico [Ric04].

30.7 Tendencias MPS

Durante las dos décadas pasadas, muchas compañías intentaron mejorar sus prácticas de ingeniería del software al aplicar un marco conceptual MPS para efectuar cambio organizacional y transición tecnológica. Como se observó anteriormente, en este capítulo, más de la mitad fracasan en esta labor. Sin importar el éxito o el fracaso, todos gastan una cantidad significativa de dinero. David Rico [Ric04] reporta que una aplicación típica de un marco conceptual MPS como el CMM SEI, ¡puede costar entre $25 000 y $70 000 por persona y tardar años en completarse! No debe sorprenderle que el futuro de MPS deba enfatizar un enfoque menos costoso y consumidor de tiempo.

Para ser efectivo en el mundo de desarrollo del software del siglo XXI, los futuros marcos conceptuales MPS deben volverse significativamente más ágiles. En lugar de un enfoque centrado en la organización (que puede tardar años en completarse exitosamente), los esfuerzos MPS contemporáneos deben enfocarse en el nivel del proyecto y trabajar para mejorar un proceso de equipo en semanas, no meses ni años. Para lograr resultados significativos (incluso en el nivel del proyecto) en un tiempo corto, los modelos de marco conceptual complejos pueden dar lugar a modelos más simples. En lugar de decenas de prácticas clave y cientos de prácticas complementarias, un marco conceptual MPS ágil enfatiza solamente algunas prácticas esenciales (por ejemplo, análogas a las actividades de marco conceptual estudiadas a lo largo de este libro).

Cualquier intento de MPS demanda una fuerza laboral conocedora, pero los gastos de educación y capacitación pueden ser onerosos y deben minimizarse (y simplificarse). En lugar de cursos en aulas (costosos y consumidores de tiempo), los esfuerzos MPS futuros deben apoyarse en capacitación basada en web que se dirija a prácticas esenciales. En lugar de intentos a largo plazo por cambiar la cultura de la organización (con todos los peligros políticos que conllevan), el cambio cultural debe ocurrir como se hace en el mundo real: un pequeño grupo a la vez hasta alcanzar un punto de inflexión.

El trabajo MPS de las dos décadas anteriores tiene mérito significativo. Los marcos conceptuales y los modelos que se desarrollaron representan activos intelectuales sustanciales para la

comunidad de la ingeniería del software. Pero, como todas las cosas, dichos activos guían los intentos futuros de MPS no al convertirse en un dogma recurrente, sino al funcionar como la base para modelos MPS mejores, más simples y más ágiles.

30.8 Resumen

Un marco conceptual de mejoramiento del proceso de software define las características que debe presentar si debe lograrse un proceso de software efectivo, un método de valoración que ayuda a determinar si dichas características están presentes y una estrategia para auxiliar a una organización de software a implementar dichas características de proceso que se encuentren debilitadas o que falten. Sin importar los grupos que defienden el MPS, la meta es mejorar la calidad del proceso y, en consecuencia, la calidad y la puntualidad en la entrega del software.

Un modelo de madurez de proceso proporciona un indicio global de la "madurez del proceso" que muestra una organización del software. Asimismo, proporciona un sentimiento cualitativo sobre la efectividad relativa del proceso de software que se usa actualmente.

El mapa de caminos MPS comienza con la valoración, una serie de actividades de evaluación que descubren tanto fortalezas como debilidades en la forma en que la organización aplica el proceso de software existente y las prácticas de ingeniería del software que pueblan el proceso. Como consecuencia de la valoración, una organización de software puede desarrollar un plan MPS global.

Unos de los elementos clave de cualquier plan MPS son la educación y la capacitación, actividades que se enfocan en mejorar el nivel de conocimiento de administradores y profesionales. Una vez que el personal está versado en las tecnologías de software actuales, comienza la selección y la justificación. Dichas tareas conducen a elegir la arquitectura del proceso de software, los métodos que lo pueblan y las herramientas que soporta. Instalación y evaluación son actividades MPS que ejemplifican los cambios del proceso y que valoran su eficacia e impacto.

Para mejorar exitosamente su proceso de software, una organización debe mostrar las siguientes características: compromiso y apoyo de los administradores para el MPS, involucramiento del personal a lo largo del proceso MPS, integración del proceso en la cultura organizacional global, una estrategia MPS que se haya hecho a la medida de las necesidades locales y administración sólida del proyecto MPS.

En la actualidad se usan algunos marcos conceptuales MPS. Los CMM y CMMI de SEI se usan ampliamente. El CMM de personal se particulariza para valorar la calidad de la cultura de la organización y del personal que la puebla. SPICE, Bootstrap, PSP, TSP y TickIT son marcos conceptuales adicionales que pueden conducir a MPS efectivo.

El MPS representa un trabajo duro que requiere inversión sustancial de dinero y de personal. Para garantizar que se logre un rendimiento razonable sobre la inversión, una organización debe medir los costos asociados con el MPS y los beneficios que pueden atribuírsele de manera directa.

Problemas y puntos por evaluar

30.1. ¿Por qué las organizaciones de software con frecuencia luchan cuando se embarcan en un esfuerzo por mejorar el proceso de software local?

30.2. Con sus palabras, describa el concepto de "madurez de proceso".

30.3. Realice una investigación (verifique el sitio web SEI) y determine la distribución de madurez de proceso para organizaciones de software en Estados Unidos y el mundo.

30.4. Usted trabaja para una organización de software muy pequeña: sólo 11 personas se involucran en el desarrollo del software. ¿MPS es para usted? Explique su respuesta.

30.5. La valoración es análoga a un examen médico anual. Con un examen médico como metáfora, describa la actividad de valoración MPS.

30.6. ¿Cuál es la diferencia entre un proceso "como es", un proceso "de aquí a allá" y un proceso "por ser"?

30.7. ¿Cómo se aplica la gestión del riesgo dentro del contexto de MPS?

30.8. Seleccione uno de los factores cruciales de éxito anotados en la sección 30.2.7. Realice investigación y escriba un breve ensayo acerca de cómo puede lograrse.

30.9. Realice investigación y explique cómo difiere la CMMI de su predecesor, el CMM.

30.10. Seleccione uno de los marcos conceptuales MPS que estudió en la sección 30.5 y escriba un breve ensayo que lo describa con más detalle.

Lecturas y fuentes de información adicionales

Uno de los recursos de información más exhaustivo y de más fácil acceso acerca de MPS lo desarrolló el Software Engineering Institute y está disponible en **www.sei.cmu.edu**. El sitio web de SEI contiene cientos de artículos, estudios y descripciones detalladas acerca de marco conceptual MPS.

Durante los años recientes se han agregado algunos libros valiosos a una amplia literatura desarrollada durante las dos décadas anteriores. Land (*Jumpstart CMM/CMMI Software Process Improvements*, Wiley-IEEE Computer Society, 2007) fusiona los requisitos definidos como parte de los CMM e CMMI de SEI con los estándares de ingeniería de software del IEEE, con énfasis en la intersección de proceso y práctica. Mutafelija y Stromberg (*Systematic Process Improvement Using ISO 9001:2000 and CMMI*, Artech House Publishers, 2007) estudian los marcos conceptuales MPS de ISO 9001:2000 e CMMI, y su "sinergia". Conradi *et al.* (*Software Process Improvement: Results and Experiencia from the Field*, Springer, 2006) presentan los resultados de una serie de estudios de caso y experimentos relacionados con MPS. Van Loon (*Process Assessment and Improvement: A Practical Guide for Managers, Quality Professionals and Assessors*, Springer, 2006) estudia el MPS dentro del contexto de ISO/IEC 15504. Watts Humphrey (*PSP*, Addison-Wesley, 2005, y *TSP*, Addison-Wesley, 2005) aborda su marco conceptual MPS de proceso de personal de equipo y su marco conceptual MPS de proceso de equipo de software en dos libros separados. Fantina (*Practical Software Process Improvement*, Artech House Publishers, 2004) brinda lineamiento pragmático con énfasis en CMMI/CMM.

En internet, está disponible una gran variedad de fuentes de información acerca del mejoramiento del proceso de software. Una lista actualizada de referencias existentes en la World Wide Web que son relevantes para MPS puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

CAPÍTULO

31

TENDENCIAS EMERGENTES EN INGENIERÍA DEL SOFTWARE

A lo largo de la relativamente breve historia de la ingeniería del software, los profesionales e investigadores desarrollaron una colección de modelos de proceso, métodos técnicos y herramientas automatizadas con la intención de fomentar el cambio fundamental en la forma de construir el software de computadoras. Aunque las experiencias anteriores indican otra cosa, existe un deseo tácito por encontrar la panacea: el proceso mágico o la tecnología trascendente que permitirá construir con facilidad grandes y complejos sistemas basados en software, sin confusión, sin errores, sin demora, sin los muchos problemas que todavía plagan el trabajo de software.

Pero la historia indica que la búsqueda de la panacea parece condenada al fracaso. Las nuevas tecnologías se introducen regularmente, publicitadas con exceso como una "solución" a muchos de los problemas que enfrentan los ingenieros del software y se incorporan en los proyectos, grandes y pequeños. Los expertos de la industria resaltan la importancia de estas "nuevas" tecnologías de software, los conocedores de la comunidad del software las adoptan con entusiasmo y, a final de cuentas, tienen un papel en el mundo de la ingeniería del software. Pero tienden a no cumplir su promesa y, como consecuencia, la búsqueda continúa.

Mili y Cowan [Mil00b] comentan acerca de los retos que se afrontan cuando se intenta aislar tendencias tecnológicas significativas:

> ***¿Qué factores determinan el éxito de una tendencia?*** ¿Qué caracteriza el éxito de las tendencias tecnológicas? ¿Su mérito técnico? ¿Su habilidad para abrir nuevos mercados? ¿Su pericia para alterar la economía de los mercados existentes?
>
> ***¿Qué ciclo de vida sigue una tendencia?*** Mientras que la visión tradicional es que las tendencias evolucionan a lo largo de un ciclo de vida predecible bien definido, que avanza de una idea de investigación a un producto terminado a través de un proceso de transferencia, se descubre que muchas tendencias actuales provocaron corto circuito en este ciclo o siguieron otro.
>
> ***¿Con cuánta anticipación puede identificarse una tendencia exitosa?*** Si se sabe cómo identificar los factores de éxito y/o se entiende el ciclo de vida de una tendencia, entonces se busca

UNA MIRADA RÁPIDA

¿Qué es? Nadie puede predecir el futuro con absoluta certeza. Pero es posible valorar las tendencias en el área de la ingeniería del software y, de dichas tendencias, sugerir posibles direcciones para la tecnología. Eso es lo que se intenta hacer en este capítulo.

¿Quién lo hace? Quienquiera que desee emplear el tiempo para situarse frente a los conflictos de la ingeniería del software puede intentar predecir la futura dirección de la tecnología.

¿Por qué es importante? ¿Por qué los antiguos reyes contrataban adivinos? ¿Por qué las grandes corporaciones multinacionales contratan firmas consultoras y a expertos para preparar pronósticos? ¿Por qué un sustancial porcentaje del público lee horóscopos? Todos ellos quieren conocer lo que está por venir para estar preparados.

¿Cuáles son los pasos? No hay una fórmula para predecir el futuro. Se intenta hacer esto al recopilar datos, organizarlos para proporcionar información útil, examinar asociaciones sutiles para extraer conocimiento y, a partir de este conocimiento, sugerir probables tendencias que predigan cómo serán las cosas en algún tiempo por venir.

¿Cuál es el producto final? Una visión del futuro cercano, que puede o no ser correcta.

¿Cómo me aseguro de que lo hice bien? Predecir el camino que está adelante es un arte, no una ciencia. De hecho, es muy raro cuando una predicción seria acerca del futuro es absolutamente correcta o inequívocamente errónea (con excepción, afortunadamente, de las predicciones del fin del mundo). Se observan las tendencias y se intenta extrapolarlas. Lo acertado de la extrapolación sólo puede valorarse conforme pasa el tiempo.

¿Cuáles son las "grandes preguntas" cuando se considera la evolución tecnológica?

identificar signos tempranos del éxito de una tendencia. De manera retórica se busca la habilidad de reconocer la siguiente tendencia antes que todos los demás.

¿Qué aspectos de la evolución son controlables? ¿Las corporaciones pueden usar su influencia en el mercado para imponer tendencias? ¿Qué papel juegan los estándares en la definición de tendencias? El análisis cuidadoso de Ada contra Java, por ejemplo, debe ser iluminador a este respecto.

No existen respuestas sencillas a estas preguntas, y puede no haber discusión acerca de que intentos anteriores por identificar tecnologías significativas fueron mediocres, cuando mucho.

En ediciones anteriores de este libro (durante los 30 años anteriores) se analizaron las tecnologías emergentes y su impacto proyectado sobre la ingeniería del software. Algunas se han adoptado ampliamente, pero otras nunca alcanzaron su potencial. La conclusión del autor es que las tecnologías vienen y van; las tendencias reales que se exploran son las más blandas. Por esto se entiende que el progreso en la ingeniería del software se guiará por las tendencias empresariales, organizativas, de mercado y culturales. Dichas tendencias conducen a innovación tecnológica.

En este capítulo se observarán algunas tendencias tecnológicas de la ingeniería del software, pero el énfasis principal se colocará en algunas tendencias empresariales, organizativas, de mercado y culturales que pueden tener una importante influencia sobre la tecnología de la ingeniería del software durante los próximos 10 o 20 años.

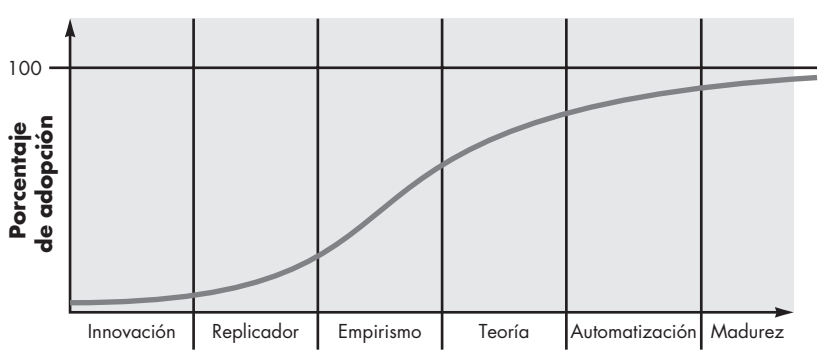
31.1 Evolución tecnológica

En un libro fascinante que ofrece una atractiva mirada sobre la evolución de las tecnologías de computación (y otras relacionadas), Ray Kurzweil [Kur05] argumenta que la evolución tecnológica es similar a la evolución biológica, pero que ocurre a un ritmo más rápido. La evolución (biológica o tecnológica) ocurre como resultado de realimentación positiva: "los métodos más capaces que resultan de una etapa del avance evolutivo se usan para crear la siguiente etapa" [Kur06].

Las grandes preguntas para el siglo XXI son: 1) ¿cuán rápidamente evoluciona la tecnología? 2) ¿cuán significativos son los efectos de la realimentación positiva? 3) ¿cuán profundos serán los cambios resultantes?

Cuando se introduce una nueva tecnología exitosa, el concepto inicial se mueve a través de un "ciclo de vida de innovación" [Gai95] razonablemente predecible, que se ilustra en la figura 31.1. En la fase de *innovación* se reconoce un problema y se realizan intentos repetidos para encontrar una solución viable. En algún punto, una solución se muestra como prometedora. El trabajo de innovación inicial se reproduce en la fase de *replicador* y obtiene un uso más amplio. El *empirismo* conduce a la creación de reglas empíricas que gobiernan el uso de la tecnología y su éxito repetido conduce a una *teoría* más amplia que da paso a la creación de herramientas

Figura 31.1
Ciclo de vida de una innovación tecnológica



Cita:

"Las predicciones son muy difíciles de hacer, en especial cuando tratan acerca del futuro."

Mark Twain

automatizadas durante la fase de *automatización*. Finalmente, la tecnología madura y se usa ampliamente.

El lector observará que muchas tendencias de investigación y tecnológicas nunca llegan a la madurez. De hecho, la gran mayoría de las tecnologías "prometedoras" en el dominio de la ingeniería del software reciben amplio interés durante algunos años y luego caen en un nicho de uso por parte de un grupo fiel de adherentes. Esto no quiere decir que dichas tecnologías carecen de mérito, sino más bien busca enfatizar que el viaje a través del ciclo de vida de la innovación es largo y duro.

PUNTO CLAVE

La tecnología de computación evoluciona a una tasa exponencial y su crecimiento pronto puede volverse explosivo.

Kurzweil [Kur05] está de acuerdo en que las tecnologías de computación evolucionan a través de una "curva S" que muestra crecimiento relativamente lento durante los años formativos de la tecnología, rápida aceleración durante su periodo de crecimiento y luego un periodo de nivelación conforme la tecnología llega a sus límites. Pero la computación y otras tecnologías relacionadas muestran crecimiento explosivo (exponencial) durante las etapas centrales que se muestran en la figura 31.1 y continuarán haciéndolo. Además, conforme una curva S termina, otra la sustituye con crecimiento incluso más explosivo durante su periodo de crecimiento.[1] En la actualidad, estamos en la rodilla de la curva S para las modernas tecnologías de computación, en la transición entre el crecimiento temprano y el crecimiento explosivo que sigue. La implicación es que, durante los próximos 20 o 40 años, se verán cambios dramáticos (incluso enloquecedores) en la capacidad de computación. Las décadas por venir darán como resultado cambios en la rapidez, tamaño, capacidad y consumo de energía de cómputo (por mencionar sólo algunas características).

Kurzweil [Kur05] sugiere que, dentro de 20 años, la evolución tecnológica acelerará a un ritmo cada vez más rápido, lo que a final de cuentas conducirá a una era de inteligencia no biológica que se fusionará con y extenderá la inteligencia humana en formas que son fascinantes de imaginar.

Y todo esto, sin importar cómo evolucione, requerirá software y sistemas que harán que, en comparación, los esfuerzos actuales parezcan infantiles. Hacia el año 2040, una combinación de computación extrema, nanotecnología, redes ubicuas con ancho de banda masivamente elevado, y robótica conducirán a un mundo diferente.[2] El software, posiblemente en formas que todavía no pueden comprenderse, continuará residiendo en el centro de este nuevo mundo. La ingeniería del software no se irá.

31.2 Observación de las tendencias en ingeniería del software

Cita:

"Creo que hay un mercado mundial para acaso cinco computadoras."

Thomas Watson, ejecutivo de IBM, 1943

La sección 31.1 consideró brevemente las fascinantes posibilidades que pueden acumularse a partir de tendencias a largo plazo en computación y tecnologías relacionadas. Pero ¿y en el corto plazo?

Barry Boehm [Boe08] sugiere que "los ingenieros del software enfrentarán los, con frecuencia, formidables desafíos de lidiar con rápidos cambios, incertidumbre y emergencia, dependencia, diversidad e interdependencia, pero que también tendrán oportunidades de realizar significativas aportaciones que harán la diferencia a fin de mejorar". Pero, ¿cuáles son las tendencias que le permitirán enfrentar dichos desafíos en los años por venir?

---

1 Por ejemplo, los límites de los circuitos integrados pueden alcanzarse dentro de la siguiente década, pero dicha tecnología puede sustituirse por tecnologías de cómputo molecular y vendrá otra curva S acelerada.

2 Kurzweil [Kur05] presenta un argumento técnico razonado que predice una fuerte inteligencia artificial (que pasará la prueba de Turing) hacia 2029 y sugiere que la evolución de los humanos y las máquinas comenzará a fusionarse hacia 2045. La gran mayoría de los lectores de este libro vivirán para ver si esto, en realidad, llega a suceder.

En la introducción a este capítulo se señaló que las "tendencias blandas" tienen un impacto significativo sobre la dirección global de la ingeniería del software. Pero otras tendencias ("más duras") orientadas a la investigación y la tecnología siguen siendo importantes. Las tendencias en investigación "están impulsadas por las percepciones generales del estado del arte y de la práctica, por percepciones del investigador acerca de las necesidades de los profesionales, por programas de financiamiento nacional que apresuran las metas estratégicas específicas y por mero interés técnico" [Mil00a]. Las tendencias tecnológicas ocurren cuando las tendencias en investigación se extrapolan para satisfacer necesidades industriales y cuando se les da forma de acuerdo con las demandas que impulsa el mercado.

PUNTO CLAVE

El "ciclo de promoción excesiva" presenta una visión realista de la integración tecnológica a corto plazo. Sin embargo, la tendencia a largo plazo es exponencial.

En la sección 31.1 se estudia el modelo de curva S para evolución tecnológica. La curva S es adecuada para considerar los efectos a largo plazo de las tecnologías centrales conforme evolucionan. ¿Pero qué hay acerca de las innovaciones, herramientas y métodos más modestos para el corto plazo? El Gartner Group [Gar08], un grupo consultor que estudia las tendencias tecnológicas a través de muchas industrias, desarrolló un *ciclo de promoción excesiva para tecnologías emergentes*, que se representa en la figura 31.2. El ciclo del Gartner Group muestra cinco fases:

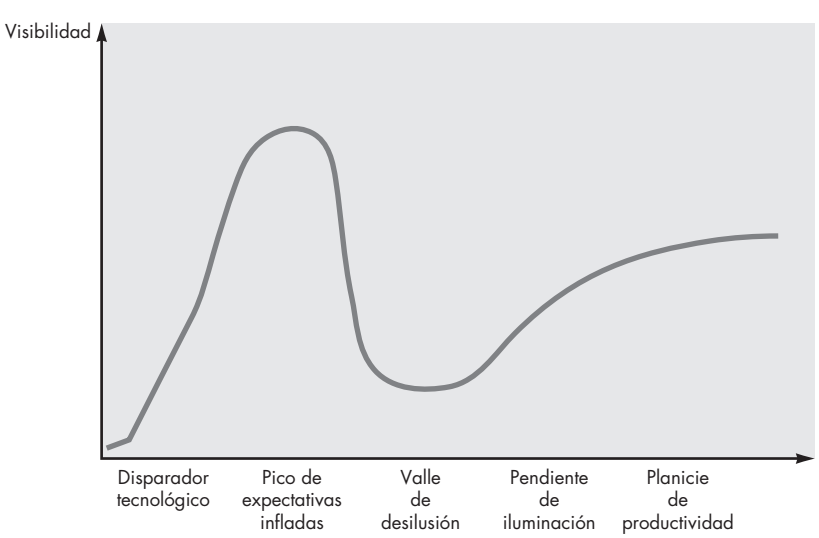
- *Disparador tecnológico:* un hallazgo de investigación o lanzamiento de un nuevo producto innovador que conduce a cobertura de los medios y a entusiasmo del público.
- *Pico de expectativas infladas:* entusiasmo exagerado y proyecciones demasiado optimistas del impacto con base en éxitos limitados, pero bien publicitados.
- *Desilusión:* las proyecciones de impacto demasiado optimistas no se satisfacen y los críticos comienzan a presionar; la tecnología pasa de moda entre los conocedores.
- *Pendiente de iluminación:* el uso creciente mediante una amplia variedad de compañías conduce a una mejor comprensión del verdadero potencial de la tecnología; surgen métodos y herramientas comerciales para apoyar la tecnología.
- *Planicie de productividad:* los beneficios en el mundo real ahora son obvios y el uso penetra en un significativo porcentaje del mercado potencial.

No toda la tecnología de ingeniería del software logra pasar a través del ciclo de promoción excesiva. En algunos casos, la desilusión se justifica y la tecnología se relega a la oscuridad.

FIGURA 31.2

Ciclo de promoción excesiva del Gartner Group para tecnologías emergentes

Fuente: [Gar08].



31.3 IDENTIFICACIÓN DE "TENDENCIAS BLANDAS"

Cita:

"640K deben ser suficientes para cualquiera."

Bill Gates, presidente de Microsoft, 1981

Cada nación con una sustancial industria de TI tiene un conjunto de características únicas que definen la forma en la que se dirigen los negocios, la dinámica organizacional que surge dentro de una compañía, los distintos conflictos de mercadeo que se aplica a los clientes locales y la decisiva cultura que dicta toda interacción humana. Sin embargo, algunas tendencias en cada una de dichas áreas son universales y tienen mucho que ver con sociología, antropología y psicología de grupos (que con frecuencia se conocen como "ciencias blandas"), como tienen que ver con la investigación académica o industrial.

Conectividad y colaboración (habilitadas por comunicación con alto ancho de banda) ya condujeron a equipos de software que no ocupan el mismo espacio físico (trabajo a distancia y empleo de tiempo parcial en un contexto local). Un equipo colabora con otros equipos que están separados por zonas horarias, lenguaje nativo y cultura. La ingeniería del software debe responder con un modelo de proceso que abarque a "equipos distribuidos" y que sea suficientemente ágil para satisfacer las demandas de inmediatez, pero suficientemente disciplinado para coordinar grupos dispares.

? ¿Qué tendencias blandas impactarán las tecnologías relacionadas con la ingeniería del software?

La *globalización* conduce a una fuerza de trabajo diversa (en idioma, cultura, resolución de problemas, filosofía administrativa, prioridades de comunicación e interacción persona a persona). Esto, a su vez, demanda una estructura organizativa flexible. Diferentes equipos (en distintos países) deben responder a los problemas de ingeniería de manera que se acomode mejor a sus necesidades únicas y, al mismo tiempo, fomentar un nivel de uniformidad que permita el avance de un proyecto global. Este tipo de organización sugiere menos niveles de administración y un mayor énfasis en la toma de decisiones por parte de cada equipo. Puede conducir a mayor agilidad, pero sólo si los mecanismos de comunicación se establecen de modo que todo equipo pueda entender el proyecto y su estado técnico (vía groupware en red) en cualquier momento. Los métodos y herramientas de ingeniería del software pueden ayudar a lograr cierto nivel de uniformidad (equipos que hablan el mismo "idioma", implementado a través de métodos y herramientas específicos). El proceso de software puede proporcionar el marco conceptual para la ejemplificación de estos métodos y herramientas.

En algunas regiones del planeta (Estados Unidos y Europa son ejemplos), la población está envejeciendo. Esta demografía innegable (y tendencia cultural) implica que muchos ingenieros y gerentes de software experimentados dejarán el campo de trabajo en la próxima década. La comunidad de la ingeniería del software debe responder con mecanismos viables que capturen el conocimiento de estos gerentes y técnicos que envejecen [por ejemplo, el uso de *patrones* (capítulo 12) es un paso en la dirección correcta], de modo que quede disponible para generaciones futuras de trabajadores del software. En otras regiones del mundo se multiplica el número de jóvenes disponibles para la industria del software. Esto proporciona una oportunidad para moldear una cultura de ingeniería del software sin la carga de 50 años de prejuicios "de la vieja escuela".

Se estima que más de mil millones de nuevos consumidores entrarán al mercado laboral mundial en la próxima década. Los consumidores que gastan en "economías emergentes superarán por mucho los US$9 billones" [PET06]. Hay poca duda de que un porcentaje no trivial de este gasto se aplicará a productos y servicios que tengan un componente digital, que se basen en software o se impulsen mediante él. La implicación: una creciente demanda para nuevo software. Entonces la pregunta es: ¿pueden desarrollarse nuevas tecnologías de ingeniería del software para satisfacer esta demanda mundial? Las modernas tendencias de mercado con frecuencia se impulsan mediante el lado suministro.[3] En otros casos, los requerimientos del lado

3 El lado suministro adopta un enfoque de mercado "construye y ellos vendrán". Se crean tecnologías únicas y los consumidores van en masa para adoptarlas... ¡a veces!

de la demanda impulsan el mercado. En cualquier caso, ¡un ciclo de innovación y demanda avanza de manera que en ocasiones hace difícil determinar cuál viene primero!

Finalmente, la cultura humana en sí impacta la dirección de la ingeniería del software. Toda generación establece su propia huella sobre la cultura y las próximas no serán diferentes. Faith Popcorn [Pop08], un consultor bien conocido que se especializa en tendencias culturales, las caracteriza en la forma siguiente: "nuestras tendencias no son fruslerías. Ellas perduran, evolucionan. Representan fuerzas subyacentes, causas primeras, necesidades humanas básicas, actitudes, aspiraciones. Nos ayudan a navegar por el mundo, a comprender lo que sucede y por qué, y nos preparan para lo que está por venir". Un análisis detallado de la manera en la que las modernas tendencias culturales tendrán un impacto sobre la ingeniería del software se deja para quienes se especializan en las "ciencias blandas".

31.3.1 Administración de la complejidad

Cuando se escribió la primera edición de este libro (1982), no existían los productos digitales al consumidor como se conocen en la actualidad, y los sistemas basados en mainframe que contenían un millón de líneas de código fuente (LOC) se consideraban muy grandes. Hoy no es raro que pequeños dispositivos digitales abarquen entre 60 000 y 200 000 líneas de software a la medida, acoplado con algunos millones de LOC para características de sistema operativo. Los modernos sistemas basados en computadora que contienen de 10 a 50 millones de líneas de código no son raros.[4] En el futuro relativamente cercano, comenzarán a surgir sistemas[5] que requieran más de mil millones de LOC.[6]

¡Piense en ello por un momento!

Considere las interfaces para un sistema de mil millones de LOC, tanto en el mundo exterior como en otros sistemas interoperables, para la internet (o su sucesor) y para los millones de componentes internos que deben trabajar en conjunto para hacer que opere exitosamente este monstruo de computación. ¿Existe alguna forma confiable de garantizar que todas estas conexiones permitirán que la información fluya de manera adecuada?

Considere el proyecto en sí. ¿Cómo se administra el flujo de trabajo y se rastrea el progreso? ¿Los enfoques convencionales escalarán hacia arriba en órdenes de magnitud?

Considere el número de personas (y sus ubicaciones) que harán el trabajo, la coordinación del personal y la tecnología, el imparable flujo de cambios, la probabilidad de una multiplataforma, en torno de un sistema multioperativo. ¿Existe alguna forma de administrar y coordinar al personal que trabaje en un proyecto monstruoso?

Considere el desafío de ingeniería. ¿Cómo pueden analizarse decenas de miles de requerimientos, limitaciones y restricciones de manera que se garantice que inconsistencia y ambigüedad, omisiones y errores categóricos se descubran y corrijan? ¿Cómo puede crearse una arquitectura de diseño que sea suficientemente robusta para manejar un sistema de este tamaño? ¿Cómo pueden los ingenieros del software establecer un sistema de gestión del cambio que tendrá que manipular cientos de miles de cambios?

Considere el reto de asegurar la calidad. ¿Cómo puede realizarse la verificación y la validación en forma significativa? ¿Cómo se pone a prueba un sistema de mil millones de LOC?

En los primeros días, los ingenieros de software intentaban administrar la complejidad en lo que sólo puede describirse como una forma *ad hoc*. En la actualidad, se usan procesos, métodos

Cita:

"No hay razón por la que alguien quiera una computadora en su casa."

Ken Olson, presidente y fundador de Digital Equipment Corp., 1977

4 Por ejemplo, los modernos sistemas operativos PC (como Linux, MacOS y Windows) tienen entre 30 y 60 millones de LOC. El software de sistema operativo para dispositivos móviles puede superar 2 millones de LOC.

5 En realidad, este "sistema" será un sistema de sistemas: cientos de aplicaciones interoperativas que trabajen en conjunto para lograr algún objetivo global.

6 No todos los sistemas complejos son grandes. Una aplicación relativamente pequeña (por ejemplo, menos de 100 000 LOC) todavía puede ser excesivamente compleja.

y herramientas para mantener bajo control la complejidad. ¿Pero mañana? ¿El enfoque actual depende de la tarea?

31.3.2 Software de mundo abierto

Conceptos tales como inteligencia ambiental,[7] aplicaciones conscientes del contexto y computación dominante/ubicua se enfocan todos en integrar sistemas basados en software en un entorno mucho más amplio que una PC, un dispositivo de computación móvil o cualquier otro dispositivo digital. Estas versiones separadas del futuro cercano de la computación sugieren de manera colectiva "software de mundo abierto", que se diseña para adaptarse a un entorno en cambio continuo "al autorganizar su estructura y autoadaptar su comportamiento" [Bar06].

El software de mundo abierto abarca inteligencia ambiental, aplicaciones conscientes del contexto y computación dominante.

Para ayudar a ilustrar los desafíos que enfrentarán los ingenieros de software en el futuro previsible, considere la noción de *inteligencia ambiental* (amI). Ducatel [Duc01] define la amI en la forma siguiente: "Las personas están rodeadas por interfaces intuitivas inteligentes que se incrustan en todo tipo de objetos. El entorno de la inteligencia ambiental es capaz de reconocer y responder a la presencia de diferentes individuos [mientras trabajan] sin obstrucciones y de manera continua".

Examine una visión del futuro cercano en el que la amI se ha vuelto ubicua. Usted acaba de comprar un comunicador personal (llamado P-com, un dispositivo móvil de bolsillo) y pasó las semanas anteriores creando[8] su "imagen": todo, desde su calendario diario, lista de actividades por hacer, libreta de direcciones, registros médicos, información relacionada con negocios, documentos de viaje, lista de deseos (cosas que busca, por ejemplo, un libro específico, una botella de vino difícil de conseguir, un curso local de soplado de vidrio) y su "Yo-digital" (D-me), que lo describe con un nivel de detalle que permite una presentación digital a otros (una especie de *MySpace* o *FaceBook* que se mueve con usted). El P-com contiene un identificador personal llamado "clave de claves", un identificador personal multifuncional que proporciona acceso y permite consultas desde un amplio rango de dispositivos amI y sistemas.

Debe ser obvio que los temas significativos de privacidad y seguridad entran en juego. Un "sistema de gestión de confianza" [Duc01] será parte integral de amI y gestionará los privilegios que permitan la comunicación con los sistemas de redes, salud, entretenimiento, finanzas, empleo y personal.

Los nuevos sistemas con capacidad amI se agregarán a la red constantemente, y cada uno ofrecerá capacidades útiles y demandará acceso a su P-com. Por tanto, el software P-com debe diseñarse de modo que pueda adaptarse a los requerimientos que surgen cuando algún nuevo sistema amI entra en línea. Existen muchas formas de lograr esto, pero la línea de referencia es la siguiente: el software P-com debe ser flexible y robusto en formas que el software convencional no puede relacionar.

31.3.3 Requerimientos emergentes

Al comienzo de un proyecto de software, existe una verdad obvia que se aplica por igual a todo participante involucrado: "no sabes lo que no sabes". Esto significa que los clientes rara vez definen requerimientos "estables". También significa que los ingenieros del software no siempre pueden prever dónde yacen las ambigüedades e inconsistencias. Los requerimientos cambian, pero eso no es algo nuevo.

---

7 Una valiosa y muy detallada introducción a la inteligencia ambiental puede encontrarse en **www.emerging-communication.com/volume6.html**. Puede obtener más información en **www.ambientintelligence.org/**

8 Toda interacción con el P-com ocurre mediante continuos comandos y enunciados de reconocimiento de voz, que evolucionaron para volverse 99 por ciento precisos.

Puesto que los requerimientos emergentes ya son una realidad, su organización debe considerar adoptar un modelo de proceso incremental.

Conforme los sistemas se vuelven más complejos, incluso un intento rudimentario por establecer requerimientos amplios está condenado al fracaso. Puede intentarse un enunciado de las metas globales, lograrse el delineado de los objetivos intermedios, pero los requerimientos estables, ¡ni por casualidad! Los requerimientos emergerán conforme todos los involucrados en la ingeniería y construcción de un sistema complejo aprendan más acerca de él, del entorno donde reside y de los usuarios con los que interactuará.

Esta realidad implica algunas tendencias en la ingeniería del software. Primero, deben diseñarse modelos de proceso para abarcar el cambio y adoptar los preceptos básicos de la filosofía ágil (capítulo 3). A continuación, deben usarse juiciosamente los métodos que producen modelos de ingeniería (por ejemplo, modelos de requerimientos y diseño) porque dichos modelos cambiarán repetidamente conforme se adquiera más conocimiento acerca del sistema. Finalmente, las herramientas que den apoyo tanto al proceso como a los métodos deben facilitar la adaptación y el cambio.

Pero existe otro aspecto de los requerimientos emergentes. La gran mayoría del software desarrollado a la fecha supone que la frontera entre el sistema basado en software y su entorno externo es estable. La frontera puede cambiar, pero lo hará en forma controlada, lo que permitirá al software adaptarse como parte de un ciclo de mantenimiento de software regular. Esta suposición comienza a cambiar. El software de mundo abierto (sección 31.2.2) demanda que los sistemas basados en computadora "se adapten y reaccionen a los cambios de manera dinámica, incluso si no se anticipan" [Bar06].

Por su naturaleza, los requerimientos emergentes conducen al cambio. ¿Cómo se controla la evolución, durante su ciclo de vida, de una aplicación o sistema que se usa ampliamente y qué efecto tiene esto sobre la forma en la que se diseña software?

Conforme crece el número de cambios, la probabilidad de efectos colaterales no intencionados también lo hace. Esto debe ser una causa de preocupación conforme los sistemas complejos con requerimientos emergentes se vuelven la norma. La comunidad de ingeniería del software debe desarrollar métodos que ayuden a los equipos de software a predecir el impacto de los cambios a través de todo un sistema, lo que, por tanto, mitiga los efectos colaterales no intencionales. En la actualidad, la capacidad para lograr esto está severamente limitada.

31.3.4 La mezcla de talento

A medida que los sistemas basados en software se vuelven más complejos, y conforme la comunicación y la colaboración entre equipos locales se vuelven un lugar común y si los requerimientos emergentes (con el flujo de cambios resultante) se vuelven la norma, la propia naturaleza de un equipo de ingeniería del software puede cambiar. Cada equipo de software debe devolver una variedad de talento creativo y habilidades técnicas a su parte de un sistema complejo, y el proceso global debe permitir que la salida de dichas islas de talento las fusione de manera efectiva.

Alexandra Weber Morales [Mor05] sugiere la mezcla de talento de un "equipo de ensueño de software". El *cerebro* es un arquitecto jefe que puede navegar entre las demandas de los participantes y mapearlas en un marco conceptual tecnológico que puede extenderse e implementarse. La *chica de datos* es una base de datos y gurú de estructuras de datos que "descompone filas y columnas con profunda comprensión de la lógica de predicados y teoría de conjuntos, como pertenecen al modelo relacional". El *bloqueador* es un líder técnico (gerente) que permite al equipo trabajar libre de interferencia de otros miembros del equipo mientras garantiza que ocurra la colaboración. El *hacker* es un programador consumado que está en casa con patrones y lenguajes y puede usarlos de manera efectiva. El *recopilador* "descubre hábilmente requerimientos de sistema con [...] comprensión antropológica" y los expresa con precisión y claridad.

31.3.5 Bloques constructores de software

Cita:

"La respuesta artística adecuada a la tecnología digital es abrazarla como una nueva ventana a todo lo que es eternamente humano, y usarla con pasión, sabiduría, valor y alegría."

Ralph Lombreglia

Quienes fomentan una filosofía de ingeniería del software enfatizan la necesidad de la reutilización de código fuente, clases orientadas a objeto, componentes, patrones y software COTS. Aunque la comunidad de ingeniería del software ha hecho progresos conforme intenta capturar el conocimiento pasado y reutilizar las soluciones probadas, un significativo porcentaje del software que se construye en la actualidad continúa construyéndose "desde cero". Parte de la razón de esto es un deseo continuo (de los participantes y profesionales de ingeniería del software) de "soluciones únicas".

En el mundo del hardware, los fabricantes de equipo original (FEO) de dispositivos digitales usan productos estándar específicos de aplicación (PEEA) producidos por proveedores de silicio de manera casi exclusiva. Estos "hardware mercantiles" proporcionan los bloques constructores necesarios para implementar todo, desde un teléfono celular hasta un reproductor HD-DVD. Cada vez más, los mismos FEO usan "software mercantil", bloques constructores de software diseñados específicamente para un dominio de aplicación único [por ejemplo, dispositivos VoIP]. Michael Ward [War07] comenta:

> Una ventaja del uso de componentes de software es que el FEO puede apalancar la funcionalidad proporcionada por el software sin tener que desarrollar experiencia doméstica en las funciones específicas o invertir tiempo de desarrollador en el esfuerzo por implementar y validar los componentes. Otras ventajas incluyen la habilidad para adquirir y desplegar sólo el conjunto de funcionalidades específicas que son necesarias para el sistema, así como la habilidad para integrar dichos componentes en una arquitectura ya existente.
>
> Sin embargo, el enfoque de componente de software tiene una desventaja porque existe un nivel dado de esfuerzo requerido para integrar los componentes individuales en el producto global. Este reto de integración puede complicarse aún más si los componentes se consiguen de varios proveedores, cada uno con su propia metodología de interfaz. Conforme se usen fuentes adicionales de componentes, el esfuerzo requerido para gestionar varios proveedores aumenta y hay un mayor riesgo de encontrar problemas relacionados con la interacción a través de los componentes de diferentes fuentes.

Además de las componentes empacadas como software mercantil, existe una creciente tendencia a adoptar *soluciones de plataforma de software* que "incorporan colecciones de funcionalidades relacionadas, por lo general proporcionados dentro de un marco conceptual de software integrado" [War07]. Una plataforma de software libera a un FEO del trabajo asociado con el desarrollo de la funcionalidad base y en su lugar permite que dedique el esfuerzo de software en aquellas características que diferencian su producto.

31.3.6 Cambio de percepciones de "valor"

Durante el último cuarto del siglo xx, la pregunta operativa que planteaban los empresarios cuando analizaban el software era: ¿por qué cuesta tanto? Esta pregunta rara vez se plantea en la actualidad y se sustituyó por: ¿por qué no podemos tenerlo (el software y/o el producto basado en software) más pronto?

Cuando se considera el software de computadora, la percepción moderna del valor cambia del valor empresarial (costo y rentabilidad) a valores de cliente que incluyen: rapidez de entrega, riqueza de funcionalidad y calidad global del producto.

31.3.7 Fuente abierta

¿Quién posee el software que usted o su organización usa? Cada vez más, la respuesta es "todo mundo". El movimiento "fuente abierta" se ha descrito de la forma siguiente [OSO08]: "Fuente abierta es un método de desarrollo para software que aprovecha el poder de la revisión de pares

distribuida y la transparencia de procesos. La promesa de la fuente abierta es mejor calidad, mayor confiabilidad, más flexibilidad, costo más bajo y terminar con el candado del proveedor depredador". El término *fuente abierta*, cuando se aplica a software de computadora, implica que los productos finales de ingeniería del software (modelos, código fuente, suites de pruebas) están abiertos al público y pueden revisarse y extenderse (con controles) por cualquiera que tenga interés y cuente con permiso.

Un "equipo" fuente abierta puede tener algunos miembros del "equipo de ensueño" de tiempo completo (sección 31.3.4), pero el número de personas que trabajan en el software se expande y contrae conforme el interés en la aplicación se fortalece o debilita. El poder del equipo de fuente abierta se deriva de la constante revisión de pares y de la refactorización de diseño/código que da como resultado un avance lento hacia una solución óptima.

Si el lector tiene más interés, Weber [Web05] proporciona una valiosa introducción y Feller *et al.* [Fel07] editaron una antología exhaustiva y objetiva que considera los beneficios y problemas asociados con la fuente abierta.

INFORMACIÓN

Tecnologías por llegar

Es probable que muchas tecnologías emergentes tengan un impacto significativo sobre los tipos de sistemas basados en computadora que evolucionan. Dichas tecnologías se agregan a los retos que enfrentan los ingenieros del software. Vale la pena anotar las siguientes tecnologías:

Computación de malla (*grid computing*): esta tecnología (disponible en la actualidad) crea una red que emplea los miles de millones de ciclos CPU que estén sin usar de cualquier máquina de la red y permite completar tareas de cómputo excesivamente complejas sin una supercomputadora dedicada a ello. Para un ejemplo de la vida real que abarca más de 4.5 millones de computadoras, visite http://setiathome.berkeley.edu/

Computación de mundo abierto: "Es ambiental, implícita, invisible y adaptativa. Consiste en que los dispositivos en red incrustados en el entorno proporcionan conectividad y servicios en todo momento sin obstrucciones" [McC05].

Microcomercio: una nueva rama del comercio electrónico que cobra cantidades muy pequeñas por acceder y/o comprar varios productos con propiedad intelectual. Apple iTunes es un ejemplo ampliamente usado.

Máquinas cognitivas: el "santo grial" en el campo de la robótica es el desarrollo de máquinas que estén conscientes de su entorno, que puedan "recoger pistas, responder a situaciones siempre cambiantes e interactuar con personas de manera natural" [PCM03]. Las máquinas cognitivas todavía están en las primeras etapas de desarrollo, pero su potencial es enorme.

Pantallas OLED: una OLED "utiliza una molécula de diseñador basada en carbono que emite luz cuando una corriente eléctrica pasa a través de ella. Coloque muchas moléculas juntas y obtendrá una pantalla superdelgada de sorprendente calidad; no se requiere iluminación trasera que extraiga energía" [PCM03]. El resultado: pantallas ultradelgadas que pueden enrollarse y doblarse, extenderse sobre superficies curvas o adaptarse de otro modo a un entorno específico.

RFID: la identificación por radiofrecuencia lleva a la computación de mundo abierto a una base industrial y a la industria de productos al consumidor. Todo, desde tubos de dentífrico hasta motores de automóviles, se identificará conforme se muevan a través de la cadena de suministro hasta su destino final.

Web 2.0: uno de varios servicios web que conducirá a una integración todavía mayor de la web tanto en comercio como en computación personal.

Para mayor análisis de las tecnologías por llegar, presentadas en una combinación única de video e impreso, visite el sitio web de Consumer Electronics Association en **www.ce.org/Press/CEA_Pubs/135.asp**

31.4 DIRECCIONES DE LA TECNOLOGÍA

La gente siempre parece creer que la ingeniería del software cambiará más rápidamente de lo que lo hace. Se introduce una nueva "promoción excesiva" de tecnología (un nuevo proceso, un método único o una herramienta excitante) y los expertos sugieren que "todo" cambiará. Pero la ingeniería del software tiene que ver menos con la tecnología y más con las personas y su habilidad para comunicar sus necesidades y para convertir dichas necesidades en realidad. Siempre que se involucran personas, ocurren cambios lentamente en irrupción repetida. Sólo

Cita:

"¿Pero para qué sirve?"

Comentario de un ingeniero de la División de Sistemas de Cómputo Avanzados de IBM, 1968, acerca del microchip

cuando se alcanza un "punto de no retorno" [Gla02], la tecnología cae en cascada a través de la comunidad de la ingeniería del software y ocurre verdaderamente un cambio muy amplio.

En esta sección se examinarán algunas tendencias sobre proceso, métodos y herramientas que es probable que tengan alguna influencia en la ingeniería del software durante la próxima década. ¿Ello conducirá a un punto de no retorno? Sólo es cuestión de esperar y ver.

31.4.1 Tendencias de proceso

Puede argumentarse que todas las tendencias empresariales, organizativas y culturales estudiadas en la sección 31.3 refuerzan la necesidad del proceso. ¿Pero los marcos conceptuales analizados en el capítulo 30 proporcionan un mapa de caminos hacia el futuro? ¿Los marcos conceptuales de proceso evolucionarán para encontrar un mejor equilibrio entre disciplina y creatividad? ¿El proceso de software se adaptará a las diferentes necesidades de los participantes que procuran el software, los que lo construyen y quienes lo usan? ¿Puede haber un medio para reducir el riesgo para los tres grupos al mismo tiempo?

Estas y muchas otras preguntas siguen abiertas. Sin embargo, algunas tendencias comienzan a surgir. Conradi y Fuggetta [Con02] sugieren seis "tesis acerca de cómo aumentar y aplicar mejor los marcos conceptuales MPS". Ellos comienzan su análisis con el siguiente enunciado:

> Una meta de quien procura el software es seleccionar al mejor contratista objetiva y racionalmente. El objetivo de una compañía de software es sobrevivir y crecer en un mercado competitivo. La de un usuario final es adquirir el producto de software que pueda resolver el problema correcto, en el momento correcto, a un precio aceptable. No podemos esperar el mismo enfoque MPS y el esfuerzo consecuente para alojar todos estos puntos de vista.

En los siguientes párrafos se adaptan las tesis propuestas por Conradi y Fuggetta [Con02] para sugerir posibles tendencias de proceso durante la próxima década.

? ¿Qué tendencias de proceso son probables durante la próxima década?

1. *Conforme evolucionan los marcos conceptuales MPS, enfatizarán "estrategias que se enfocan en la orientación de metas y en la innovación del producto"* [Con02]. En el mundo acelerado del desarrollo de software, las estrategias MPS a largo plazo rara vez sobreviven en un entorno empresarial dinámico. Demasiados cambios muy rápidamente. Esto significa que es posible que un mapa estable de caminos paso a paso deba sustituirse con un marco conceptual que enfatice las metas a corto plazo y que tenga una orientación de producto. Si los requerimientos para una nueva línea de productos basados en software surgirá durante una serie de liberaciones de producto incrementales (para entrega a usuarios finales mediante la web), la organización de software puede reconocer la necesidad de mejorar su capacidad para gestionar el cambio. El mejoramiento de proceso asociado con la gestión del cambio debe coordinarse con los ciclos de liberación del producto, de manera que mejorará la gestión del cambio mientras al mismo tiempo no lo perturbe.
2. *Puesto que los ingenieros del software tienen un buen sentido de dónde está débil el proceso, los cambios en el proceso por lo general deben impulsarse por sus necesidades y deben comenzar en forma ascendente.* Conradi y Fuggetta [Con02] sugieren que las futuras actividades MPS deben "usar una tarjeta de calificaciones simple y enfocada con la cual comenzar, no con una gran valoración". Al enfocar los esfuerzos MPS estrechamente y trabajar de manera ascendente, los profesionales comenzarán a ver cambios sustantivos más pronto, cambios que hacen una diferencia real en la forma como se realiza el trabajo en ingeniería del software.
3. *La tecnología de procesos de software automatizados (PSA) se alejará de la gestión de proceso global (amplio apoyo de todo el proceso de software) y se enfocará en aquellos aspectos del proceso de software que puedan beneficiarse mejor de la automatización.* Nadie está en

contra de las herramientas y de la automatización, pero, en muchas instancias, la PSA no cumple su promesa (vea la sección 31.2). Para ser más efectiva, debe enfocarse en actividades sombrilla (capítulo 2), los elementos más estables del proceso de software.

4. *Se colocará mayor énfasis en el rendimiento sobre la inversión de las actividades MPS.* En el capítulo 30 aprendió que el rendimiento sobre la inversión (RSI) puede definirse como:

$$\text{RSI} = \frac{\Sigma(\textit{beneficios}) - \Sigma(\textit{costos})}{\Sigma(\textit{costos})} \times 100\%$$

A la fecha, las organizaciones de software han luchado por delinear con claridad los "beneficios" en forma cuantitativa. Puede argumentarse [Con02] que "por tanto, necesitamos un modelo estandarizado con valor de mercado, como el que se utiliza en Cocomo II (vea el capítulo 26) para explicar las iniciativas de mejoramiento del software".

5. *Conforme pasa el tiempo, la comunidad del software puede llegar a entender que la experiencia en sociología y antropología puede tener tanto o más que ver con el éxito de MPS que otras disciplinas más técnicas.* Sobre todo, MPS cambia la cultura de la organización, y el cambio cultural involucra individuos y grupos de personas. Conradi y Fuggetta [Con02] anotan correctamente que "los desarrolladores de software son trabajadores del conocimiento. Tienden a responder negativamente a los dictados de esferas superiores acerca de cómo trabajar o cambiar los procesos". Es posible aprender mucho al examinar la sociología de los grupos para entender mejor las formas efectivas de introducir el cambio.

6. *Nuevos modos de aprendizaje pueden facilitar la transición hacia un proceso de software más efectivo.* En este contexto, "aprendizaje" implica aprendizaje de éxitos y errores. Una organización de software que recopila métricas (capítulos 23 y 25) se permite entender cómo los elementos de un proceso afectan la calidad del producto final.

31.4.2 El gran desafío

Existe una tendencia que es innegable: los sistemas basados en software sin duda se volverán más grandes y más complejos conforme pase el tiempo. Es la ingeniería de estos grandes sistemas complejos, sin importar la plataforma de entrega o el dominio de aplicación, la que impone el "gran desafío" [Bro06] a los ingenieros del software. Manfred Broy [Bro06] sugiere que los ingenieros del software pueden enfrentar el "intimidante reto de desarrollar sistemas de software complejos" al crear nuevos enfoques para entender los modelos de sistema y usar dichos modelos como base para la construcción de software de próxima generación de alta calidad.

Conforme la comunidad de ingeniería del software desarrolla nuevos enfoques impulsados por modelo (que se estudian más adelante en esta sección) para la representación de los requerimientos del sistema y su diseño, pueden abordarse las siguientes características [Bro06]:

? ¿Qué características del sistema deben considerar analistas y diseñadores para futuras aplicaciones?

- *Multifuncionalidad:* conforme los dispositivos digitales evolucionan hacia su segunda y tercera generaciones, comienzan a entregar un rico conjunto de, en ocasiones, funciones no relacionadas. El teléfono celular, alguna vez considerado un dispositivo de comunicación, ahora se utiliza para tomar fotografías, conservar un calendario, navegar por un diario y como reproductor de música. Si las interfaces de mundo abierto llegan a trascender, estos dispositivos móviles se usarán para mucho más durante los próximos años. Como anota Broy [Bro06], "los ingenieros deben describir el contexto detallado en el que se entregarán las funciones y, más importante, deben identificar las interacciones potencialmente dañinas entre las diferentes características del sistema".
- *Reactividad y oportunidad:* los dispositivos digitales interactúan cada vez más con el mundo real y deben reaccionar a estímulos externos en forma oportuna. Deben poner

interfaz con una amplia serie de sensores y responder en un marco temporal que sea adecuado a la tarea que se busca realizar. Deben desarrollarse nuevos métodos que 1) ayuden a los ingenieros de software a predecir la temporalidad de varias características reactivas y 2) implementen dichas características de manera que las haga menos dependientes de la máquina y más portátiles.

- *Nuevos modos de interacción con el usuario:* el teclado y el ratón funcionan bien en un entorno PC, pero las tendencias de mundo abierto para software señalan que deben modelarse e implementarse nuevos modos de interacción. Ya sea que dichos nuevos enfoques usen interfaces de toque múltiple, reconocimiento de voz o interfaces de mente directa,[9] las nuevas generaciones de software para dispositivos digitales deben modelar estas nuevas interfaces humano-computadora.
- *Arquitecturas complejas:* un automóvil lujoso tiene más de 2 000 funciones controladas mediante software que residen dentro de una compleja arquitectura de hardware que incluye múltiples CPU, una sofisticada estructura de bus, actuadores, sensores, una interfaz humana cada vez más sofisticada, y muchos componentes con clasificación de seguridad. Sistemas incluso más complejos están en el horizonte inmediato, lo que presenta retos significativos para los diseñadores de software.
- *Sistemas heterogéneos distribuidos:* los componentes en tiempo real para cualquier moderno sistema incrustado pueden conectarse mediante un bus interno, una red inalámbrica o a través de internet (o todo junto).
- *Crucialidad:* el software se ha convertido en el componente pivote en virtualmente todos los sistemas cruciales para los negocios y en la mayoría de los sistemas importantes para la seguridad. Sin embargo, la comunidad de ingeniería del software apenas comienza a aplicar incluso los principios más básicos de la seguridad de software.
- *Variabilidad de mantenimiento:* la vida del software dentro de un dispositivo digital rara vez dura más allá de 3 a 5 años, pero los complejos sistemas de aviónica dentro de una aeronave tienen una vida útil de al menos 20 años. El software de los automóviles falla en alguna parte intermedia. ¿Esto tendrá algún impacto sobre el diseño?

Broy [Bro06] argumenta que éstas y otras características del software pueden gestionarse solamente si la comunidad de ingeniería del software desarrolla una filosofía de ingeniería del software distribuida de manera más efectiva y colaborativa, mejores enfoques de ingeniería de requerimientos, un enfoque más robusto al desarrollo impulsado por modelo y mejores herramientas de software. En las secciones que siguen se explorará brevemente cada una de estas áreas.

31.4.3 Desarrollo colaborativo

PUNTO CLAVE
La colaboración involucra la diseminación oportuna de la información y un proceso efectivo para comunicarse y tomar decisiones.

Parece casi demasiado obvio de afirmar, pero se hará de cualquier forma: *la ingeniería del software es una tecnología de información.* Desde el inicio de cualquier proyecto de software, cada participante debe compartir información: acerca de las metas y objetivos empresariales básicos, de los requerimientos de sistema específicos, de conflictos de diseño arquitectónico, de casi todo aspecto del software que se va a construir.

En la actualidad, los ingenieros de software colaboran a través de zonas horarias y fronteras internacionales, y cada uno de ellos debe compartir información. Lo mismo es cierto para los proyectos de fuente abierta en los que trabajan cientos o miles de desarrolladores de software

9 Un breve estudio acerca de las interfaces de mente directa puede encontrarse en **http://en.wikipedia.org/wiki/Brain-computer_interface**, y un ejemplo comercial se describe en **http://au.gamespot.com/news/6166959.html**

para construir una aplicación de fuente abierta. De nuevo, la información debe diseminarse de modo que pueda ocurrir la colaboración abierta.

El reto durante la próxima década es desarrollar métodos y herramientas que faciliten dicha colaboración. Hoy día, continúa la lucha por facilitar la colaboración. Eugene Kim [Kim04] comenta:

> Considere una tarea de colaboración básica: compartir documentos. Algunas aplicaciones (tanto comerciales como de fuente abierta) afirman resolver el problema de compartir documentos y, sin embargo, el método predominante para compartir archivos es enviarlos por correo electrónico de ida y vuelta. Éste es el equivalente computacional de la *sneakernet*. Si las herramientas que tienen el propósito de resolver este problema son buenas, ¿por qué no se utilizan?
>
> En otras áreas básicas se ven problemas similares. Puedo ir a una reunión en cualquier parte del mundo con un trozo de papel en la mano, y puedo estar seguro de que la gente querrá leerlo, marcarlo, darle vueltas y archivarlo. No puedo decir lo mismo de los documentos electrónicos. No puedo anotar en una página web ni usar el mismo sistema de llenado para el correo electrónico y para los documentos de Word, al menos no en una forma que garantice la interoperabilidad con las aplicaciones de mi propia máquina o de otras. ¿Por qué no?
>
> ...Con la finalidad de tener un impacto real en el espacio colaborador, las herramientas no sólo deben ser buenas, deben ser interoperables.

Pero la falta de herramientas colaborativas amplias sólo es una parte del reto que enfrentan quienes deben desarrollar software de manera colaborativa.

En la actualidad, un porcentaje significativo[10] de los proyectos IT se subcontratan a nivel internacional, y el número crecerá sustancialmente durante la siguiente década. No es de sorprender que Bhat *et al.* [Bha06] aseveren que la ingeniería de requerimientos es la actividad crucial en un proyecto subcontratado. Los autores identifican algunos factores de éxito que conducen a esfuerzos de colaboración exitosos:

- *Metas compartidas:* las metas del proyecto deben enunciarse con claridad, y todos los participantes deben comprenderlas y estar de acuerdo con su intención.
- *Cultura compartida:* las diferencias culturales deben definirse con claridad; debe desarrollarse un enfoque educativo (que ayudará a mitigar dichas diferencias) y un enfoque de comunicación (que facilitará la transferencia de conocimiento).
- *Proceso compartido:* en algunas formas, el proceso funciona como el esqueleto de un proyecto colaborativo, lo que proporciona un medio uniforme para valorar el progreso y la dirección, y para introducir un "lenguaje" técnico común para todos los miembros del equipo.
- *Responsabilidad compartida:* todo miembro del equipo debe reconocer la importancia de la ingeniería de requerimientos y trabajar para ofrecer la mejor definición posible del sistema.

Cuando se combinan, dichos factores de éxito conducen a la "confianza": un equipo global que puede apoyarse en grupos dispares para lograr el trabajo que se les asignó.

31.4.4 Ingeniería de requerimientos

En los capítulos del 5 al 7 se presentaron las acciones básicas de la ingeniería de requerimientos: adquisición, elaboración, negociación, especificación y validación. El éxito o fracaso de dichas

10 Actualmente, alrededor de 20 por ciento de un presupuesto IT típico de las compañías grandes se dedica a subcontratación (outsourcing) y el porcentaje crece cada año. (Fuente: www.logicacmg.com/page/400002849.)

acciones tiene una influencia muy fuerte sobre el éxito o el fracaso de todo el proceso de ingeniería del software. Y, sin embargo, la ingeniería de requerimientos (IR) se compara con "intentar poner una abrazadera de sujeción alrededor de una gelatina" [Gon04]. Como se señala en muchos lugares a lo largo de este libro, los requerimientos de software tienen la tendencia a seguir cambiando, y con la llegada de los sistemas de mundo abierto, los requerimientos emergentes (y el cambio casi continuo) pueden volverse la norma.

En la actualidad, la mayoría de los enfoques de ingeniería de requerimientos "informales" comienzan con la creación de escenarios de usuario (por ejemplo, casos de uso). Los enfoques más formales crean uno o más modelos de requerimientos y el uso de los mismos como base para el diseño. Los métodos formales permiten que un ingeniero de software represente los requerimientos, usando una notación matemática verificable. Todo puede funcionar razonablemente bien cuando los requerimientos son estables, pero no se resuelve fácilmente el problema de los requerimientos dinámicos o emergentes.

Existen algunas direcciones distintas en la investigación en ingeniería de requerimientos, incluidos el procesamiento de lenguaje natural a partir de descripciones textuales traducidas en representaciones más estructuradas (por ejemplo, clases de análisis), mayor apoyo sobre bases de datos para estructurar y comprender los requerimientos de software, el uso de patrones IR para describir problemas y soluciones usuales cuando se realizan las tareas de ingeniería de requerimientos y la ingeniería de requerimientos orientada a metas. Sin embargo, industrialmente, las acciones de IR siguen siendo más o menos informales y sorprendentemente básicas. Para mejorar la forma en la que se definen los requerimientos, la comunidad de ingeniería del software probablemente implementará tres subprocesos distintos conforme se lleve a cabo la IR [Gli07]: 1) adquisición de conocimiento mejorado y compartición de conocimiento que permita comprensión más completa de las restricciones del dominio de aplicación y las necesidades de los participantes, 2) mayor énfasis en la iteración conforme se definen los requerimientos y 3) herramientas de comunicación y coordinación más efectivas que permitan a todos los participantes colaborar de manera efectiva.

PUNTO CLAVE

"Los nuevos subprocesos IR incluyen: 1) adquisición de conocimiento mejorado, 2) incluso más iteración y 3) herramientas de comunicación y coordinación más efectivas".

Los subprocesos IR señalados en el párrafo anterior solamente triunfarán si se integran de manera adecuada en un enfoque evolutivo a la ingeniería del software. Conforme la resolución de problemas basada en patrones y las soluciones basadas en componentes comiencen a dominar muchos dominios de aplicación, la IR debe acomodar el deseo de agilidad (rápida entrega incremental) y los requerimientos emergentes inherentes que resulten. La naturaleza concurrente de muchos modelos de proceso en ingeniería del software significa que la IR se integrará con actividades de diseño y construcción. En consecuencia, la noción de una "especificación de software" estática comienza a desaparecer, para ser sustituida por "requerimientos impulsados por valores" [Som05] derivada conforme los participantes responden a las características y funciones entregadas en los incrementos de software anteriores.

31.4.5 Desarrollo de software impulsado por modelo

Los ingenieros de software se aferran a la abstracción virtualmente en cada paso del proceso de ingeniería del software. Conforme comienza el diseño, las abstracciones arquitectónicas y en el nivel de componente se representan y valoran. Luego deben traducirse en una representación en lenguaje de programación que transforme el diseño (un nivel de abstracción relativamente alto) en un sistema operativo con un entorno de computación específico (un nivel de abstracción bajo). El *desarrollo de software impulsado por modelo*[11] acopla lenguajes de modelado específicos de dominio con motores y generadores de transformación, de manera que facilita la representación de la abstracción en niveles altos y luego los transforma en niveles más bajos [Sch06].

PUNTO CLAVE

El enfoque impulsado por modelo enfrenta un desafío permanente para todos los desarrolladores de software: cómo representar el software a un nivel mas alto de abstracción que en código.

11 También se usa el término *ingeniería impulsada por modelo* (IIM).

Los *lenguajes de modelado específicos de dominio* (LMED) representan "la estructura, comportamiento y requerimientos de la aplicación dentro de dominios de aplicación particulares" y se describen con metamodelos que "definen las relaciones entre conceptos en el domino y especifican con precisión la semántica y restricciones clave asociadas con dichos conceptos de dominio" [Sch06]. La diferencia principal entre un LMED y un lenguaje de modelado de propósito general como UML (apéndice 1) es que el primero se sintoniza con los conceptos de diseño inherentes en el dominio de aplicación y, por tanto, puede representar relaciones y restricciones entre elementos de diseño en forma eficiente.

31.4.6 Diseño posmoderno

En un interesante artículo acerca del diseño de software en la "era posmoderna", Philippe Kruchten [Kru05] hace la siguiente observación:

> La ciencia de la computación no ha logrado la gran narrativa que explique todo, el *gran cuadro*: no hemos encontrado las leyes fundamentales del software que jugarían el papel que las leyes fundamentales de la física juegan en otras disciplinas de la ingeniería. Todavía vivimos con el resabio amargo de la explosión de burbujas de internet y el día del juicio final Y2K. De modo que, en esta era posmoderna, donde parece que todo importa un poco, aunque realmente no importa mucho, ¿cuáles son las siguientes direcciones para el diseño de software?

Parte de cualquier intento por comprender las tendencias en el diseño del software es establecer fronteras al diseño. ¿Dónde se detiene la ingeniería de requerimientos y comienza el diseño? ¿Dónde se detiene el diseño y comienza la generación de código? Las respuestas a estas preguntas no son sencillas como podrían parecer al principio. Aun cuando el modelo de requerimientos deba enfocarse en "qué", no en "cómo", todo analista hace un poco de diseño y casi todos los diseñadores hacen un poco de análisis. De igual modo, conforme el diseño de componentes de software se acerca un poco más al detalle algorítmico, un diseñador comienza a representar el componente en un nivel de abstracción que está cerca del código.

El diseño posmoderno continuará enfatizando la importancia de la arquitectura del software (capítulo 9). Un diseñador debe enunciar un conflicto arquitectónico, tomar una decisión que aborde el conflicto y luego definir con claridad las suposiciones, restricciones e implicaciones que la decisión impone sobre el software como un todo. Pero, ¿existe un marco conceptual donde los conflictos pueden describirse y la arquitectura puede definirse? El desarrollo de software orientado a aspecto (capítulo 2) o el desarrollo de software impulsado por modelo (sección 31.4.4) pueden convertirse en importantes enfoques de diseño en los años por venir, pero todavía es muy pronto para decirlo. Puede ser que la innovación en el desarrollo basado en componentes (capítulo 10) pueda conducir a una filosofía de diseño que enfatice el ensamblado de los componentes existentes. Si el pasado es prólogo, es enormemente probable que surjan muchos "nuevos" métodos de diseño, pero pocos remontarán la curva de la promoción excesiva (figura 31.2) mucho más allá del "valle de la desilusión".

31.4.7 Desarrollo impulsado por pruebas

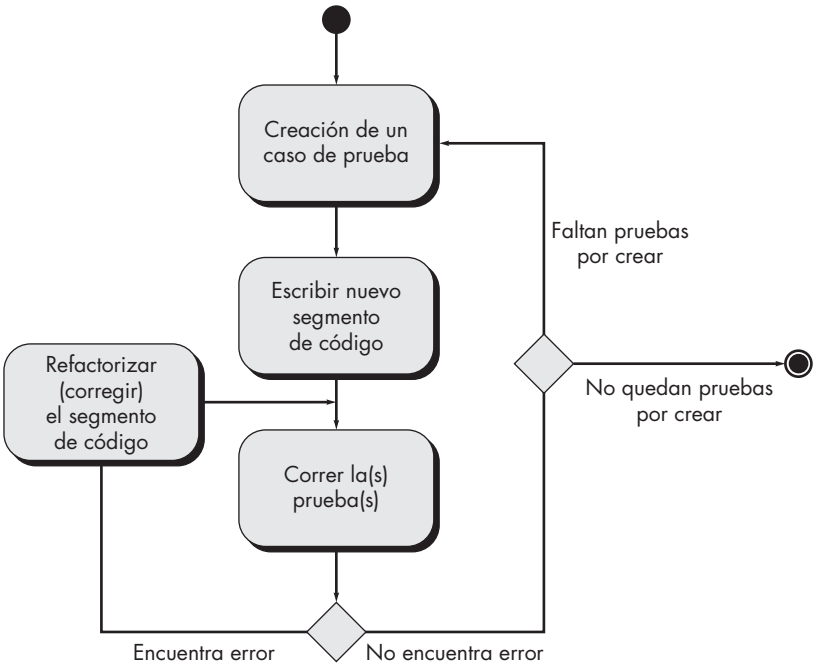
Los requerimientos impulsan el diseño y éste establece un cimiento para construcción. Esta simple realidad en ingeniería del software funciona razonablemente bien y es esencial conforme se crea una arquitectura de software. Sin embargo, un cambio sutil puede proporcionar beneficios significativos cuando se consideran el diseño en el nivel de componentes y la construcción.

PUNTO CLAVE

"DIP es una tendencia que enfatiza el diseño de casos de prueba antes de la creación de código fuente".

En el *desarrollo impulsado por pruebas* (DIP), los requerimientos para un componente de software funcionan como la base para la creación de una serie de casos de prueba que ejerciten la interfaz y que intenten encontrar errores en las estructuras de datos y en la funcionalidad que

FIGURA 31.3
Flujo de proceso de desarrollo impulsado por pruebas



entrega el componente. El DIP realmente no es una nueva tecnología, sino más bien una tendencia que enfatiza el diseño de casos de prueba *antes* de la creación de código fuente.[12]

El proceso DIP sigue el simple flujo procedural que se ilustra en la figura 31.3. Antes de crear el primer pequeño segmento de código, un ingeniero de software crea una prueba para ejercitar el código (intentar que fracase el código). Entonces el código se escribe para satisfacer la prueba. Si la pasa, se crea una nueva prueba para el siguiente segmento de código que se va a desarrollar. El proceso continúa hasta que el componente está completamente codificado y todas las pruebas se ejecutan sin error. Sin embargo, si alguna prueba triunfa al encontrar un error, el código existente se refactoriza (corrige) y todas las pruebas creadas para dicho punto se vuelven a ejecutar. Este flujo iterativo continúa hasta que no hay pruebas pendientes de crear, lo que implica que los componentes satisfacen todos los requerimientos definidos para él.

Durante el DIP, el código se desarrolla en incrementos muy pequeños (una subfunción a la vez) y no se escribe código hasta que exista una prueba que lo ejercite. Debe observar que cada iteración resulta en una o más pruebas nuevas que se agregan a una suite de pruebas de regresión que corren con cada cambio. Esto se hace para garantizar que el nuevo código no generó efectos colaterales que causen errores en el código anterior.

En DIP, las pruebas impulsan el diseño de componentes detallados y el código fuente resultante. Los resultados de dichas pruebas causan modificaciones inmediatas al diseño de componentes (vía el código) y, más importante, el componente resultante (cuando se completa) se verificó en forma independiente. Si tiene más interés en el DIP, consulte [Bec04b] o [Ast04].

31.5 TENDENCIAS RELACIONADAS CON HERRAMIENTAS

Cada año se introducen cientos de herramientas de ingeniería del software de grado industrial. La mayoría las aportan los proveedores de herramientas, quienes afirman que su herramienta

12 Recuerde que la programación extrema (capítulo 3) enfatiza este enfoque como parte de su modelo de proceso ágil.

mejorará la administración del proyecto, el análisis de requerimientos, el modelado de diseño, la generación de código, las pruebas, la gestión del cambio o cualquiera de las muchas actividades, acciones y tareas de la ingeniería del software que se estudian a lo largo de este libro. Otras herramientas se desarrollaron como ofrecimientos de fuente abierta. La mayoría de las herramientas de fuente abierta se enfocan en las actividades de "programación" con un énfasis específico en la actividad de construcción (particularmente la generación de código). Incluso otras herramientas se derivan de esfuerzos de investigación en universidades y laboratorios gubernamentales. Aunque tienen atractivo en aplicaciones muy limitadas, la mayoría no está lista para aplicación industrial amplia.

En el nivel industrial, los paquetes de herramientas más amplios forman *entornos de ingeniería del software* (EIS)[13] que integran una colección de herramientas individuales en torno de una base de datos central (repositorio). Cuando se considera como un todo, un EIS integra información a través del proceso de software y auxilia en la colaboración que se requiere para sistemas basados en computadora muy grandes y complejos. Pero los entornos actuales no son fácilmente extensibles (es difícil integrar una herramienta COTS que no sea parte del paquete) y tienden a ser de propósito general (es decir, no son específicas de dominio de aplicación). También existe una considerable demora temporal entre la introducción de las nuevas soluciones tecnológicas (por ejemplo, desarrollo de software impulsado por modelo) y la disponibilidad de EIS viables que den soporte a la nueva tecnología.

Las futuras tendencias en las herramientas de software seguirán dos rutas distintas: una *ruta enfocada a lo humano* que responda a algunas de las "tendencias blandas" estudiadas en la sección 31.3 y una ruta centrada en tecnología que aborde las nuevas tecnologías (sección 31.4) conforme se introduzcan y adopten. En las siguientes secciones se examinará brevemente cada ruta.

31.5.1 Herramientas que responden a tendencias blandas

Las tendencias blandas analizadas en la sección 31.3 (la necesidad de administrar la complejidad, acomodar requerimientos emergentes, establecer modelos de proceso que aborden el cambio, coordinar equipos globales con una mezcla de talento cambiante, entre otros) sugiere una nueva era en la que las herramientas que apoyen la colaboración de los participantes se volverán tan importantes como las herramientas que apoyan la tecnología. ¿Pero qué tipo de conjunto de herramientas soportan dichas tendencias blandas?

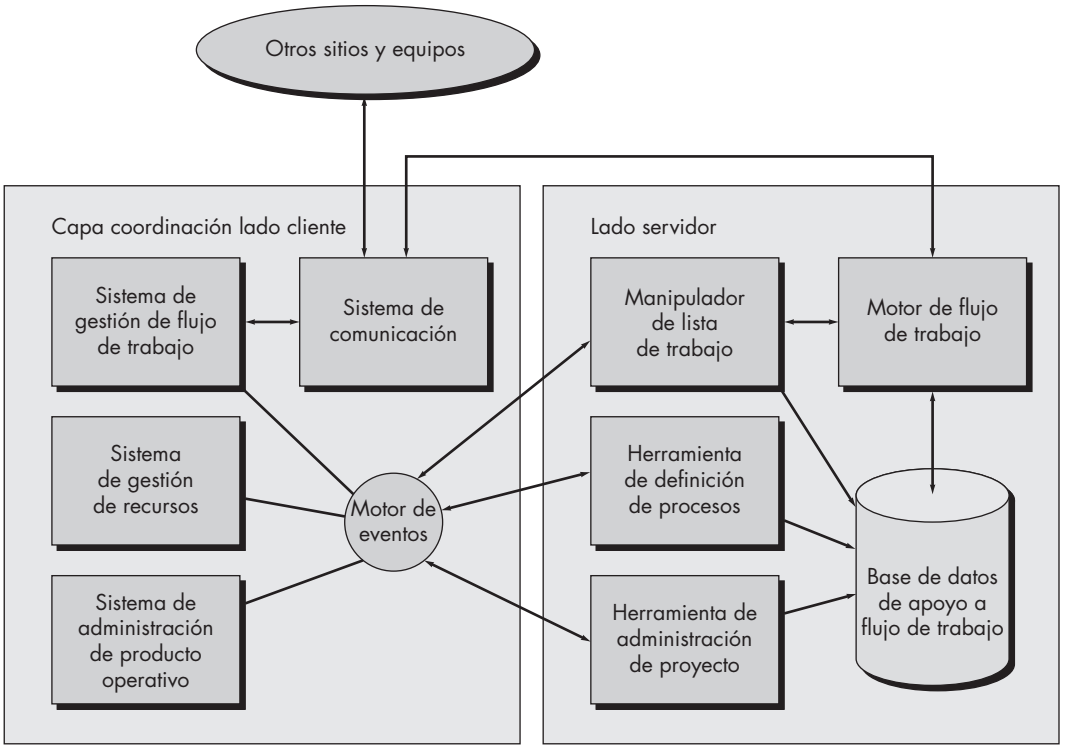
Un ejemplo de investigación en esta área es GENESIS, un entorno generalizado de fuente abierta diseñado para soportar trabajo colaborativo de ingeniería del software [Ave04]. El entorno GENESIS puede o no volverse de uso amplio, pero sus elementos básicos son representativos de la dirección de EIS colaboradores que evolucionarán para dar apoyo a las tendencia blandas anotadas en este capítulo.

Un EIS colaborativo "soporta la cooperación y la comunicación entre ingenieros de software que pertenecen a equipos de desarrollo distribuidos, involucrados en modelar, controlar y medir el desarrollo del software y los procesos de mantenimiento. Más aún, incluye una función de administración de artefacto que almacena y gestiona artefactos de software (productos operativos) producidos por diferentes equipos en el curso de su "trabajo" [Bol02].

La figura 31.4 ilustra una arquitectura para un EIS colaborativo. La arquitectura, basada en el entorno GENESIS [Ave04], se construye con subsistemas que se integran dentro de un cliente web común y que se complementa mediante componentes basados en servidor que proporcionan apoyo a todos los clientes. Cada organización de desarrollo tiene sus propios subsistemas en el lado cliente que se comunican con otros clientes. En la figura 31.4, un *subsistema de gestión*

13 También se usa el término *entorno de desarrollo integrado* (EDI).

FIGURA 31.4
Arquitectura EIS colaborativa
Fuente: Adaptado de [Ave04].



de recursos administra la asignación de recursos humanos a diferentes proyectos o subproyectos; un *sistema de administración de producto de trabajo* es "responsable de la creación, modificación, borrado", indexado, búsqueda y almacenamiento de todos los productos operativos de la ingeniería del software [Ave04]; un *subsistema de gestión de flujo de trabajo* coordina la definición, ejemplificación e implementación de las actividades, acciones y tareas del proceso de software; un *motor de evento* "recopila eventos" que ocurren durante el proceso de software (por ejemplo, una revisión exitosa de un producto operativo, la conclusión de pruebas de unidad de un componente) y lo notifica a otros; un *sistema de comunicación* soporta comunicación sincrónica y asíncrona entre los equipos distribuidos.

En el lado servidor, cuatro componentes comparten una base de datos de apoyo al flujo de trabajo. Los componentes implementan las siguientes funciones:

- *Definición de proceso:* un conjunto de herramientas que permiten a un equipo definir nuevas actividades, acciones o tareas de proceso y que define las reglas que gobiernan la manera en la que interactúan dichos elementos unos con otros y los productos operativos que producen.
- *Administración de proyecto:* un conjunto de herramientas que permiten al equipo construir un plan de proyecto y coordinar el plan con otros equipos o proyectos.
- *Motor de flujo de trabajo:* "interactúa con el motor de eventos para propagar eventos que son relevantes para la ejecución de procesos cooperativos que se ejecutan en otros sitios" [Ave04].
- *Manipulador de lista de trabajo:* interactúa con la base de datos del lado servidor para brindar a un ingeniero de software información acerca de la tarea que actualmente se desarrolla o cualquier tarea futura que se derive del trabajo que actualmente se realiza.

Aunque la arquitectura de un EIS colaborativo puede variar considerablemente del que se estudió en esta sección, los elementos funcionales básicos (sistemas y componentes de administración) parecerán lograr el nivel de coordinación que se requiere para un proyecto de ingeniería del software distribuida.

31.5.2 Herramientas que abordan tendencias tecnológicas

La agilidad en la ingeniería del software (capítulo 3) se logra cuando los participantes trabajan como un equipo. Por tanto, la tendencia hacia los EIS colaborativos (sección 31.5.1) brindará beneficios aun cuando el software se desarrolle de manera local. Pero, ¿qué hay acerca de las herramientas tecnológicas que complementan el sistema y los componentes que fortalecen una mejor colaboración?

Una de las tendencias dominantes en las herramientas tecnológicas es la creación de un conjunto de herramientas que da apoyo al desarrollo impulsado por modelo (sección 31.4.4) con énfasis en el diseño impulsado por arquitectura. Oren Novotny [Nov04] sugiere que el modelo, más que el código fuente, se convierte en el foco central de la ingeniería del software:

> En UML se crean modelos independientes de plataforma y luego se experimentan varios niveles de transformación para eventualmente devanarse como código fuente para una plataforma específica. Entonces, es lógico que el modelo, no el archivo, deba convertirse en la nueva unidad de salida. Un modelo tiene muchas visiones diferentes en diferentes niveles de abstracción. En el más alto, los componentes independientes de plataforma pueden especificarse en el análisis; en el más bajo, existe una implementación específica a plataforma que se reduce a un conjunto de clases en código.

Novotny argumenta que una nueva generación de herramientas funcionará en conjunción con un repositorio para: crear modelos en todos los niveles necesarios de abstracción, establecer relaciones entre varios modelos, traducir los modelos de un nivel de abstracción a otro (por ejemplo, traducir un modelo de diseño a código fuente), gestionar cambios y versiones, y coordinar las acciones de control y aseguramiento de la calidad en contraste con los modelos de software.

Además de completar los entornos de ingeniería del software, las herramientas de solución puntual que abordan todo, desde recopilación de requerimientos hasta refactorización de diseño/código y pruebas, continuarán evolucionando y se volverán más funcionalmente capaces. En algunas instancias, las herramientas de modelado y pruebas en un dominio de aplicación específico proporcionarán beneficios aumentados cuando se comparen con sus equivalentes genéricos.

31.6 Resumen

Las tendencias que tienen efecto sobre la tecnología de ingeniería del software con frecuencia provienen de las áreas de negocios, organizacional, de mercado y cultural. Dichas "tendencias blandas" pueden guiar la dirección de la investigación y la tecnología que se deriva como consecuencia de la investigación.

Conforme se introduce nueva tecnología, se avanza a través de un ciclo de vida que no siempre conduce a una adopción extensa, aun cuando las expectativas originales sean altas. El grado en el que cualquier tecnología de ingeniería del software gana adopción extensa está ligado a su habilidad para abordar los problemas impuestos por las tendencias blandas y duras.

Las tendencias blandas (la creciente necesidad de conectividad y colaboración, proyectos globales, transferencia de conocimiento, el impacto de las economías emergentes y la influencia de la cultura humana en sí) conducen a un conjunto de retos que abarcan la complejidad administrativa y los requerimientos emergentes, hasta hacer malabares con una mezcla de talentos siempre cambiante entre equipos de software dispersos geográficamente.

Las tendencias duras (el ritmo siempre acelerado del cambio tecnológico) fluyen desde las tendencias blandas y afectan la estructura del software y el ámbito del proceso y la forma en la que se caracteriza un marco conceptual de proceso. El desarrollo colaborativo, nuevas formas de ingeniería de requerimientos, desarrollo basado en modelo e impulsado por pruebas y el diseño posmoderno cambiarán el panorama de los métodos. Los entornos de herramientas responderán a una necesidad creciente de comunicación y colaboración y al mismo tiempo integrarán soluciones puntuales específicas de dominio que pueden cambiar la naturaleza de las actuales tareas de la ingeniería del software.

Problemas y puntos por evaluar

31.1. Obtenga una copia del libro *The Tipping Point*, de Malcolm Gladwell (disponible mediante Google Book Search) y analice cómo se aplican sus teorías a la adopción de nuevas tecnologías en ingeniería del software.

31.2. ¿Por qué el software de mundo abierto presenta un desafío a los enfoques convencionales de la ingeniería del software?

31.3. Revise el *ciclo de promoción excesiva para tecnologías emergentes* del Gartner Group. Seleccione un producto tecnológico muy conocido y presente una breve historia que ilustre cómo viajó a lo largo de la curva. Seleccione otro producto tecnológico muy conocido que no siguió la ruta sugerida por la curva.

31.4. ¿Qué es una "tendencia blanda"?

31.5. Usted se enfrenta con un problema extremadamente complejo que requerirá una solución extensa. ¿Cómo abordaría la complejidad y crearía una respuesta?

31.6. ¿Cuáles son los "requerimientos emergentes" y por qué presentan un reto para los ingenieros de software?

31.7. Seleccione un esfuerzo de desarrollo de fuente abierta (distinto a Linux) y presente una breve historia de su evolución y éxito relativo.

31.8. Describa cómo cree que cambiará el proceso de software durante la próxima década.

31.9. Usted está ubicado en Los Ángeles y trabaja en un equipo de ingeniería de software global. Usted y sus colegas en Londres, Mumbai, Hong Kong y Sidney deben editar una especificación de requerimientos de 245 páginas para un sistema grande. La primera edición debe completarse en tres días. Describa el conjunto ideal de herramientas en línea que le permitirían colaborar de manera efectiva.

31.10. Describa el desarrollo de software impulsado por modelo con sus propias palabras. Haga lo mismo para el desarrollo impulsado por pruebas.

Lecturas y fuentes de información adicionales

Los libros que estudian el camino por venir para el software y la computación abarcan una gran variedad de temas técnicos, científicos, económicos, políticos y sociales. Kurweil (*The Singularity Is Near*, Penguin Books, 2005) presenta una mirada persuasiva de un mundo que cambiará en formas realmente profundas hacia mediados de este siglo. Sterling (*Tomorrow Now*, Random House, 2002) recuerda que el progreso real rara vez es ordenado y eficiente. Teich (*Technology and the Future*, Wadworth, 2002) presenta ensayos concienzudos acerca del impacto social de la tecnología y cómo la cultura cambiante da forma a la tecnología. Naisbitt, Phillips y Naisbitt (*High Tech/High Touch*, Nicholas Brealey, 2001) observan que muchas personas se han "intoxicado" con la alta tecnología y que "la gran ironía de la era de la alta tecnología es que nos estamos volviendo esclavos de los dispositivos que se suponía que nos darían libertad". Zey (*The Future Factor*, McGraw-Hill, 2000) estudia cinco fuerzas que darán forma al destino humano durante este siglo. El de Negroponte (*Being Digital*, Alfred A. Knopf, 1995) fue un *best seller* a mediados de los años de 1990 y continúa ofreciendo una visión comprensiva de la computación y de su impacto global.

Conforme el software se vuelve parte del tejido de virtualmente cada faceta de la vida, la "cibernética" evolucionó como un importante tema de estudio. Los libros de Spinello (*Cyberethics: Morality and Law in Cyberspace*, Jones & Bartlett Publishers, 2002), Halbert y Ingulli (*Cyberethics,* South-Western College Publish-

ers, 2001), y Baird *et al*. (*Cyberethics: Social and Moral Issues in the Computer Age*, Prometheus Books, 2000) consideran el tema con detalle. El gobierno estadounidense publicó un voluminoso reporte en CD-ROM (*21st Century Guide to Cybercrime*, Progressive Management, 2003) que considera todos los aspectos del crimen computacional, los temas de la propiedad intelectual y el National Infraestrutura Protection Center (NIPC, Centro Nacional de Protección a la Infraestructura).

En internet, está disponible una gran variedad de fuentes de información acerca de las direcciones futuras en las tecnologías relacionadas con software y la ingeniería del software. Una lista actualizada de referencias en la World Wide Web que son relevantes para las futuras tendencias en ingeniería del software puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

CAPÍTULO

32

COMENTARIOS FINALES

En los 31 capítulos que precedieron al actual, se exploró un proceso de ingeniería del software que abarca procedimientos administrativos y métodos técnicos, conceptos y principios básicos, técnicas especializadas, actividades orientadas al personal y tareas que son sensibles a la automatización, notación en papel y lápiz, y herramientas de software. Se argumentó que medición, disciplina y un enfoque centrado sobre todo en la agilidad y la calidad darán como resultado software que satisfaga las necesidades del cliente, que sea confiable, mantenible y mejor. Sin embargo, nunca se prometió que la ingeniería del software fuese una panacea.

Conforme se avanza en la segunda década de este nuevo siglo, las tecnologías de software y de sistemas siguen siendo un desafío para todo profesional de software y para toda compañía que construya sistemas basados en computadora. Aunque las siguientes palabras se escribieron con una perspectiva del siglo XX, Max Hopper [Hop90] describe con precisión el estado actual de las cosas:

> Puesto que los cambios en tecnología de la información se vuelven tan rápidos e implacables, y ya que las consecuencias de retrasarse son tan irreversibles, las compañías dominarán la tecnología o morirán... Piense en ello como en una caminadora de tecnología. Las compañías tendrán que correr cada vez más rápido sólo para mantener el paso.

Los cambios en la tecnología de ingeniería del software son de hecho "rápidos e implacables", pero, al mismo tiempo, el progreso con frecuencia es muy lento. Para cuando se toma una decisión a fin de adoptar un nuevo proceso, método o herramienta; realizar la capacitación necesaria para comprender su aplicación e introducir la tecnología en la cultura de desarrollo de software, llega algo más nuevo (e incluso mejor) y el proceso comienza de nuevo.

Una cosa que el autor ha aprendido a través de sus años en este campo es que los profesionales de la ingeniería del software están "conscientes de la moda". El camino por venir estará lleno de cascarones de excitantes nuevas tecnologías (la última moda) que realmente nunca lograron serlo (a pesar de la promoción excesiva). Ese camino se formará con tecnologías más modestas que de alguna manera modificarán la orientación y el ámbito de la vía pública. En el capítulo 31 se estudiaron algunas de ellas.

En este capítulo de conclusiones se tomará una visión más amplia y se considerará dónde se ha estado y a dónde se va desde una perspectiva más filosófica.

UNA MIRADA RÁPIDA

¿Qué es? Conforme se aproxima el final de un viaje relativamente largo a través de la ingeniería del software, es momento de poner las cosas en perspectiva y hacer algunos comentarios finales.

¿Quién lo hace? Autores como el actual. Cuando se llega al final de un largo y desafiante libro es bueno cerrar las cosas en forma significativa.

¿Por qué es importante? Siempre vale la pena recordar dónde se estuvo y considerar hacia dónde se va.

¿Cuáles son los pasos? Se considerará dónde se estuvo y se abordarán algunos temas centrales y orientaciones para el futuro.

¿Cuál es el producto final? Un análisis que le ayudará a entender el gran cuadro.

¿Cómo me aseguro de que lo hice bien? Es difícil de lograr en tiempo real. Sólo después de algunos años podrá decirse si los conceptos, principios, métodos y técnicas de ingeniería del software que se estudiaron en este libro le ayudaron a convertirse en un mejor ingeniero del software.

32.1 La importancia del software-revisión

La importancia del software de computadora puede establecerse de muchas maneras. En el capítulo 1 se caracterizó como un diferenciador. La función que entrega el software diferencia a productos, sistemas y servicios, y ofrece ventaja competitiva en el mercado. Pero el software es más que un diferenciador. Cuando se toma como un todo, los productos operativos de la ingeniería del software generan el artículo más importante que cualquier individuo, negocio o gobierno puede adquirir: información.

En el capítulo 31 se analizó brevemente la computación de mundo abierto (inteligencia ambiental, aplicaciones atentas al contexto y computación predominante/ubicua), una dirección que cambiará de modo fundamental la percepción de las computadoras, las cosas que uno hace con ellas (y las que ellas hacen para uno) y la percepción de la información como guía, producto y necesidad. También se señaló que el software requerido para dar soporte a la computación de mundo abierto presentará nuevos desafíos dramáticos para los ingenieros del software. Pero, mucho más importante, la penetración venidera del software de computadora presentará retos más dramáticos para la sociedad como un todo. Siempre que una tecnología tiene amplio impacto (un impacto que puede salvar vidas o ponerlas en peligro, construir empresas o destruirlas, informar o malinformar a los líderes del gobierno), debe "manejarse con cuidado".

32.2 Las personas y la forma en la que construyen sistemas

El software requerido para sistemas de alta tecnología se vuelve más complejo con cada año que transcurre y el tamaño de los programas resultantes aumenta de manera proporcional. El rápido crecimiento en tamaño del programa "promedio" presentaría algunos problemas si no fuese por un simple hecho: conforme aumenta el tamaño del programa, el número de personas que deben trabajar en el programa también debe aumentar.

Cita:

"El shock del futuro [es] la aplastante tensión y desorientación que inducimos en los individuos al sujetarlos a demasiado cambio en un periodo muy corto."

Alvin Toffler

La experiencia indica que, conforme aumenta el número de personas de un equipo de proyecto de software, la productividad global del grupo puede disminuir. Una forma de resolver este problema es crear algunos equipos de ingeniería del software, lo que, por tanto, divide al personal en grupos de trabajo menores. Sin embargo, conforme crece el número de equipos de ingeniería del software, la comunicación entre ellos se vuelve tan difícil y consumidora de tiempo como la comunicación entre individuos. Peor aún, la comunicación (entre individuos o equipos) tiende a ser ineficiente, es decir, se emplea mucho tiempo transfiriendo muy poco contenido de información y también, con mucha frecuencia, la información importante "cae en las grietas".

Si la comunidad de ingeniería del software ha de lidiar de manera efectiva con el dilema de la comunicación, el camino por delante para los ingenieros del software debe incluir cambios radicales en la forma en la que los individuos y los equipos se comunican entre ellos. En el capítulo 31 se estudiaron los entornos colaboradores que pueden proporcionar mejorías dramáticas en las formas en las que se comunican los equipos.

En última instancia, la comunicación es la transferencia de conocimiento, y la adquisición (y transferencia) de conocimiento representa cambiar profundamente. Conforme los motores de búsqueda se vuelven cada vez más sofisticados y las aplicaciones Web 2.0 ofrecen mejor sinergia, la biblioteca más grande del mundo de artículos y reportes, tutoriales, comentarios y referencias de investigación se vuelve más accesible y utilizable.

Si la historia pasada es un indicio, es justo decir que las personas no cambiarán. Sin embargo, las formas en las que se comunican, el entorno en el que trabajan, la forma en la que adquieren conocimiento, los métodos y herramientas que usan, la disciplina que aplican y, en consecuencia, la cultura global para el desarrollo del software cambiará significativa e, incluso, profundamente.

32.3 NUEVOS MODOS PARA REPRESENTAR LA INFORMACIÓN

Cita:

"La mejor preparación para un buen trabajo mañana es hacer un buen trabajo hoy."

Elbert Hubbard

A través de la historia de la computación, ha ocurrido una transición sutil en la terminología que se usa para describir el trabajo de desarrollo del software que realiza la comunidad empresarial. Hace 40 años, el término *procesamiento de datos* era la frase operativa que describía el uso de las computadoras en un contexto empresarial. En la actualidad, el procesamiento de datos originó otra frase, *tecnología de la información*, que implica lo mismo pero con un cambio sutil de enfoque. El énfasis ya no es simplemente procesar grandes cantidades de datos, sino, más bien, extraer información significativa de estos datos. Obviamente, ésta siempre fue la intención, pero el cambio en la terminología refleja una modificación mucho más importante en la filosofía administrativa.

Cuando hoy se estudian aplicaciones de software, las palabras *datos*, *información* y *contenido* ocurren repetidamente. La palabra *conocimiento* se encuentra en algunas aplicaciones de inteligencia artificial, pero su uso es relativamente raro. Virtualmente, nadie discute *sabiduría* en el contexto de aplicaciones de software.

Los datos son información bruta: colecciones de hechos que deben procesarse para ser significativas. La información se entrega al asociar hechos dentro de un contexto determinado. El conocimiento asocia información obtenida en un contexto con otra información obtenida en un contexto diferente. Finalmente, la sabiduría ocurre cuando se derivan principios generalizados a partir de conocimiento dispar. Cada una de estas cuatro visiones de "información" se representa de manera esquemática en la figura 32.1.

Cita:

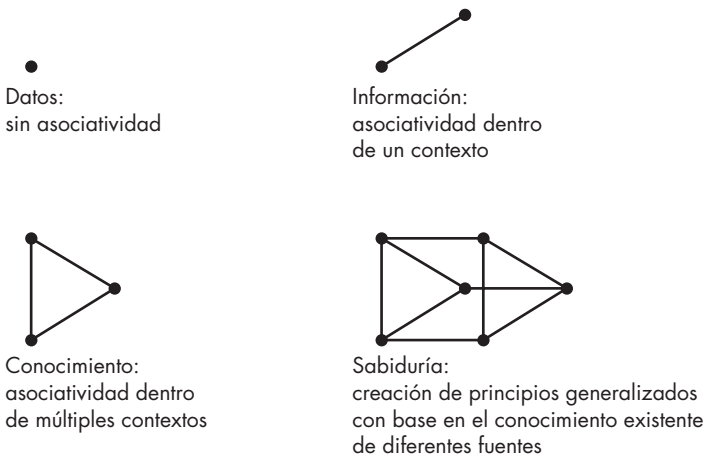
"La sabiduría es el poder que nos permite usar el conocimiento para beneficio de nosotros mismos y de los demás."

Thomas J. Watson

A la fecha, la gran mayoría del software se construyó para procesar datos o información. Los ingenieros de software ahora están igualmente preocupados con los sistemas que procesan conocimiento.[1] El conocimiento es bidimensional. La información recopilada acerca de varios temas relacionados y no relacionados se conecta para formar un cuerpo de hechos que se llama *conocimiento*. La clave es la habilidad para asociar información de una variedad de fuentes diferentes, que pueden no tener alguna conexión obvia, y para combinarla de manera que ofrezca algún beneficio diferente.[2]

Para ilustrar la progresión que conduce desde datos hasta conocimiento, considere los datos censales que indican que la tasa de natalidad en 1996 en Estados Unidos fue de 4.9 millones.

FIGURA 32.1
Un espectro de "información"



1 El rápido crecimiento de tecnologías de minado y de data warehouses refleja esta tendencia creciente.

2 La semántica web (Web 2.0) permite la creación de "purés" que pueden proporcionar un mecanismo sencillo para lograr esto.

Este número representa un valor de datos. Al relacionar este trozo de datos con las tasas de natalidad de los 40 años anteriores, puede inferir una pieza de información útil: los *baby boomers* de los años cincuenta y principios de los sesenta del siglo pasado, que hoy envejecen, hicieron un último esfuerzo para tener hijos antes del final de sus años de crianza infantil. Además, los de la generación X comenzaron sus años de crianza infantil. Los datos censales pueden conectarse entonces a otras piezas de información aparentemente no relacionadas. Por ejemplo, el número actual de profesores de enseñanza básica que se retirarán durante la próxima década, el número de estudiantes universitarios que se gradúan en educación primaria y secundaria, la presión sobre los políticos para mantener bajos los impuestos y, por tanto, limitar los aumentos de sueldo para los profesores. Todas estas piezas de información pueden combinarse para formular una representación de conocimiento: habrá una presión significativa sobre el sistema educativo de Estados Unidos a principios del siglo XXI y continuará durante algunas décadas. Al usar este conocimiento puede surgir una oportunidad de negocios. Puede haber una significativa oportunidad para desarrollar nuevos modos de aprendizaje que sean más efectivos y menos costosos que los enfoques actuales.

El camino futuro para el software conduce a sistemas que procesan el conocimiento. Durante más de 50 años se han procesado datos usando computadoras, y durante más de tres décadas se ha extraído información. Uno de los retos más significativos que enfrenta la comunidad de ingeniería del software es construir sistemas que den el siguiente paso en el espectro: sistemas que extraigan conocimiento a partir de datos e información, de manera práctica y benéfica.

32.4 La vista larga

En la sección 32.3 se sugirió que el camino por venir conduce a sistemas que "procesen conocimiento". Pero el futuro de la computación en general y de los sistemas basados en software en particular puede conducir a eventos que sean considerablemente más profundos.

En un libro fascinante que debe leer toda persona involucrada en las tecnologías de la computación, Ray Kurzweil [Kur05] sugiere que se ha llegado a un momento en el que "el ritmo del cambio tecnológico será tan rápido, y su impacto tan profundo, que la vida humana será transformada de manera irreversible". Kurzweil[3] plantea un argumento de peso: la humanidad actualmente está en la "rodilla" de una curva de crecimiento exponencial que conducirá a un enorme incremento en la capacidad de computación durante las siguientes dos décadas. Cuando se acople con avances equivalentes en nanotecnología, genética y robótica, será posible aproximarse a un momento, a mediados de este siglo, donde la distinción entre humanos (como se les conoce hoy día) y máquinas comience a empañarse: un momento en el que la evolución humana se acelere en formas que son tanto atemorizantes (para algunos) como espectaculares (para otros).

Kurzweil argumenta que, en algún momento en la década de 2030, la capacidad de computación y el software requerido serán suficientes para modelar cada aspecto del cerebro humano: todo, desde conexiones físicas, procesos analógicos y recubrimientos químicos. Cuando esto ocurra, los seres humanos habrán logrado "una elevada IA" (inteligencia artificial) y, como consecuencia, máquinas que realmente piensen (dentro del uso convencional actual de la palabra). Pero habrá una diferencia fundamental. Los procesos del cerebro humano son excesivamente complejos y sólo se conectan de manera holgada con las fuentes de información externas. También son computacionalmente lentos, incluso en comparación con la tecnología de compu-

3 Es importante notar que Kurzweil no es un escritor de ciencia ficción ordinario, o un futurólogo sin cartera. Es un tecnólogo serio que (según Wikipedia) "fue pionero en los campos de reconocimiento óptico de caracteres (OCR), síntesis de texto a habla, tecnología de reconocimiento de voz e instrumentos de teclado electrónico".

tación actual. Cuando ocurra la simulación completa del cerebro humano, el "pensamiento" ocurrirá con rapideces miles de veces mayores que su contraparte humana, con íntimas conexiones a un mar de información (piense en la web de la actualidad como un ejemplo primitivo). El resultado es... bueno... tan fantástico que es mejor dejar a Kurzweil describirlo.

Es importante notar que nadie cree que el futuro que Kurzweil describe sea algo bueno. En un ensayo ahora famoso, titulado "The Future Doesn't Need Us" (El futuro no nos necesita), Bill Joy [Joy00], uno de los fundadores de Sun Microsystems, argumenta que "robótica, ingeniería genética y nanotecnología amenazan con hacer a los humanos una especie en peligro de extinción". Sus argumentos, que predicen una distopia tecnológica, representan un contrapunto al futuro utópico predicho por Kurzweil. Ambos deben considerarse seriamente como ingenieros de software que juegan uno de los papeles principales en la definición de la visión larga de la especie humana.

32.5 La responsabilidad del ingeniero de software

La ingeniería del software ha evolucionado en una profesión mundial respetada. Como profesionales, los ingenieros de software deben acatar un código de ética que guíe el trabajo que realizan y los productos que elaboran. Una fuerza de trabajo conjunta ACM/IEEE-CS produjo un *Código de ética y práctica profesional de la ingeniería del software* (versión 5.1). El código [ACM98] afirma:

> Los ingenieros de software deben comprometerse con hacer del análisis, la especificación, el diseño, el desarrollo, la prueba y el mantenimiento del software una profesión benéfica y respetada. En concordancia con su compromiso con la salud, la seguridad y el bienestar del público, los ingenieros de software deben adherirse a los siguientes ocho principios:
>
> 1. PÚBLICO: Los ingenieros del software deben actuar consistentemente con el interés del público.
> 2. CLIENTE Y EMPLEADOR: Los ingenieros de software deben actuar en función del mejor interés de sus clientes y empleadores, coincidente con el interés del público.
> 3. PRODUCTO: Los ingenieros de software deben garantizar que sus productos y modificaciones relacionadas satisfagan los más altos estándares profesionales posibles.
> 4. JUICIO: Los ingenieros de software deben mantener integridad e independencia en su juicio profesional.
> 5. ADMINISTRACIÓN: Los administradores y líderes de ingeniería del software deben suscribirse y promover un enfoque ético acerca de la administración del desarrollo y del mantenimiento del software.
> 6. PROFESIÓN: Los ingenieros de software deben promover la integridad y reputación de la profesión, consistente con el interés del público.
> 7. COLEGAS: Los ingenieros de software deben ser justos con sus colegas y darles apoyo.
> 8. UNO MISMO: Los ingenieros de software deben aprender toda la práctica de su profesión y deben promover un enfoque ético acerca de ella.

WebRef

Un análisis completo del código de ética de ACM/IEEE puede encontrarse en **seeri.etsu.edu/Codes/default.shtm**

Aunque cada uno de estos ocho principios es igualmente importante, aparece un tema sensible: un ingeniero de software necesita trabajar en función del interés del público. En el nivel personal, un ingeniero de software debe conducirse de acuerdo con las siguientes reglas:

- Nunca robar datos para ganancia personal.
- Nunca distribuir o vender información con derechos de autor obtenida como parte de su trabajo en un proyecto de software.
- Nunca destruir o modificar maliciosamente los programas, archivos o datos de otra persona.

- Nunca violar la privacidad de un individuo, grupo u organización.
- Nunca *hackear* un sistema por deporte o beneficio.
- Nunca crear o propagar un virus o gusano.
- Nunca usar tecnología de computación para facilitar la discriminación o el hostigamiento.

Durante la década pasada, ciertos miembros de la industria del software acudieron a las autoridades en busca de legislación protectora que [SEE03]: 1) permita a las compañías liberar software sin revelar defectos conocidos, 2) exentar a los desarrolladores de responsabilidad por cualquier daño que resulte de dichos defectos conocidos, 3) restringir a otros en revelar defectos sin permiso del desarrollador original, 4) permitir la incorporación de software de "autoayuda" dentro de un producto que pueda deshabilitar (mediante comando remoto) la operación del producto y 5) exentar a los desarrolladores de software con "autoayuda" de daños en caso de que el software sea deshabilitado por una tercera persona.

Como toda legislación, el debate frecuentemente se centra en temas que son políticos, no tecnológicos. Sin embargo, muchas personas (incluido el autor) creen que la legislación protectora, si se emite de manera inadecuada, entra en conflicto con el código de ética de la ingeniería del software al exentar de manera indirecta a los ingenieros de software de su responsabilidad de producir software de alta calidad.

32.6 Un comentario final

Hace 30 años comenzó el trabajo en la primera edición de este libro. El autor todavía se recuerda sentado en su escritorio como joven profesor, escribiendo el manuscrito para un libro acerca de una materia por la que pocas personas se preocupaban e incluso aún menos entendían realmente. Recuerda las cartas de rechazo de los editores, quienes argumentaban (cortés, pero firmemente) que nunca habría un mercado para un libro acerca de "ingeniería del software". Por fortuna, McGraw-Hill decidió darle una oportunidad,[4] y el resto, como dicen, es historia.

Durante los pasados 30 años, este libro cambió dramáticamente: en visión, en tamaño, en estilo y en contenido. Como la ingeniería del software, creció y (con fortuna) maduró con los años.

Un enfoque de ingeniería centrado en el desarrollo del software de computadora ahora es sabiduría convencional. Aunque el debate continúa acerca del "paradigma correcto", la importancia de la agilidad, el grado de automatización y los métodos más efectivos, los principios subyacentes de la ingeniería del software, ahora se aceptan en toda la industria. ¿Por qué, entonces, se ha visto su adopción amplia sólo recientemente?

La respuesta, acaso, se encuentra en la dificultad de la transición tecnológica y el cambio cultural que la acompaña. Aun cuando la mayoría de las personas aprecian la necesidad de una disciplina de ingeniería para el software, se lucha contra la inercia de la práctica pasada y se enfrentan nuevos dominios de aplicación (y de los desarrolladores que los trabajan) que parecen listos a repetir los errores del pasado. Para facilitar la transición se necesitan muchas cosas: un proceso de software ágil, adaptable y sensible; métodos más efectivos; herramientas más poderosas; mejor aceptación por parte de los profesionales y apoyo de los administradores; y no pequeñas dosis de educación.

Acaso el lector no esté de acuerdo con todos los enfoques descritos en este libro. Algunas de las técnicas y opiniones son controvertidas; otras deben afinarse para trabajar bien en diferentes

4 En realidad, el crédito debe ir para Peter Freeman y Eric Munson, quienes convencieron a McGraw-Hill de que valía la pena probar. Más de un millón de copias después, es justo decir que tomaron una buena decisión.

entornos de desarrollo de software. Sin embargo, el autor desea sinceramente que *Ingeniería del software. Un enfoque práctico* haya delineado el problema que se enfrenta, demostrado la fuerza de los conceptos de la ingeniería del software y ofrecido un marco conceptual de métodos y herramientas.

Conforme se avanza aún más en el siglo XXI, el software sigue siendo el producto y la industria más importantes en la escena mundial. Su impacto e importancia han recorrido un largo camino. Y, sin embargo, una nueva generación de desarrolladores de software debe satisfacer muchos de los mismos desafíos que enfrentaron las generaciones anteriores. Con la esperanza de que las personas que enfrenten el reto, ingenieros del software, tendrán la sabiduría para desarrollar sistemas que mejoren la condición humana.

APÉNDICE 1

INTRODUCCIÓN A UML[1]

CONCEPTOS CLAVE

El *Lenguaje de Modelado Unificado* (UML) es "un lenguaje estándar para escribir diseños de software. El UML puede usarse para visualizar, especificar, construir y documentar los artefactos de un sistema de software intensivo" [Boo05]. En otras palabras, tal como los arquitectos de edificios crean planos para que los use una compañía constructora, los arquitectos de software crean diagramas de UML para ayudar a los desarrolladores de software a construir el software. Si usted entiende el vocabulario del UML (los elementos pictóricos de los diagramas y su significado) puede comprender y especificar con mucha más facilidad un sistema, y explicar su diseño a otros.

Grady Booch, Jim Rumbaugh e Ivar Jacobson desarrollaron el UML a mediados de los años noventa del siglo pasado con mucha realimentación de la comunidad de desarrollo de software. El UML fusionó algunas notaciones de modelado que competían entre sí y que se usaban en la industria del software en la época. En 1997, UML 1.0 se envió al Object Management Group, un consorcio sin fines de lucro involucrado en especificaciones de mantenimiento para su empleo en la industria de la computación. El UML 1.0 se revisó y dio como resultado la adopción del UML 1.1 ese mismo año. El estándar actual es UML 2.0 y ahora es un estándar ISO. Puesto que este estándar es tan nuevo, muchas antiguas referencias, como [Gam95], no usan notación de UML.

UML 2.0 proporciona 13 diferentes diagramas para su uso en modelado de software. En este apéndice se analizarán solamente diagramas de *clase, implementación, caso de uso, secuencia, comunicación, actividad* y *estado*, diagramas que se usan en esta edición de *Ingeniería del software. Un enfoque práctico*.

Debe observar que existen muchas características opcionales en diagramas de UML. El UML ofrece dichas opciones (en ocasiones complejas) de modo que pueda expresar todos los aspectos importantes de un sistema. Al mismo tiempo, tiene la flexibilidad para suprimir aquellas partes del diagrama que no son relevantes para el aspecto que se va a modelar, con la finalidad de evitar confundir el diagrama con detalles irrelevantes. Por tanto, la omisión de una característica particular no significa que ésta se encuentre ausente; puede significar que la característica se suprimió. En este apéndice, *no* se presenta la cobertura exhaustiva de todas las características de los diagramas de UML. El apéndice se enfocará en las opciones estándar, en especial en aquellas que se usaron en este libro.

DIAGRAMAS DE CLASE

Para modelar clases, incluidos sus atributos, operaciones, relaciones y asociaciones con otras clases,[2] el UML proporciona un *diagrama de clase*, que aporta una visión estática o de estructura de un sistema, sin mostrar la naturaleza dinámica de las comunicaciones entre los objetos de las clases.

1 Este apéndice fue una aportación de Dale Skrien y se adaptó de su libro *An Introduction to Object-Oriented Design and Design Patterns in Java* (McGraw-Hill, 2008). Todo el contenido se usa con permiso.

2 Si el lector no está familiarizado con los conceptos orientados a objeto, en el apéndice 2 se presenta una breve introducción.

Los elementos principales de un diagrama de clase son cajas, que son los íconos utilizados para representar clases e interfaces. Cada caja se divide en partes horizontales. La parte superior contiene el nombre de la clase. La sección media menciona sus atributos. Un *atributo* es algo que un objeto de dicha clase conoce o puede proporcionar todo el tiempo. Por lo general, los atributos se implementan como campos de la clase, pero no necesitan serlo. Podrían ser valores que la clase puede calcular a partir de sus variables o valores instancia y que puede obtener de otros objetos de los cuales está compuesto. Por ejemplo, un objeto puede conocer siempre la hora actual y regresarla siempre que se le solicite. Por tanto, sería adecuado mencionar la hora actual como un atributo de dicha clase de objetos. Sin embargo, el objeto muy probablemente no tendría dicha hora almacenada en una de sus variables instancia, porque necesitaría actualizar de manera continua ese campo. En vez de ello, el objeto probablemente calcularía la hora actual (por ejemplo, a través de consulta con objetos de otras clases) en el momento en el que se le solicite la hora. La tercera sección del diagrama de clase contiene las operaciones o comportamientos de la clase. Una *operación* es lo que pueden hacer los objetos de la clase. Por lo general, se implementa como un método de la clase.

La figura A1.1 presenta un ejemplo simple de una clase **Thoroughbred** (pura sangre) que modela caballos de pura sangre. Muestra tres atributos: **mother** (madre), **father** (padre) y **birthyear** (año de nacimiento). El diagrama también muestra tres operaciones: *getCurrentAge()* (obtener edad actual), *getFather()* (obtener padre) y *getMother()* (obtener madre); puede haber otros atributos y operaciones suprimidos que no se muestren en el diagrama.

Cada atributo puede tener un nombre, un tipo y un nivel de visibilidad. El tipo y la visibilidad son opcionales. El tipo sigue al nombre y se separa de él mediante dos puntos. La visibilidad se indica mediante un –, #, ~ o + precedente, que indica, respectivamente, visibilidad *privada*, *protegida*, *paquete* o *pública*. En la figura A1.1, todos los atributos tienen visibilidad privada, como se indica mediante el signo menos que los antecede (–). También es posible especificar que un atributo es estático o de clase, subrayándolo. Cada operación puede desplegarse con un nivel de visibilidad, parámetros con nombres y tipos, y un tipo de retorno.

Una clase abstracta o un método abstracto se indica con el uso de cursivas en el nombre del diagrama de clase. Vea, por ejemplo, la clase **Horse** (caballo) en la figura A1.2. Una interfaz se indica con la frase "<<interface>>" (llamada *estereotipo*) arriba del nombre. Vea la interfaz **OwnedObject** (objeto posesión) en la figura A1.2. Una interfaz también puede representarse gráficamente mediante un círculo hueco.

Vale la pena mencionar que el ícono que representa una clase puede tener otras partes opcionales. Por ejemplo, puede usarse una cuarta sección en el fondo de la caja de clase para mencionar las responsabilidades de la clase. Esta sección es particularmente útil cuando se realiza la transición de tarjetas CRC (capítulo 6) a diagramas de clase, donde las responsabilidades mencionadas en las tarjetas CRC pueden agregarse a esta cuarta sección en la caja de clase en el diagrama UML antes de crear los atributos y operaciones que llevan a cabo dichas responsabilidades. Esta cuarta sección no se muestra en ninguna de las figuras de este apéndice.

FIGURA A1.1
Diagrama de clase para una clase Thoroughbred

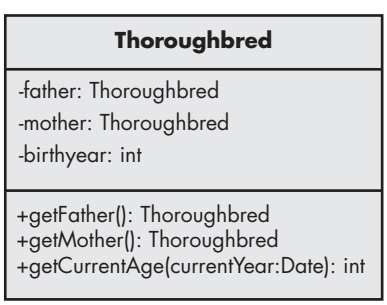
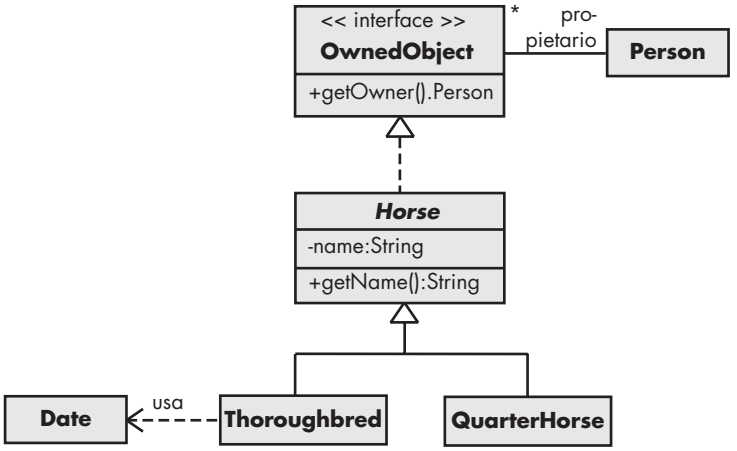


FIGURA A1.2
Diagrama de clase concerniente a caballos



Los diagramas de clase también pueden mostrar relaciones entre clases. Una clase que sea una subclase de otra clase se conecta con ella mediante una flecha con una línea sólida y con una punta triangular hueca. La flecha apunta de la subclase a la superclase. En UML, tal relación se llama *generalización*. Por ejemplo, en la figura A1.2, las clases **Thoroughbred** y **QuarterHorse** (caballo cuarto de milla) se muestran como subclases de la clase abstracta **Horse**. Una flecha con una línea punteada indica implementación de una interfaz. En UML, tal relación se llama *realización*. Por ejemplo, en la figura A1.2, la clase **Horse** implementa o realiza la interfaz **OwnedObject**.

Una *asociación* entre dos clases significa que existe una relación estructural entre ellas. Las asociaciones se representan mediante líneas sólidas. Una asociación tiene muchas partes opcionales. Puede etiquetarse, así como cada una de sus terminaciones, para indicar el papel de cada clase en la asociación. Por ejemplo, en la figura A1.2, existe una asociación entre **OwnedObject** y **Person**, en la que **Person** juega el papel de owner (propietario). Las flechas en cualquiera o en ambos lados de una línea de asociación indican navegabilidad. Además, cada extremo de la línea de asociación puede tener un valor de multiplicidad desplegado. Navegabilidad y multiplicidad se explican con más detalle más adelante, en esta sección. Una asociación también puede conectar una clase consigo misma, mediante un bucle. Tal asociación indica la conexión de un objeto de la clase con otros objetos de la misma clase.

Una asociación con una flecha en un extremo indica navegabilidad en un sentido. La flecha significa que, desde una clase, es posible acceder con facilidad a la segunda clase asociada hacia la que apunta la asociación; sin embargo, desde la segunda clase, no necesariamente puede accederse con facilidad a la primera clase. Otra forma de pensar en esto es que la primera clase está al tanto de la segunda, pero el segundo objeto de clase no necesariamente está directamente al tanto de la primera clase. Una asociación sin flechas por lo general indica una asociación de dos vías, que es lo que se pretende en la figura A1.2; pero también simplemente podría significar que la navegabilidad no es importante y, por tanto, que queda fuera.

Debe observarse que un atributo de una clase es muy parecido a una asociación de la clase con el tipo de clase del atributo. Es decir, para indicar que una clase tiene una propiedad llamada "name" (nombre) de tipo String, podría desplegarse dicha propiedad como atributo, como en la clase **Horse** en la figura A1.2. De manera alternativa, podría crearse una asociación de una vía desde la clase **Horse** hasta la clase **String** donde el papel de la clase String es "name". El enfoque de atributo es mejor para tipos de datos primitivos, mientras que el enfoque de asociación con frecuencia es mejor si la clase propietaria juega un papel principal en el diseño, en cuyo caso es valioso tener una caja de clase para dicho tipo.

Una relación de *dependencia* representa otra conexión entre clases y se indica mediante una línea punteada (con flechas opcionales en los extremos y con etiquetas opcionales). Una clase depende de otra si los cambios en la segunda clase pueden requerir cambios en la primera. Una asociación de una clase con otra automáticamente indica una dependencia. No se necesitan líneas punteadas entre clases si ya existe una asociación entre ellas. Sin embargo, para una relación transitoria (es decir, una clase que no mantiene alguna conexión de largo plazo con otra, sino que usa dicha clase de manera ocasional), debe dibujarse una línea punteada desde la primera clase hasta la segunda. Por ejemplo, en la figura A1.2, la clase **Thoroughbred** usa la clase **Date** (fecha) siempre que se invoca su método *getCurrentAge()*; por eso la dependencia se etiqueta "usa".

La *multiplicidad* de un extremo de una asociación significa el número de objetos de dicha clase que se asocia con la otra clase. Una multiplicidad se especifica mediante un entero no negativo o mediante un rango de enteros. Una multiplicidad especificada por "0..1" significa que existen 0 o 1 objetos en dicho extremo de la asociación. Por ejemplo, cada persona en el mundo tiene un número de seguridad social o no lo tiene (especialmente si no son ciudadanos estadounidenses); por tanto, una multiplicidad de 0..1 podría usarse en una asociación entre una clase **Person** y una clase **SocialSecurityNumber** (número de seguridad social) en un diagrama de clase. Una multiplicidad que se especifica como "1...*" significa uno o más, y una multiplicidad especificada como "0...*" o sólo "*" significa cero o más. Un * se usa como la multiplicidad en el extremo **OwnedObject** de la asociación con la clase **Person** en la figura A1.2, porque una **Person** puede poseer cero o más objetos.

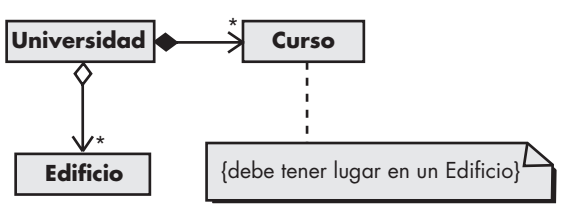
Si un extremo de una asociación tiene multiplicidad mayor que 1, entonces los objetos de la clase a la que se refiere en dicho extremo de la asociación probablemente se almacenan en una colección, como un conjunto o lista ordenada. También podría incluirse dicha clase colección en sí misma en el diagrama UML, pero tal clase por lo general queda fuera y se supone, de manera implícita, que está ahí debido a la multiplicidad de la asociación.

Una *agregación* es un tipo especial de asociación que se indica mediante un diamante hueco en un extremo del ícono. Ello indica una relación "entero/parte", en la que la clase a la que apunta la flecha se considera como una "parte" de la clase en el extremo diamante de la asociación. Una *composición* es una agregación que indica fuerte propiedad de las partes. En una composición, las partes viven y mueren con el propietario porque no tienen papel en el sistema de software independiente del propietario. Vea la figura A1.3 para ejemplos de agregación y composición.

Una **Universidad** tiene una agregación de objetos **Edificio**, que representan los edificios que constituyen el campus. La universidad también tiene una colección de cursos. Si la universidad quebrara, los edificios todavía existirían (si se supone que la universidad no se destruye físicamente) y podría usar otras cosas, pero un objeto **Curso** no tiene uso fuera de la universidad en la cual se ofrece. Si la universidad deja de existir como una entidad empresarial, el objeto **Curso** ya no sería útil y, por tanto, también dejaría de existir.

Otro elemento común de un diagrama de clase es una *nota*, que se representa mediante una caja con una esquina doblada y se conecta a otros íconos mediante una línea punteada. Puede

FIGURA A1.3
Relación entre universidades, cursos y edificios



tener contenido arbitrario (texto y gráficos) y es similar a comentarios en lenguajes de programación. Puede contener comentarios acerca del papel de una clase o restricciones que todos los objetos de dicha clase deban satisfacer. Si los contenidos son una restricción, se encierran entre llaves. Observe la restricción unida a la clase **Curso** en la figura A1.3.

Diagramas de implementación

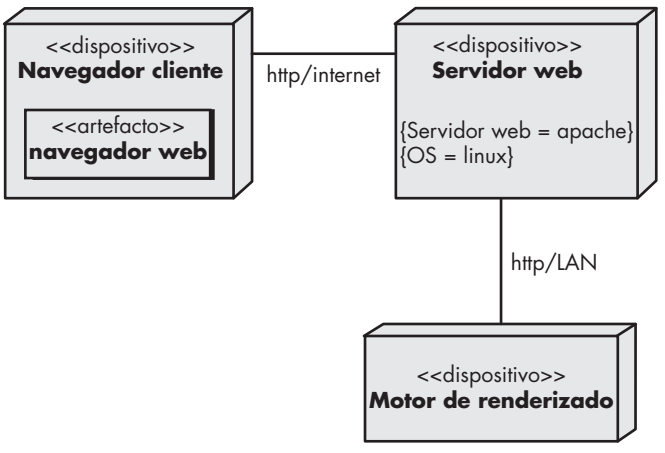
Un *diagrama de implementación* UML se enfoca en la estructura de un sistema de software y es útil para mostrar la distribución física de un sistema de software entre plataformas de hardware y entornos de ejecución. Suponga, por ejemplo, que desarrolla un paquete de renderizado de gráficos basado en web. Los usuarios de su paquete usarán su navegador web para ir a su sitio web e ingresar la información que se va a renderizar. Su sitio web renderizaría una imagen gráfica de acuerdo con la especificación del usuario y la enviaría de vuelta al usuario. Puesto que las gráficas renderizadas pueden ser computacionalmente costosas, usted decide mover el renderizado afuera del servidor web y hacia una plataforma separada. Por tanto, habrá tres dispositivos de hardware involucrados en su sistema: el cliente web (la computadora que corre un navegador del usuario), la computadora que alberga el servidor web y la computadora que alberga el motor de renderizado.

La figura A1.4 muestra el diagrama de implementación para tal paquete. En ese diagrama, los componentes de hardware se dibujan en cajas marcadas con "<<dispositivo>>". Las rutas de comunicación entre componentes de hardware se dibujan con líneas con etiquetas opcionales. En la figura A1.4, las rutas se etiquetan con el protocolo de comunicación y con el tipo de red utilizado para conectar los dispositivos.

Cada nodo que hay en un diagrama de implementación también puede anotarse con detalles del dispositivo. Por ejemplo, en la figura A1.4 se ilustra el navegador cliente para mostrar que contiene un artefacto que consiste en el software del navegador web. Un artefacto por lo general es un archivo que contiene software que corre en un dispositivo. También puede especificar valores etiquetados, como se muestra en la figura A1.4 en el nodo del servidor web. Dichos valores definen al proveedor del servidor web y al sistema operativo que usa el servidor.

Los diagramas de implementación también pueden mostrar nodos de entorno de ejecución, que se dibujan como cajas que contienen la etiqueta "<<entorno de ejecución>>". Dichos nodos representan sistemas, como sistemas operativos, que pueden albergar otro software.

Figura A1.4
Diagrama de implementación



Diagramas de uso de caso

Los casos de uso (capítulos 5 y 6) y el *diagrama de uso de caso* UML ayudan a determinar la funcionalidad y características del software desde la perspectiva del usuario. Para proporcionarle una aproximación a la manera en la que funcionan los casos de uso y los diagramas de uso de caso, se crearán algunos para una aplicación de software que gestiona archivos de música digital, similar al software iTunes de Apple. Algunas de las cosas que puede incluir el software son funciones para:

- Descargar un archivo de música MP3 y almacenarlo en la biblioteca de la aplicación.
- Capturar música de streaming (transmisión continua) y almacenarla en la biblioteca de la aplicación.
- Gestionar la biblioteca de la aplicación (por ejemplo, borrar canciones u organizarlas en listas de reproducción).
- Quemar en CD una lista de las canciones de la biblioteca.
- Cargar una lista de las canciones de la biblioteca en un iPod o reproductor MP3.
- Convertir una canción de formato MP3 a formato AAC y viceversa.

Ésta no es una lista exhaustiva, pero es suficiente para entender el papel de los casos de uso y los diagramas de uso de caso.

Un *caso de uso* describe la manera en la que un usuario interactúa con el sistema, definiendo los pasos requeridos para lograr una meta específica (por ejemplo, quemar una lista de canciones en un CD). Las variaciones en la secuencia de pasos describen varios escenarios (por ejemplo, ¿y si todas las canciones de la lista no caben en un CD?).

Un diagrama UML de uso de caso es un panorama de todos los casos de uso y sus relaciones. El mismo proporciona un gran cuadro de la funcionalidad del sistema. En la figura A1.5 se muestra un diagrama de uso de caso para la aplicación de música digital.

En este diagrama, la figura de palitos representa a un *actor* (capítulo 5) que se asocia con una categoría de usuario (u otro elemento de interacción). Por lo general, los sistemas complejos tienen más de un actor. Por ejemplo, una aplicación de máquina expendedora puede tener tres actores que representan clientes, personal de reparación y proveedores que rellenan la máquina.

En el diagrama de uso de caso, los casos de uso se muestran como óvalos. Los actores se conectan mediante líneas a los casos de uso que realizan. Observe que ninguno de los detalles de los casos de uso se incluye en el diagrama y, en vez de ello, necesita almacenarse por separado. Observe también que los casos de uso se colocan en un rectángulo, pero los actores no. Este rectángulo es un recordatorio visual de las fronteras del sistema y de que los actores están afuera del sistema.

Algunos casos de uso en un sistema pueden relacionarse mutuamente. Por ejemplo, existen pasos similares al de quemar una lista de canciones en un CD y cargar una lista de canciones en un iPod. En ambos casos, el usuario crea primero una lista vacía y luego agrega las canciones de la biblioteca a la lista. Para evitar duplicación en casos de uso, por lo general es mejor crear un nuevo caso de uso que represente la actividad duplicada y luego dejar que los otros casos de uso incluyan este nuevo caso de uso como uno de sus pasos. Tal inclusión se indica en los diagramas de uso de caso, como en la figura A1.6, mediante una flecha punteada etiquetada como “incluye”, que conecta un caso de uso con un caso de uso incluido.

Dado que despliega todos los casos de uso, un diagrama de uso de caso es un auxiliar útil para asegurar que cubrió toda la funcionalidad del sistema. En el organizador de música digital, seguramente querría más casos de uso, tal como uno para reproducir una canción de la biblioteca. Pero tenga en mente que la contribución más valiosa de casos de uso al proceso de desa-

FIGURA A1.5

Diagrama de caso de uso para el sistema de música

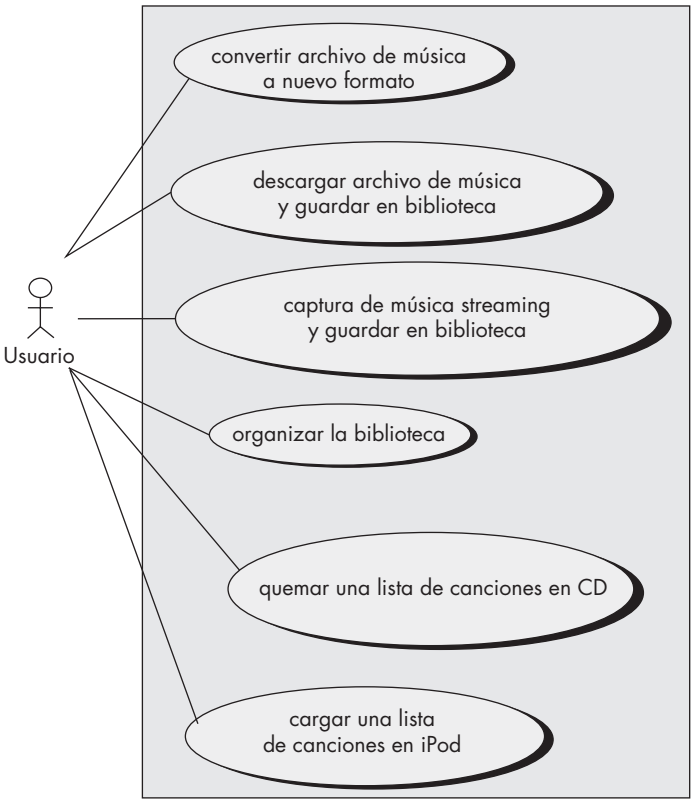
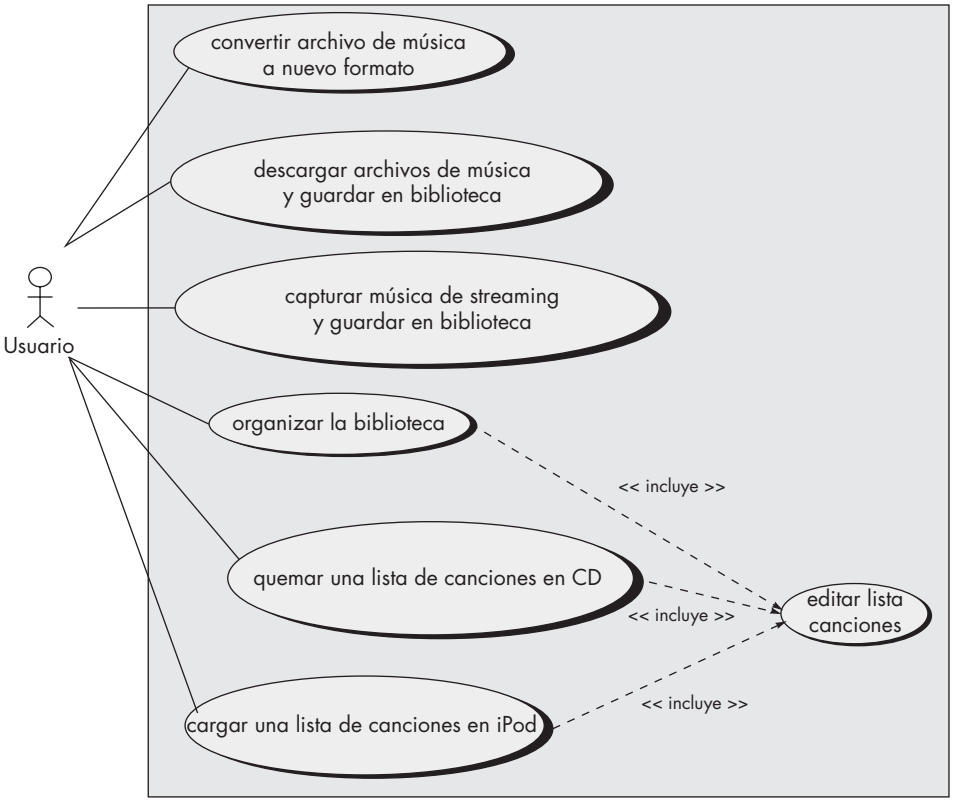


FIGURA A1.6

Diagrama de caso de uso con casos de uso incluidos



rrollo de software es la descripción textual de cada caso de uso, no el diagrama de uso de caso global [Fow04b]. Es a través de las descripciones que usted puede tener una comprensión clara de las metas del sistema que desarrolla.

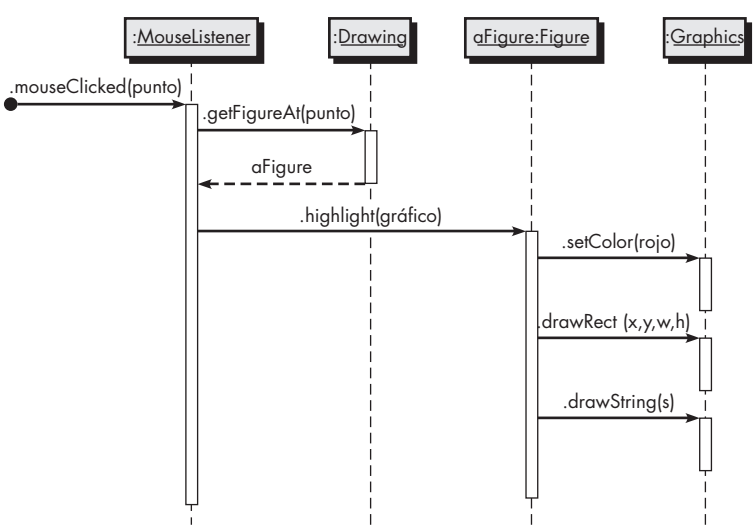
Diagramas de secuencia

En contraste con los diagramas de clase y con los diagramas de implementación, que muestran la estructura estática de un componente de software, un *diagrama de secuencia* se usa para mostrar las comunicaciones dinámicas entre objetos durante la ejecución de una tarea. Este tipo de diagrama muestra el orden temporal en el que los mensajes se envían entre los objetos para lograr dicha tarea. Puede usarse un diagrama de secuencia para mostrar las interacciones en un caso de uso o en un escenario de un sistema de software.

En la figura A1.7 se ve un diagrama de secuencia para un programa de dibujo. El diagrama muestra los pasos involucrados, resaltando una figura en un dibujo cuando se le da clic. Por lo general, cada caja de la fila que hay en la parte superior del diagrama corresponde a un objeto, aunque es posible hacer que las cajas modelen otras cosas, como clases. Si la caja representa un objeto (como es el caso en todos los ejemplos), entonces dentro de la caja puede establecerse de manera opcional el tipo del objeto, precedido por dos puntos. También se puede escribir un nombre del objeto antes de los dos puntos, como se muestra en la tercera caja de la figura A1.7. Abajo de cada caja hay una línea punteada llamada *línea de vida* del objeto. El eje vertical que hay en el diagrama de secuencia corresponde al tiempo, donde el tiempo aumenta conforme se avanza hacia abajo.

Un diagrama de secuencia muestra llamadas de método usando flechas horizontales desde el *llamador* hasta el *llamado*, etiquetado con el nombre del método y que opcionalmente incluye sus parámetros, sus tipos y el tipo de retorno. Por ejemplo, en la figura A1.7, **MouseListener** (escucha de ratón) llama al método *getFigureAt()* (obtener figura en) de **Drawing** (dibujo). Cuando un objeto ejecuta un método (es decir, cuando tiene un marco de activación en la pila), opcionalmente puede mostrar una barra blanca, llamada *barra de activación*, abajo de la línea de vida del objeto. En la figura A1.7, las barras de activación se dibujan para todas las llamadas de método. El diagrama también puede mostrar opcionalmente el retorno de una llamada de método con una flecha punteada y una etiqueta opcional. En la figura A1.7, el retorno de la llamada de método *getFigureAt()* se muestra con una etiqueta del nombre del objeto que regresa.

Figura A1.7
Ejemplo de diagrama de secuencia



Una práctica común, como se hizo en la figura A1.7, es dejar fuera la flecha de retorno cuando se llama un método nulo, pues desordena el diagrama a la vez que proporciona poca información de importancia. Un círculo negro con una flecha que sale de él indica un *mensaje encontrado* cuya fuente se desconoce o es irrelevante.

Ahora debe poder entenderse la tarea que implementa la figura A1.7. Una fuente desconocida llama al método *mouseClicked()* (clic del ratón) de un **MouseListener**, que pasa como argumento el punto donde ocurrió el clic. A su vez, el **MouseListener** llama al método *getFigureAt()* de un **Drawing**, que regresa una **Figure**. Luego el **MouseListener** llama el método resaltado de **Figure** y lo pasa como argumento al objeto **Graphics**. En respuesta, **Figure** llama tres métodos del objeto **Graphics** para dibujar la figura en rojo.

El diagrama en la figura A1.7 es muy directo y no contiene condicionales o bucles. Si se requieren estructuras de control lógico, probablemente sea mejor dibujar un diagrama de secuencia separado para cada caso, es decir, si el flujo del mensaje puede tomar dos rutas diferentes dependiendo de una condición, entonces dibuje dos diagramas de secuencia separados, uno para cada posibilidad.

Si se insiste en incluir bucles, condicionales y otras estructuras de control en un diagrama de secuencia, se pueden usar *marcos de interacción*, que son rectángulos que rodean partes del diagrama y que se etiquetan con el tipo de estructuras de control que representan. La figura A1.8 ilustra lo anterior, y muestra el proceso involucrado, resaltando todas las figuras dentro de un rectángulo determinado. El **MouseListener** envía el mensaje **rectDragged** (arrastre de recta). Entonces el **MouseListener** dice al dibujo que resalte todas las figuras en el rectángulo, llamando al método *highlightFigures()* (resaltar figuras) y pasando al rectángulo como argumento. El método hace bucles a través de todos los objetos **Figure** en el objeto **Drawing** y, si **Figure** intersecta el rectángulo, se pide a **Figure** que se resalte a sí mismo. Las frases entre corchetes se llaman *guardias*, que son condiciones booleanas que deben ser verdaderas si debe continuar la acción dentro del marco de interacción.

Existen muchas otras características especiales que pueden incluirse en un diagrama de secuencia. Por ejemplo:

1. Se puede distinguir entre mensajes sincrónicos y asíncronos. Los primeros se muestran con puntas de flecha sólidas mientras que los asíncronos lo hacen con puntas de flecha huecas.
2. Se puede mostrar un objeto que envía él mismo un mensaje con una flecha que parte del objeto, gira hacia abajo y luego apunta de vuelta hacia el mismo objeto.

FIGURA A1.8
Diagrama de secuencia con dos marcos de interacción

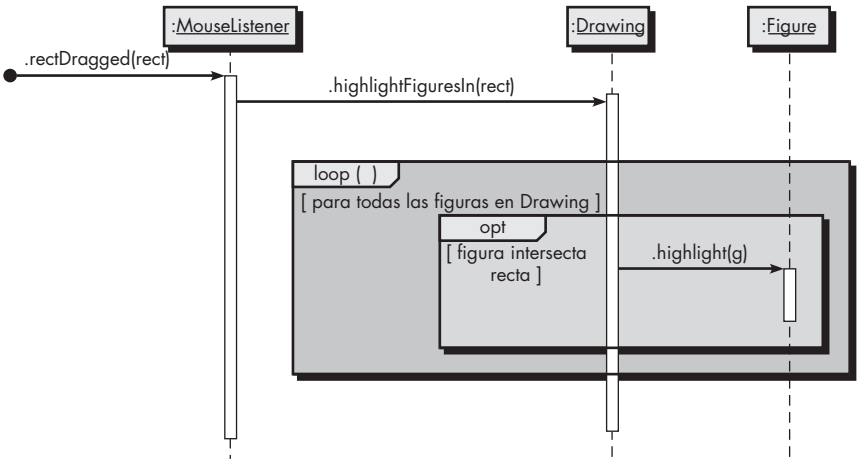
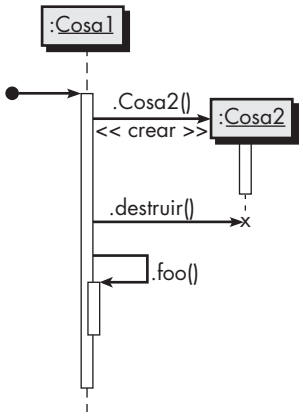


FIGURA A1.9
Creación, destrucción y bucles en diagramas de secuencia



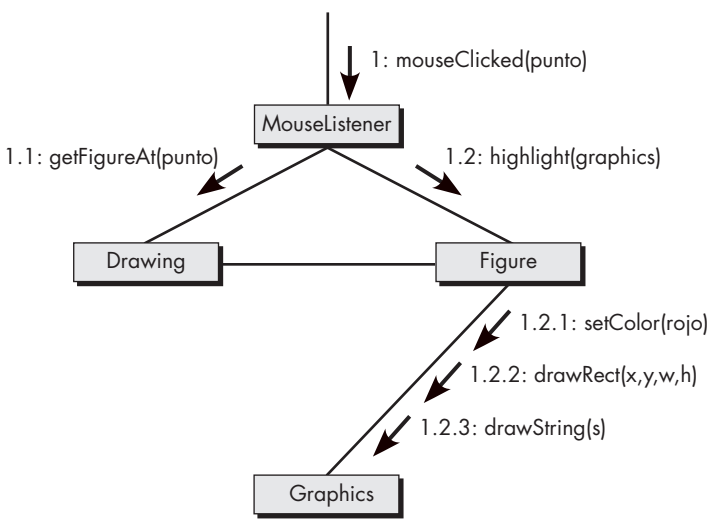
3. Se puede mostrar la creación de objeto dibujando una flecha etiquetada de manera adecuada (por ejemplo, con una etiqueta <<crear>>) hacia una caja de objeto. En este caso, la caja aparecerá en el diagrama más abajo que las cajas correspondientes a objetos que ya existen cuando comienza la acción.
4. Se puede mostrar destrucción de objeto mediante una gran X al final de la línea de vida del objeto. Otros objetos pueden destruir un objeto, en cuyo caso una flecha apunta desde el otro objeto hacia la X. Una X también es útil para indicar que un objeto ya no se usa y que, por tanto, está listo para la colección de basura.

Las últimas tres características se muestran en el diagrama de secuencia de la figura A1.9.

DIAGRAMAS DE COMUNICACIÓN

El *diagrama de comunicación* UML (llamado "diagrama de colaboración" en UML 1.X) proporciona otro indicio del orden temporal de las comunicaciones, pero enfatiza las relaciones entre los objetos y clases en lugar del orden temporal. El diagrama de comunicación que se ilustra en la figura A1.10 despliega las mismas acciones que se muestran en el diagrama de secuencia de la figura A1.7.

FIGURA A1.10
Diagrama de comunicación UML



En un diagrama de comunicación, los objetos interactuantes se representan mediante rectángulos. Las asociaciones entre objetos lo hacen mediante líneas que conectan los rectángulos. Por lo general, en el diagrama existe una flecha entrante hacia un objeto que comienza la secuencia de pase de mensaje. Esa flecha se etiqueta con un número y un nombre de mensaje. Si el mensaje entrante se etiqueta con el número 1 y si hace que el objeto receptor invoque otros mensajes en otros objetos, entonces los mencionados mensajes se representan mediante flechas desde el emisor hacia el receptor a lo largo de una línea de asociación y reciben números 1.1, 1.2, etc., en el orden en el que se llaman. Si tales mensajes a su vez invocan otros mensajes, se agrega otro punto decimal y otro número al número que etiqueta dichos mensajes para indicar un anidado posterior del pase de mensaje.

En la figura A1.10 se ve que el mensaje **mouseClicked** invoca los métodos *getFigureAt()* y luego *highlight()*. El mensaje *highligh t()* invoca otros tres mensajes: *setColor()* (establecer color), *drawRect()* (dibujar recta) y *drawstring()* (dibujar cadena). La numeración en cada etiqueta muestra el anidado y la naturaleza secuencial de cada mensaje.

Existen muchas características opcionales que pueden agregarse a las etiquetas de flecha. Por ejemplo, puede preceder el número con una letra. Una flecha entrante podría etiquetarse **A1: mouseClicked(point)**, lo que indica una hebra de ejecución, A. Si otros mensajes se ejecutan en otras hebras, su etiqueta estaría precedida por una letra diferente. Por ejemplo, si el método *mouseClicked()* se ejecuta en la hebra A, pero crea una nueva hebra B e invoca *highlight()* en dicha hebra, entonces la flecha desde **MouseListener** hacia **Figure** se etiquetaría **1.B2: highlight(graphics)**.

Si el lector está interesado en mostrar las relaciones entre los objetos, además de los mensajes que se envíen entre ellos, probablemente el diagrama de comunicación es una mejor opción que el diagrama de secuencia. Si está más interesado en el orden temporal del paso de mensajes, entonces un diagrama de secuencia probablemente es mejor.

Diagramas de actividad

Un *diagrama de actividad* UML muestra el comportamiento dinámico de un sistema o de parte de un sistema a través del flujo de control entre acciones que realiza el sistema. Es similar a un diagrama de flujo, excepto porque un diagrama de actividad puede mostrar flujos concurrentes.

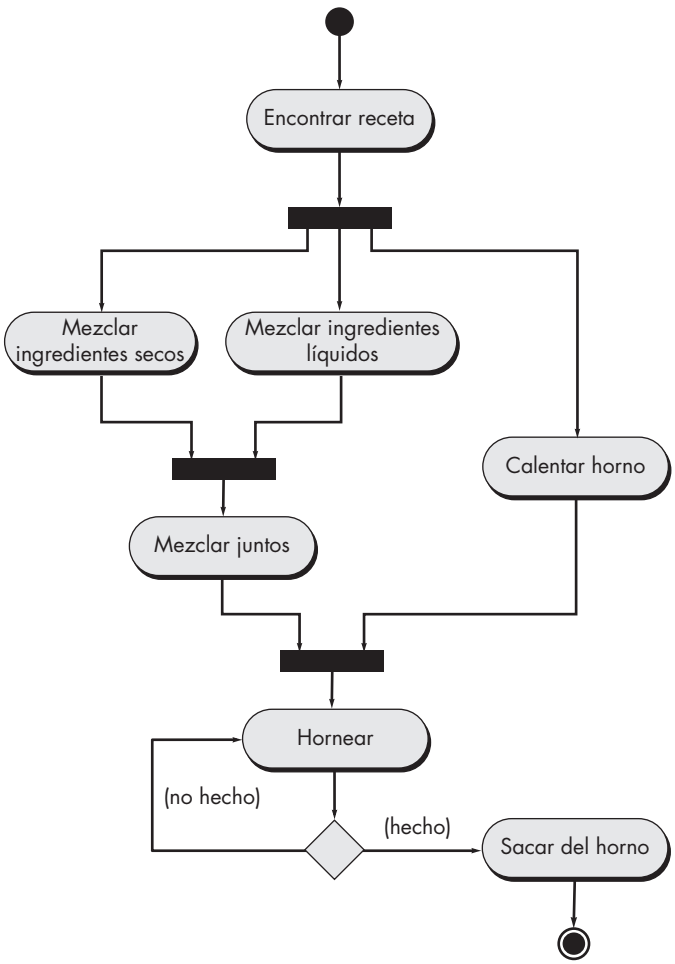
El componente principal de un diagrama de actividad es un nodo *acción*, representado mediante un rectángulo redondeado, que corresponde a una tarea realizada por el sistema de software. Las flechas desde un nodo acción hasta otro indican el flujo de control; es decir, una flecha entre dos nodos acción significa que, después de completar la primera acción, comienza la segunda acción. Un punto negro sólido forma el *nodo inicial* que indica el punto de inicio de la actividad. Un punto negro rodeado por un círculo negro es el *nodo final* que indica el fin de la actividad.

Un *tenedor* (*fork*) representa la separación de actividades en dos o más actividades concurrentes. Se dibuja como una barra negra horizontal con una flecha apuntando hacia ella y dos o más flechas apuntando en sentido opuesto. Cada flecha continua representa un flujo de control que puede ejecutarse de manera concurrente con los flujos correspondientes a las otras flechas continuas. Dichas actividades concurrentes pueden realizarse en una computadora, usando diferentes hebras o incluso diferentes computadoras.

La figura A1.11 muestra un ejemplo de diagrama de actividad que involucra hornear un pastel. El primer paso es encontrar la receta. Una vez encontrada pueden medirse los ingredientes secos y líquidos, mezclarse y precalentar el horno. La mezcla de los ingredientes secos puede hacerse en paralelo con la mezcla de los ingredientes líquidos y el precalentado del horno.

Una *unión* (*join*) es una forma de sincronizar flujos de control concurrentes. Se representa mediante una barra negra horizontal con dos o más flechas entrantes y una flecha saliente. El

FIGURA A1.11
Diagrama de actividad UML que muestra cómo hornear un pastel

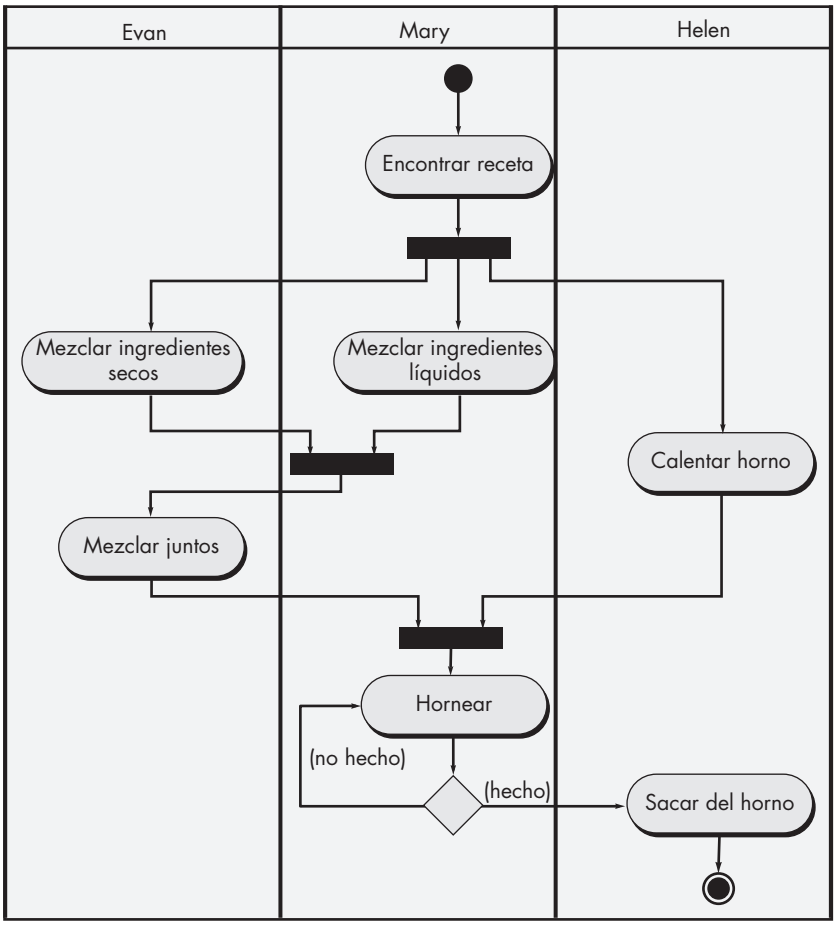


flujo de control representado por la flecha saliente no puede comenzar la ejecución hasta que todos los flujos representados por las flechas entrantes se hayan completado. En la figura A1.11 se tiene una unión antes de la acción de mezclar en conjunto los ingredientes líquidos y secos. Esta unión indica que todos los ingredientes secos deben mezclarse y que debe hacerse lo mismo con todos los ingredientes líquidos antes de poder combinar las dos mezclas. La segunda unión en la figura indica que, antes de comenzar a hornear el pastel, todos los ingredientes deben mezclarse juntos y el horno debe estar a la temperatura correcta.

Un nodo de *decisión* corresponde a una rama en el flujo de control con base en una condición. Tal nodo se despliega como un triángulo blanco con una flecha entrante y dos o más flechas salientes. Cada flecha saliente se etiqueta con una guardia (una condición dentro de corchetes). El flujo de control sigue la flecha saliente cuya guardia es verdadera. Es recomendable asegurarse de que las condiciones cubran todas las posibilidades, de modo que exactamente una de ellas sea verdadera cada vez que se llegue a un nodo de decisión. La figura A1.11 muestra un nodo de decisión que sigue al horneado del pastel. Si el pastel está hecho, entonces se saca del horno. De otro modo, se hornea un poco más.

Una de las cosas que no dice el diagrama de actividad de la figura A1.11 es quién o qué hace cada una de las acciones. Con frecuencia, no importa la división exacta de la mano de obra. Pero si quiere indicar cómo se dividen las acciones entre los participantes, puede decorar el diagrama de actividad con "canales", como se muestra en la figura A1.12. Los *canales*, como el nombre

FIGURA A1.12
Diagrama de actividad de horneado de pastel con "carriles de natación" agregados



implica, se forman dividiendo el diagrama en tiras o "carriles" (como si fuera una alberca con carriles de natación), cada uno de los cuales corresponde a uno de los participantes. Todas las acciones en un carril las realiza el participante correspondiente. En la figura A1.12, Evan es responsable de la mezcla de los ingredientes secos y, luego, de mezclar juntos los ingredientes secos y los líquidos; Helen es responsable de calentar el horno y sacar el pastel; y Mary es responsable de todo lo demás.

DIAGRAMAS DE ESTADO

El comportamiento de un objeto en un punto particular en el tiempo con frecuencia depende del estado del objeto; es decir, de los valores de sus variables en dicho momento. Como ejemplo trivial, considere un objeto con una variable de instancia booleana. Cuando se pide realizar una operación, el objeto puede hacer una cosa si dicha variable es *verdadera* y hacer algo más si es *falsa*.

Un *diagrama de estado* UML modela los estados de un objeto, las acciones que se realizan dependiendo de dichos estados y las transiciones entre los estados del objeto.

Como ejemplo, considere el diagrama de estado para una parte de un compilador Java. La entrada al compilador es un archivo de texto, que puede considerarse como una larga cadena de caracteres. El compilador lee caracteres uno a uno y a partir de ellos determina la estructura del programa. Una pequeña parte de este proceso de lectura de caracteres involucra ignorar

caracteres de "espacio blanco" (por ejemplo, los caracteres *espacio*, *tabulador*, *línea nueva* y *retorno*) y caracteres dentro de un comentario.

Suponga que el compilador delega a un **EliminadorEspacioBlancoyComentario** la labor de avanzar sobre los caracteres de espacio blanco y sobre los caracteres dentro de un comentario, es decir, la labor de dicho objeto es leer los caracteres de entrada hasta que todos los caracteres de espacio blanco y comentario se leyeron, punto donde regresa el control al compilador para leer y procesar caracteres no en blanco y no de comentario. Piense en la manera en la que el objeto **EliminadorEspacioBlancoyComentario** lee los caracteres y determina si el siguiente carácter es de espacio blanco o parte de un comentario. El objeto puede verificar los espacios blancos, probando los siguientes caracteres contra " ", "\t", "\n" y "\r". ¿Pero cómo determina si el siguiente carácter es parte de un comentario? Por ejemplo, cuando ve "/" por primera vez, todavía no sabe si dicho carácter representa un operador división, parte del operador /= o el comienzo de una línea o bloque de comentario. Para hacer esta determinación, **EliminadorEspacioBlancoyComentario** necesita anotar el hecho de que vio una "/" y luego moverse hacia el siguiente carácter. Si el carácter siguiente a "/" es otra "/" o un "*", entonces **EliminadorEspacioBlancoyComentario** sabe que ahora lee un comentario y puede avanzar al final del comentario sin procesar o guardar algún carácter. Si el carácter siguiente al primer "/" es distinto a "/" o a "*", entonces **EliminadorEspacioBlancoyComentario** sabe que "/" representa el operador división o parte del operador /= y, por tanto, avanza a través de los caracteres.

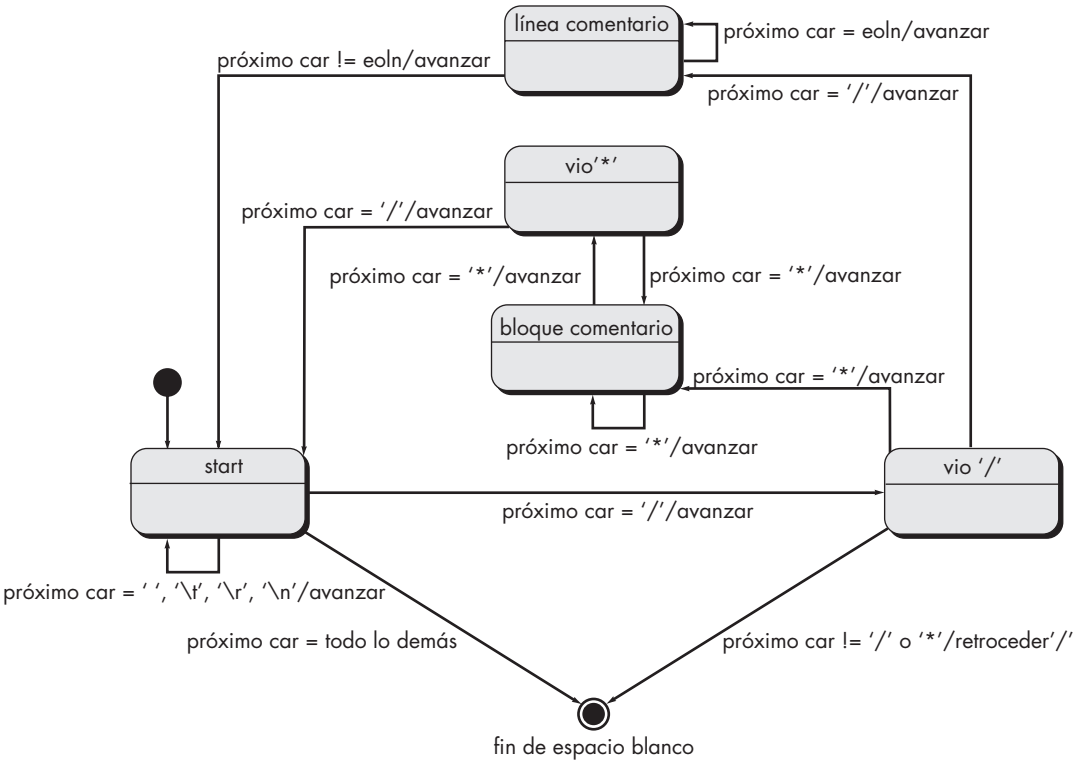
En resumen, conforme **EliminadorEspacioBlancoyComentario** lee los caracteres, necesita seguir la pista de muchas cosas, incluido si el carácter actual es espacio blanco, si el carácter previo que lee fue "/", si actualmente lee caracteres en un comentario, si llegó al final del comentario, etc. Todos éstos corresponden a diferentes estados del objeto **EliminadorEspacioBlancoyComentario**. En cada uno de estos estados, **EliminadorEspacioBlancoyComentario** se comporta de manera diferente con respecto al siguiente carácter que lee.

Para ayudarlo a visualizar todos los estados de este objeto y la manera en la que cambia de estado, puede usar un diagrama de estado UML, como se muestra en la figura A1.13. Un diagrama de estado muestra los estados mediante rectángulos redondeados, cada uno de los cuales tiene un nombre en su mitad superior. También existe un círculo llamado "pseudoestado inicial", que en realidad no es un estado y en vez de ello sólo apunta al estado inicial. En la figura A1.13, el estado **start** es el estado inicial. Las flechas de un estado a otro estado indican transiciones o cambios en el estado del objeto. Cada transición se etiqueta con un evento disparador, una diagonal (/) y una actividad. Todas las partes de las etiquetas de transición son opcionales en los diagramas de estado. Si el objeto está en un estado y el evento disparador ocurre para una de sus transiciones, entonces se realiza dicha actividad de transición y el objeto toma un nuevo estado, indicado por la transición. Por ejemplo, en la figura A1.13, si el objeto **EliminadorEspacioBlancoyComentario** está en el estado **start** y el siguiente carácter es "/", entonces **EliminadorEspacioBlancoyComentario** avanza desde dicho carácter y cambia al estado **vio '/'**. Si el carácter después de "/" es otra "/", entonces el objeto avanza al estado **línea comentario** y permanece ahí hasta que lee un carácter de fin de línea. Si en vez de ello el siguiente carácter después de "/" es "*", entonces el objeto avanza al estado **bloque comentario** y permanece ahí hasta que ve otro "*" seguido por un "/", que indica el final del bloque comentario. Estudie el diagrama para asegurarse de que lo entiende. Observe que, después de avanzar por el espacio en blanco o por un comentario, **EliminadorEspacioBlancoyComentario** regresa al estado **start** y comienza de nuevo. Dicho comportamiento es necesario, pues puede haber varios comentarios sucesivos o caracteres de espacio en blanco antes de cualquier otro carácter en el código fuente Java.

Un objeto puede transitar a un estado final, lo que se indica mediante un círculo negro con un círculo blanco alrededor de él, lo que indica que ya no hay más transiciones. En la figura

FIGURA A1.13

Diagrama de estado para avanzar sobre espacio blanco y comentarios en Java



A1.13, el objeto **EliminadorEspacioBlancoyComentario** termina cuando el siguiente carácter no es espacio en blanco o parte de un comentario. Observe que todas las transiciones, excepto las dos que conducen al estado final, tienen actividades que consisten en avanzar al siguiente carácter. Las dos transiciones hacia el estado final no avanzan sobre el siguiente carácter porque el siguiente carácter es parte de una palabra o símbolo de interés para el compilador. Observe que, si el objeto está en el estado **vio '/'**, pero el siguiente carácter no es "/" o "*", entonces "/" es un operador división o parte del operador /= y, por tanto, no se quiere avanzar. De hecho, se quiere regresar un carácter para hacer el "/" en el siguiente carácter, de modo que "/" pueda usarse por parte del compilador. En la figura A1.13, esta actividad se etiqueta como retroceder '/'.

Un diagrama de estado le ayudará a descubrir situaciones perdidas o inesperadas, es decir, con un diagrama de estado, es relativamente sencillo garantizar que todos los posibles eventos disparadores para todos los estados posibles se representaron. Por ejemplo, en la figura A1.13, puede verificar fácilmente que cada estado incluyó transiciones para todos los posibles caracteres.

Los diagramas de estado UML pueden contener muchas otras características no incluidas en la figura A1.13. Por ejemplo, cuando un objeto está en un estado, por lo general no hace más que sentarse y esperar que ocurra un evento disparador. Sin embargo, existe un tipo especial de estado, llamado *estado de actividad*, donde el objeto realiza alguna actividad, llamada *hacer actividad*, mientras está en dicho estado. Para indicar que un estado es un estado de actividad en el diagrama de estado, se incluye, en la mitad inferior del rectángulo redondeado del estado, la frase "do/" seguida por la actividad que debe realizar mientras está en dicho estado. El "hacer actividad" puede terminar antes de que ocurra cualquier transición de estado, después de lo cual el estado de actividad se comporta como un estado normal de espera. Si una transición del estado de actividad ocurre antes de terminar el "hacer actividad", entonces se interrumpe el "hacer actividad".

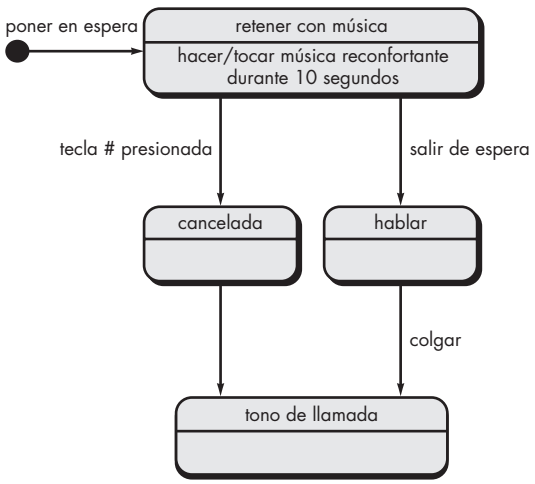


FIGURA A1.14
Diagrama de estado con un estado de actividad y una transición sin disparador

Puesto que un evento disparador es opcional cuando ocurre una transición, es posible que ningún evento disparador pueda mencionarse como parte de una etiqueta de transición. En tales casos, para estados de espera normales, el objeto inmediatamente transitará de dicho estado al nuevo estado. Para estados de actividad, tal transición se realiza tan pronto como termina el "hacer actividad".

La figura A1.14 ilustra esta situación, usando los estados para la operación de un teléfono. Cuando un llamador se coloca en espera, la llamada pasa al estado **retener con música** (la música reconfortante suena durante 10 segundos). Después de 10 segundos, el "hacer actividad" del estado se completa y el estado se comporta como un estado normal de no actividad. Si el llamador presiona la tecla # cuando la llamada está en el estado **retener con música**, la llamada transita hacia el estado **cancelado** e inmediatamente al estado **tono de llamada**. Si la tecla # se presiona antes de completar los 10 segundos de música reconfortante, el "hacer actividad" se interrumpe y la música cesa inmediatamente.

Lenguaje de restricción de objeto (panorama)

La gran variedad de diagramas disponibles como parte de UML le brindan un rico conjunto de formas representativas para el modelo de diseño. Sin embargo, con frecuencia son suficientes las representaciones gráficas. Acaso necesite un mecanismo para representar información de manera explícita y formal que restrinja algún elemento del modelo de diseño. Desde luego, es posible describir las restricciones en lenguaje natural como el inglés, pero este enfoque invariablemente conduce a inconsistencia y ambigüedad. Por esta razón, parece apropiado un lenguaje más formal, extraído de la teoría de conjuntos y de los lenguajes de especificación formales (vea el capítulo 21), pero que tenga la sintaxis un tanto menos matemática de un lenguaje de programación.

El *lenguaje de restricción de objeto* (LRO) complementa al UML, permitiéndole usar una gramática y sintaxis formales para construir enunciados sin ambigüedades acerca de varios elementos del modelo de diseño (por ejemplo, clases y objetos, eventos, mensajes, interfaces). Los enunciados LRO más simples se construyen en cuatro partes: 1) un *contexto* que define la situación limitada en la que es válido el enunciado, 2) una *propiedad* que representa algunas características del contexto (por ejemplo, una propiedad puede ser un atributo si el contexto es una clase), 3) una *operación* (por ejemplo, aritmética, orientada a conjunto) que manipule o califique

una propiedad y 4) palabras clave (por ejemplo, **if, then, else, and, or, not, implies**) que se usan para especificar expresiones condicionales.

Como ejemplo simple de una expresión LRO, considere el sistema de impresión que se estudió en el capítulo 10. La condición guardia se coloca en el evento **jobCostAccepted** que causa una transición entre los estados *computingJobCost* y *formingJob* dentro del diagrama de estado para el objeto **PrintJob** (figura 10.9). En el diagrama (figura 10.9), la condición guardia se expresa en lenguaje natural e implica que la autorización sólo puede ocurrir si el cliente está autorizado para aprobar el costo del trabajo. En LRO, la expresión puede tomar la forma:

```
customer
   self.authorizationAuthority = 'yes'
```

donde un atributo booleano, **authorizationAuthority**, de la clase (en realidad una instancia específica de la clase) llamada **Customer** debe establecerse en "sí" para que se satisfaga la condición guardia.

Conforme se crea el modelo de diseño, con frecuencia existen instancias en las que deben satisfacerse pre o poscondiciones antes de completar alguna acción especificada por el diseño. LRO proporciona una poderosa herramienta para especificar pre y poscondiciones de manera formal. Como ejemplo, considere una extensión al sistema local de impresión (que se estudió como ejemplo en el capítulo 10) donde el cliente proporcione un límite de costo superior (upper cost bound) para el trabajo de impresión y una fecha de entrega "fatal", al mismo tiempo que se especifican otras características del trabajo de impresión. Si las estimaciones de costo y entrega superan dichos límites, el trabajo no se presenta y debe notificársele al cliente. En LRO, un conjunto de pre y poscondiciones puede especificarse de la forma siguiente:

```
context PrintJob::validate(upperCostBound : Integer, custDeliveryReq :
   Integer)
 pre: upperCostBound > 0
   and custDeliveryReq > 0
   and self.jobAuthorization = 'no'
 post: if self.totalJobCost <= upperCostBound
      and self.deliveryDate <= custDeliveryReq
    then
     self.jobAuthorization = 'yes'
    endif
```

Este enunciado LRO define una invariante (**inv**): condiciones que deben existir antes (pre) y después (pos) de algún comportamiento. Inicialmente, una precondición establece que el límite de costo y la fecha de entrega deben especificarse por parte del cliente, y la autorización debe establecerse en "no". Después de determinar costos y entrega, se aplica la poscondición especificada. También debe observarse que la expresión:

```
self.jobAuthorization = 'yes'
```

no asigna el valor "sí", sino que declara que **jobAuthorization** debe establecerse en "sí" para cuando la operación termine. Una descripción completa del LRO está más allá del ámbito de este apéndice. La especificación LRO completa puede obtenerse en **www.omg.org/technology/documents/formal/ocl.htm**

Lecturas y fuentes de información adicionales

Decenas de libros estudian UML. Los que abordan la versión más reciente incluyen: Miles y Hamilton (*Learning UML 2.0,* O'Reilly Media, Inc., 2006); Booch, Rumbaugh, y Jacobson (*Unified Modeling Language User*

Guide, 2a. ed., Addison-Wesley, 2005), Ambler (*The Elements of UML 2.0 Style,* Cambridge University Press, 2005), y Pilone y Pitman (*UML 2.0 in a Nutshell,* O'Reilly Media, Inc., 2005).

En internet está disponible una gran variedad de fuentes de información acerca del UML en el modelado de ingeniería del software. Una lista actualizada de referencias en la World Wide Web puede encontrarse bajo "análisis" y "diseño" en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

APÉNDICE

2

CONCEPTOS ORIENTADOS A OBJETO

CONCEPTOS CLAVE

¿Qué es un punto de vista orientado a objeto (OO)? ¿Por qué un método se considera orientado a objeto? ¿Qué es un objeto? Conforme los conceptos OO ganaron muchos adherentes durante las décadas de 1980 y 1990, había muchas opiniones diferentes acerca de las respuestas correctas a dichas preguntas, pero en la actualidad surgió una visión coherente de los conceptos OO. Este apéndice se diseñó para ofrecer un breve panorama de este importante tema e introducir los conceptos y terminología básicos.

Para entender el punto de vista orientado a objeto, considere un ejemplo de un objeto del mundo real: el objeto sobre el cual se sienta en este momento, una silla. **Silla** es una subclase de una clase mucho más grande que puede llamar **PiezaDeMobiliario**. Las sillas individuales son miembros (por lo general llamadas instancias) de la clase **Silla**. Un conjunto de atributos genéricos pueden asociarse con cada objeto en la clase **PiezaDeMobiliario**. Por ejemplo, todos los muebles tienen un costo, dimensiones, peso, ubicación y color, entre muchos posibles atributos. Lo mismo se aplica si se habla de una mesa, una silla, un sofá o un armario. Puesto que **Silla** es miembro de **PiezaDeMobiliario**, **Silla** hereda todos los atributos definidos para la clase.

Se busca una definición anecdótica de una clase al describir sus atributos, pero algo falta. Todo objeto de la clase **PiezaDeMobiliario** puede manipularse de varias formas. Puede comprarse y venderse, modificarse físicamente (por ejemplo, puede serruchar una pata o pintar el objeto de morado) o moverlo de un lugar a otro. Cada una de estas *operaciones* (otros términos son *servicios* o *métodos*) modificarán uno o más atributos del objeto. Por ejemplo, si el atributo ubicación es un ítem de datos compuesto definido como

ubicación = edificio + piso + habitación

entonces una operación denominada *mover()* modificaría uno o más de los ítems de datos (**edificio, piso** o **habitación**) que forman el atributo **ubicación**. Para hacer esto, *mover()* debe tener "conocimiento" de dichos ítems de datos. La operación *mover()* podría usarse para una silla o una mesa, en tanto ambos sean instancias de la clase **PiezaDeMobiliario**. Las operaciones válidas para la clase **PiezaDeMobiliario** [*comprar()*, *vender()*, *peso()*] son especificadas como parte de la definición de clase y son heredadas por todas las instancias de la clase.

La clase **Silla** (y todos los objetos en general) encapsulan datos (los valores de atributo que definen la silla), operaciones (las acciones que se aplican para cambiar los atributos de silla), otros objetos, constantes (valores de conjunto) y otra información relacionada. *Encapsulación* significa que toda esta información se empaca bajo un nombre y puede reutilizarse como un componente de especificación o programa.

Ahora que se introdujeron algunos conceptos básicos, una definición más formal de *orientado a objeto* resultará más significativa. Coad y Yourdon [Coa91] definen el término de esta forma:

Orientado a objeto = objetos + clasificación + herencia + comunicación

Tres de los conceptos ya se introdujeron. La comunicación se estudia más tarde, en este apéndice.

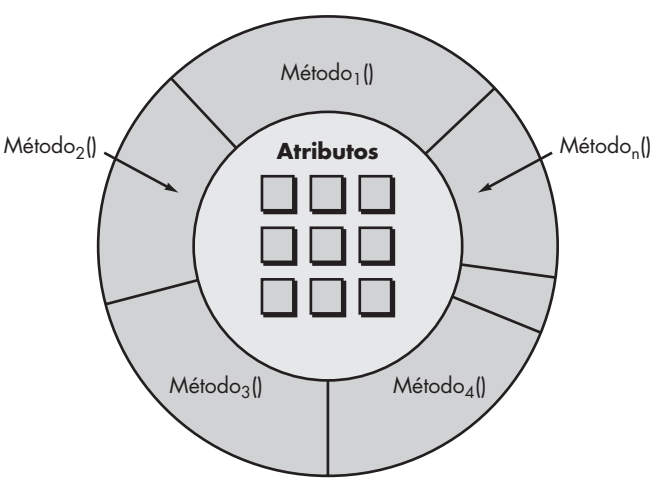
Clases y objetos

Una clase es un concepto OO que encapsula los datos y abstracciones procedurales requeridos para describir el contenido y el comportamiento de alguna entidad del mundo real. Las abstracciones de datos que describen la clase se encierran mediante una "pared" de abstracciones procedurales [Tay90] (representadas en la figura A2.1) que son capaces de manipular los datos de alguna forma. En una clase bien diseñada, la única forma de llegar a los atributos (y operar sobre ellas) es ir a través de uno de los métodos que forman la "pared" que se ilustra en la figura. Por tanto, la clase encapsula datos (dentro de la pared) y el procesamiento que los manipula (los métodos que constituyen la pared). Esto logra que la información se oculte (capítulo 8) y que se reduzca el impacto de los efectos colaterales asociados con el cambio. Dado que los métodos tienden a manipular un número limitado de atributos, su cohesión mejora, y puesto que la comunicación ocurre solamente a través de los métodos que constituyen la "pared", la clase tiende a estar menos fuertemente acoplada a otros elementos de un sistema.[1]

Dicho de otra forma, una clase es una descripción generalizada (por ejemplo, una plantilla o plano) de una colección de objetos similares. Por definición, los objetos son instancias de una clase específica y heredan sus atributos y las operaciones que están disponibles para manipular esos atributos. Una *superclase* (con frecuencia llamada *clase base*) es una generalización de un conjunto de clases que se relacionan con ella. Una *subclase* es una especialización de la superclase. Por ejemplo, la superclase **VehículoDeMotor** es una generalización de las clases **Camión**, **SUV**, **Automóvil** y **Van**. La subclase **Automóvil** hereda todos los atributos de **VehículoDeMotor**, pero además incorpora atributos adicionales que son específicos solamente de automóviles.

Estas definiciones implican la existencia de una jerarquía de clase en la que los atributos y operaciones de la superclase se heredan por parte de la subclase, que puede agregar atributos y métodos "privados" adicionales. Por ejemplo, las operaciones *sentarEn()* y *voltear()* pueden ser privativas de la subclase **Silla**.

Figura A2.1
Representación esquemática de una clase



1 Sin embargo, debe observar que el acoplamiento puede convertirse en un serio problema en los sistemas OO, cuando las clases de varias partes del sistema se usan como los tipos de atributos de datos y los argumentos como métodos. Aun cuando el acceso a los objetos sólo puede ser a través de llamados a procedimiento, esto no significa que el acoplamiento es necesariamente bajo, sólo más bajo que si se permitiera el acceso directo al interior de los objetos.

Atributos

Usted aprendió que los atributos se vinculan a las clases y que describen la clase en alguna forma. Un atributo puede tomar un valor definido por un *dominio* enumerado. En la mayoría de los casos, un dominio es simplemente un conjunto de valores específicos. Por ejemplo, suponga que una clase **Automóvil** tiene un atributo **color**. El dominio de valores para **color** es {**blanco, negro, plata, gris, azul, rojo, amarillo, verde**}. En situaciones más complejas, el dominio puede ser una clase. Continuando con el ejemplo, la clase **Automóvil** también tiene un atributo **trenPotencia** que en sí mismo es una clase. La clase **TrenPotencia** contendría atributos que describen el motor y la transmisión específicos del vehículo.

Las *características* (valores del dominio) pueden aumentar al asignar un valor por defecto (característica) a un atributo. Por ejemplo, el atributo **color** por defecto es **blanco**. También puede ser útil asociar una probabilidad con una característica particular al asignar pares {valor, probabilidad}. Considere el atributo **color** para automóvil. En algunas aplicaciones (por ejemplo, planificar la fabricación) puede ser necesario asignar una probabilidad a cada uno de los colores (por ejemplo, blanco y negro son enormemente probables como colores de automóvil).

Operaciones, métodos y servicios

Un objeto encapsula datos (representados como una colección de atributos) y los algoritmos que los procesan. Dichos algoritmos se llaman *operaciones*, *métodos* o *servicios*[2] y pueden verse como componentes de procesamiento.

Cada una de las operaciones que se encapsula mediante un objeto proporciona una representación de uno de los comportamientos del objeto. Por ejemplo, la operación *ObtenerColor()* para el objeto **Automóvil** extraerá el color almacenado en el atributo **color**. La implicación de la existencia de estas operaciones es que la clase **Automóvil** se diseñó para recibir un estímulo (a los estímulos se les llama *mensajes*) que solicitan el color de la instancia particular de una clase. Siempre que un objeto recibe un estímulo, inicia cierto comportamiento. Esto puede ser tan simple como recuperar el color del automóvil o tan complejo como el inicio de una cadena de estímulos que pasan entre varios objetos diferentes. En el último caso, considere un ejemplo donde el estímulo inicial recibido por el **Objeto 1** da como resultado la generación de otros dos estímulos que se envían al **Objeto 2** y al **Objeto 3**. Las operaciones encapsuladas por el segundo y tercer objetos actúan sobre los estímulos, y regresan información necesaria al primer objeto. Entonces el **Objeto 1** usa la información devuelta para satisfacer el comportamiento demandado por el estímulo inicial.

Conceptos de análisis y diseño orientado a objeto

El modelado de requerimientos (también llamado modelado de análisis) se enfoca principalmente en clases que se extraen directamente del enunciado del problema. Dichas *clases de entidad* por lo general representan cosas que deben almacenarse en una base de datos y que persisten a lo largo de la duración de la aplicación (a menos que se borren de manera específica).

El diseño refina y extiende el conjunto de clases de entidad. Las clases frontera y controlador se desarrollan y/o refinan durante el diseño. Las *clases frontera* crean la interfaz (por ejemplo, pantalla interactiva y reportes impresos) que el usuario ve y con la cual interactúa conforme se

2 En el contexto de esta discusión se usa el término *operaciones*, pero los términos *métodos* y *servicios* son igualmente populares.

usa el software. Las clases frontera se diseñan con la responsabilidad de gestionar la forma en la que los objetos entidad se presentan a los usuarios.

Las *clases controlador* se diseñan para gestionar: 1) la creación o actualización de objetos entidad, 2) la instanciación de objetos frontera conforme obtienen información de objetos entidad, 3) comunicación compleja entre conjuntos de objetos y 4) validación de datos comunicados entre objetos o entre el usuario y la aplicación.

Los conceptos analizados en los párrafos que siguen pueden ser útiles en trabajos de análisis y diseño.

Herencia. La herencia es uno de los diferenciadores clave entre sistemas convencionales y orientados a objeto. Una subclase **Y** hereda todos los atributos y operaciones asociadas con su superclase **X**. Esto significa que todas las estructuras de datos y algoritmos originalmente diseñados e implementados para **X** están disponibles de inmediato para **Y**; no se necesita hacer más trabajo. La reutilización se logra directamente.

Cualquier cambio a los atributos u operaciones contenidos dentro de una superclase se hereda inmediatamente para todas las subclases. Por tanto, la jerarquía de clase se convierte en un mecanismo mediante el cual los cambios (en niveles superiores) pueden propagarse inmediatamente a través de un sistema.

Es importante observar que en cada nivel de la jerarquía de clase pueden agregarse nuevos atributos y operaciones a las que se heredaron de niveles superiores en la jerarquía. De hecho, siempre que se crea una nueva clase, tendrá algunas opciones:

- La clase puede diseñarse y construirse desde cero, es decir, no se usa herencia.
- La jerarquía de clase puede revisarse para determinar si una clase superior en la jerarquía contiene más de los atributos y operaciones requeridos. La nueva clase hereda de la clase superior y entonces pueden agregarse adiciones, según se requiera.
- La jerarquía de clase puede reestructurarse de modo que los atributos y operaciones requeridos pueden heredarse en la nueva clase.
- Las características de una clase existente pueden ser excesivas, y diferentes versiones de atributos u operaciones se implementan para la nueva clase.

Como todo concepto de diseño fundamental, la herencia puede proporcionar beneficios significativos para el diseño, pero si se usa de manera inadecuada,[3] puede complicar un diseño de manera innecesaria y conducir a software proclive a errores, que es difícil de mantener.

Mensajes. Las clases deben interactuar unas con otras para lograr las metas del diseño. Un mensaje estimula la ocurrencia de algunos comportamientos en el objeto receptor. El comportamiento se logra cuando una operación se ejecuta.

La interacción entre los objetos se ilustra de manera esquemática en la figura A2.2. Una operación dentro de **ObjetoEmisor** genera un mensaje de la forma *mensaje* (<parámetros>) donde los parámetros identifican **ObjetoReceptor** con el objeto que se va a estimular mediante el mensaje; la operación dentro de **ObjetoReceptor** consiste en recibir el mensaje y los ítems de datos que proporcionan información requerida para que la operación sea exitosa. La colaboración definida entre clases como parte del modelo de requerimientos proporciona lineamientos útiles en el diseño de mensajes.

Cox [Cox86] describe el intercambio entre clases en la forma siguiente:

> Un objeto [clase] se solicita para realizar una de sus operaciones al enviarle un mensaje que diga qué hacer al objeto. El receptor [objeto] responde al mensaje al elegir primero la operación que imple-

3 Por ejemplo, diseñar una subclase que herede atributos y operaciones de más de una superclase (en ocasiones llamada "herencia múltiple") es mal vista por la mayoría de los diseñadores.

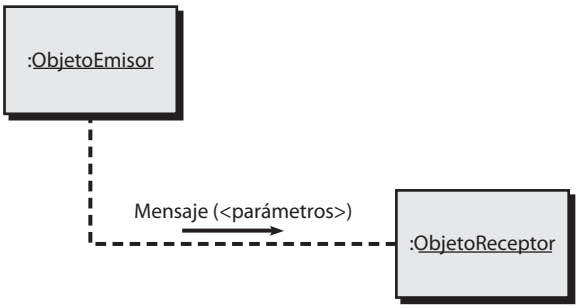


FIGURA A2.2
Mensaje que pasa entre objetos

menta el nombre del mensaje, ejecutar esta operación y luego regresar el control al solicitante. Los mensajes ligan el sistema orientado a objeto. Los mensajes proporcionan comprensión acerca del comportamiento de objetos individuales y del sistema OO como un todo.

Polimorfismo. El *polimorfismo* es una característica que reduce enormemente el esfuerzo requerido para extender el diseño de un sistema orientado a objeto existente. Para entender el polimorfismo, considere una aplicación convencional que debe dibujar cuatro tipos diferentes de gráficas: gráficas de línea, gráficas de pastel, histogramas y diagramas Kiviat. De manera ideal, una vez recopilados los datos para un tipo particular de gráfica, ésta debe dibujarse a sí misma. Para lograrlo en una aplicación convencional (y mantener cohesión de módulo) sería necesario desarrollar módulos de dibujo para cada tipo de gráfica. Entonces, dentro del diseño, debería incrustarse lógica de control similar a la siguiente:

```
caso de gráficatipo:
   if gráficatipo = graficalinea then DibujarGráficaLinea (datos);
   if gráficatipo = graficapie then DibujarGráficaPie (datos);
   if gráficatipo = histograma then DibujarHisto (datos);
   if gráficatipo = kiviat then DibujarKiviat (datos);
termina caso;
```

Aunque este diseño es razonablemente directo, agregar nuevos tipos de gráfica podría ser truculento. Tendría que crearse un nuevo módulo de dibujo para cada tipo de gráfica y luego la lógica de control tendría que actualizarse para reflejar el nuevo tipo de gráfica.

Para resolver este problema en un sistema orientado a objeto, todas las gráficas se convierten en subclases de una clase general llamada **Gráfica**. Usando un concepto llamado *sobrecargar* [Tay90], cada subclase define una operación llamada *dibujar*. Un objeto puede enviar un mensaje *dibujar* a cualquiera de los objetos instanciados de cualquiera de las subclases. El objeto que recibe el mensaje invocará su propia operación *dibujo* para crear la gráfica adecuada. Por tanto, el diseño se reduce a

```
dibujar <gráficatipo>
```

Cuando un nuevo tipo de gráfica se agrega al sistema, se crea una subclase con su propia operación *dibujar*. Pero no se requieren cambios dentro de algún objeto que quiera una gráfica dibujada porque el mensaje **dibujar <gráficatipo>** sigue invariable. Para resumir, el polimorfismo permite que algunas operaciones diferentes tengan el mismo nombre. Esto a su vez desacopla objetos uno de otro, lo que los hace más independientes.

Clase de diseño. El modelo de requerimientos define un conjunto completo de clases de análisis. Cada una describe algún elemento del dominio de problema, y se enfoca en aspectos del

problema que son visibles al usuario o cliente. El nivel de abstracción de cualquier clase de análisis es relativamente alto.

Conforme evoluciona el modelo de diseño, el equipo de software debe definir un conjunto de *clase de diseño* que 1) refinen las clases de análisis, proporcionando detalle de diseño que permitirá la implementación de las clases y 2) crear un nuevo conjunto de clases de diseño que implementen una infraestructura de software que dé apoyo a la solución empresarial. Se sugieren cinco tipos diferentes de clases de diseño, cada una como representación de una capa diferente de la arquitectura de diseño [Amb01]:

- *Clases de interfaz de usuario:* definen todas las abstracciones que se necesitan para la interacción humano-computadora (IHC).
- *Clases de dominio empresarial:* con frecuencia son refinamientos de las clases de análisis definidas anteriormente. Las clases identifican los atributos y operaciones (métodos) que se requieren para implementar algún elemento del dominio empresarial.
- *Clases de proceso:* implementan abstracciones empresariales de nivel inferior requeridas para gestionar por completo las clases de dominio empresarial.
- *Clases persistentes:* representan almacenes de datos (por ejemplo, una base de datos) que persistirá más allá de la ejecución del software.
- *Clases de sistema:* implementan gestión de software y funciones de control que permiten al sistema operar y comunicarse dentro de su entorno de computación y con el mundo exterior.

Conforme evoluciona el diseño arquitectónico, el equipo de software debe desarrollar un conjunto completo de atributos y operaciones para cada clase de diseño. El nivel de abstracción se reduce conforme cada clase de análisis se transforma en una representación de diseño. Es decir, las clases de análisis representan objetos (y métodos asociados que se les aplican), usando la jerga del dominio empresarial. Las clases de diseño presentan detalle significativamente más técnico como una guía para la implementación.

Arlow y Neustadt [Arl02] sugieren que cada clase de diseño se revise para garantizar que está "bien formada". Definen cuatro características de una clase de diseño bien formada:

Completa y suficiente. Una clase de diseño debe ser el encapsulamiento completo de todos los atributos y métodos que razonablemente pueda esperarse que existan para la clase (con base en una interpretación enterada del nombre de la clase). Por ejemplo, la clase **Escena** definida para software de edición de video es completa sólo si contiene todos los atributos y métodos que razonablemente puedan asociarse con la creación de una escena de video. La suficiencia garantiza que la clase de diseño contiene solamente aquellos métodos que son suficientes para lograr la intención de la clase, ni más y ni menos.

Primitividad. Los métodos asociados con una clase de diseño deben enfocarse en lograr una función específica para la clase. Una vez implementada la función con un método, la clase no debe proporcionar otra vía para lograr lo mismo. Por ejemplo, la clase **VideoClip** del software de edición puede tener atributos **punto-inicial** y **punto-final** para indicar los puntos de inicio y finalización del clip (observe que el video en bruto cargado en el sistema puede ser más largo que el clip que se usa). Los métodos, *setStartPoint()* y *setEndPoint()* proporcionan el único medio para establecer puntos de inicio y finalización para el clip.

Alta cohesión. Una clase de diseño cohesiva tiene un solo propósito: tiene un pequeño conjunto enfocado en responsabilidades y aplica atributos y métodos decisivos para implementar dichas responsabilidades. Por ejemplo, la clase **VideoClip** del software de edición de video puede contener un conjunto de métodos para editar el clip de video. En tanto cada

método se enfoque exclusivamente en atributos asociados con el videoclip, la cohesión se mantiene.

Low coupling. Dentro del modelo de diseño es necesario que las clases de diseño colaboren unas con otras. No obstante, la colaboración debe mantenerse en un mínimo aceptable. Si un modelo de diseño está enormemente acoplado (todas las clases de diseño colaboran con todas las otras clases de diseño), el sistema es difícil de implementar, probar y mantener con el tiempo. En general, las clases de diseño dentro de un subsistema deben tener solamente conocimiento limitado de otras clases. Esta restricción, llamada *ley de Démeter* [Lie03], sugiere que un método sólo debe enviar mensajes a métodos en clases vecinas.[4]

Lecturas y fuentes de información adicionales

Durantes las tres décadas pasadas se escribieron cientos de libros acerca de programación, análisis y diseño orientado a objeto. Weisfeld (*The Object-Oriented Thought Process*, 2a. ed., Sams Publishing, 2003) presenta un valioso tratamiento de los conceptos y principios generales OO. McLaughlin *et al.* (*Head First Object-Oriented Analysis and Design: A Brain Friendly Guide to OOA&D*, O'Reilly Media, Inc., 2006) proporcionan un tratamiento accesible y disfrutable del análisis OO y de los enfoques al diseño. Un tratamiento más profundo del análisis y diseño OO se presenta en Booch *et al.* (*Objetct-Oriented Analysis and Design with Applications*, 3a. ed., Addison-Wesley, 2007). Wu (*An Introduction to Object-Oriented Programming with Java*, McGraw-Hill, 2005) escribió un libro exhaustivo acerca de programación OO que es clásico de decenas de escritos para muchos diferentes lenguajes de programación.

En internet está disponible una gran variedad de fuentes de información acerca de tecnologías orientadas a objeto. Una lista actualizada de referencias en la World Wide Web puede encontrarse bajo "análisis" y "diseño" en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

4 Una manera menos formal de plantear la ley de Démeter es, "Cada unidad sólo debe hablar con sus amigos; no hablar con extraños".

Referencias

[Abb83] Abbott, R., "Program Design by Informal English Descriptions", *CACM*, vol. 26, núm. 11, noviembre 1983, pp. 892-894.

[ACM98] ACM/IEEE-CS Joint Task Force, *Software Engineering Code of Ethics and Professional Practice*, 1998, disponible en www.acm.org/serving/se/code.htm.

[Ada93] Adams, D., *Mostly Harmless*, Macmillan, 1993.

[AFC88] *Software Risk Abatement*, AFCS/AFLC Pamphlet 800-845, U.S. Air Force, septiembre 30, 1988.

[Agi03] The Agile Alliance Home Page, www.agilealliance.org/home.

[Air99] Airlie Council, "Performance Based Management: The Program Manager's Guide Based on the 16-Point Plan and Related Metrics", Draft Report, marzo 8, 1999.

[Aka04] Akao, Y., *Quality Function Deployment*, Productivity Press, 2004.

[Ale77] Alexander, C., *A Pattern Language*, Oxford University Press, 1977.

[Ale79] Alexander, C., *The Timeless Way of Building*, Oxford University Press, 1979.

[Amb95] Ambler, S., "Using Use-Cases", *Software Development*, julio 1995, pp. 53-61.

[Amb98] Ambler, S., *Process Patterns: Building Large-Scale Systems Using Object Technology*, Cambridge University Press/SIGS Books, 1998.

[Amb01] Ambler, S., *The Object Primer*, 2a. ed., Cambridge University Press, 2001.

[Amb02a] Ambler, S., "What Is Agile Modeling (AM)?" 2002, www.agilemodeling.com/index.htm.

[Amb02b] Ambler, S. y R. Jeffries, *Agile Modeling*, Wiley, 2002.

[Amb02c] Ambler, S., "UML Component Diagramming Guidelines", disponible en www. modelingstyle.info/, 2002.

[Amb04] Ambler, S., "*Examining the Cost of Change Curve*", en *The Object Primer*, 3a. ed., Cambridge University Press, 2004.

[Amb06] Ambler, S., "The Agile Unified Process (AUP)", 2006, disponible en www.ambysoft.com/unifiedprocess/agileUP.html.

[And06] Andrews, M. y J. Whittaker, *How to Break Web Software: Functional and Security Testing of Web Applications and Web Services*, Addison-Wesley, 2006.

[ANS87] ANSI/ASQC A3-1987, *Quality Systems Terminology*, 1987.

[Ant06] Anton, D. y C. Anton, *ISO 9001 Survival Guide*, 3a. ed., AEM Consulting Group, 2006.

[AOS07] AOSD.net (Aspect-Oriented Software Development), glosario, disponible en http://aosd.net/wiki/index.php?title=Glossary.

[App00] Appleton, B., "Patterns and Software: Essential Concepts and Terminology", febrero 2000, disponible en www.cmcrossroads.com/bradapp/docs/patterns-intro.html.

[App08] Apple Computer, *Accessibility*, 2008, disponible en www.apple.com/disability/.

[Arl02] Arlow, J. y I. Neustadt, *UML and the Unified Process*, Addison-Wesley, 2002.

[Arn89] Arnold, R. S., "Software Restructuring", *Proc. IEEE*, vol. 77, núm. 4, abril 1989, pp. 607-617.

[Art97] Arthur, L. J., "Quantum Improvements in Software System Quality", *CACM*, vol. 40, núm. 6, junio 1997, pp. 47-52.

[Ast04] Astels, D., *Test Driven Development: A Practical Guide*, Prentice Hall, 2004.

[Ave04] Aversan, L., *et al*., "Managing Coordination and Cooperation in Distributed Software Processes: The GENESIS Environment", *Software Process Improvement and Practice*, vol. 9, Wiley Interscience, 2004, pp. 239-263.

[Baa07] de Baar, B., "Project Risk Checklist", 2007, disponible en www.softwareprojects.org/ project_riskmanagement_starting62.htm.

[Bab86] Babich, W. A., *Software Configuration Management*, Addison-Wesley, 1986.

[Bac97] Bach, J., "Good Enough Quality: Beyond the Buzzword", *IEEE Computer*, vol. 30, núm. 8, agosto 1997, pp. 96-98.

[Bac98] Bach, J., "The Highs and Lows of Change Control", *Computer*, vol. 31, núm. 8, agosto 1998, pp. 113-115.

[Bae98] Baetjer, Jr., H., *Software as Capital*, IEEE Computer Society Press, 1998, p. 85.

[Bak72] Baker, F. T., "Chief Programmer Team Management of Production Programming", *IBM Systems Journal.*, vol. 11, núm. 1, 1972, pp. 56-73.

[Ban06] Baniassad, E., *et al*., "Discovering Early Aspects", *IEEE Software*, vol. 23, núm. 1, enero-febrero, 2006, pp. 61-69.

[Bar06] Baresi, L., E. DiNitto y C. Ghezzi, "Toward Open-World Software: Issues and Challenges", *IEEE Computer*, vol. 39, núm. 10, octubre 2006, pp. 36-43.

[Bas84] Basili, V. R. y D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data", *IEEE Trans. Software Engineering*, vol. SE-10, 1984, pp. 728-738.

[Bas03] Bass, L., P. Clements y R. Kazman, *Software Architecture in Practice*, 2a. ed., Addison-Wesley, 2003.
[Bec00] Beck, K., *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.
[Bec01a] Beck, K., *et al.*, "Manifesto for Agile Software Development", www.agilemanifesto.org/.
[Bec04a] Beck, K., *Extreme Programming Explained: Embrace Change*, 2a. ed., Addison-Wesley, 2004.
[Bec04b] Beck, K., *Test-Driven Development: By Example*, 2a. ed., Addison-Wesley, 2002.
[Bee99] Beedle, M., *et al.*, "SCRUM: An Extension Pattern Language for Hyperproductive Software Development", incluido en: *Pattern Languages of Program Design 4*, Addison-Wesley Longman, Reading MA, 1999, descargable de http://jeffsutherland.com/scrum/ scrum_plop.pdf.
[Bei84] Beizer, B., *Software System Testing and Quality Assurance*, Van Nostrand-Reinhold, 1984.
[Bei90] Beizer, B., *Software Testing Techniques*, 2a. ed., Van Nostrand-Reinhold, 1990.
[Bei95] Beizer, B., *Black-Box Testing*, Wiley, 1995.
[Bel81] Belady, L., prólogo de *Software Design: Methods and Techniques* (L. J. Peters, autor), Yourdon Press, 1981.
[Bel95] Bellinzona R., M. G. Gugini y B. Pernici, "Reusing Specifications in OO Applications", *IEEE Software*, marzo 1995, pp. 65-75.
[Ben99] Bentley, J., *Programming Pearls*, 2a. ed., Addison-Wesley, 1999.
[Ben00] Bennatan, E. M., *Software Project Management: A Practitioner's Approach*, 3a. ed., McGraw-Hill, 2000.
[Ben02] Bennett, S., S. McRobb y R. Farmer, *Object-Oriented Analysis and Design*, 2a. ed., McGraw-Hill, 2002.
[Ber80] Bersoff, E. H., V. D. Henderson y S. G. Siegel, *Software Configuration Management*, Prentice Hall, 1980.
[Ber93] Berard, E., *Essays on Object-Oriented Software Engineering*, vol. 1, Addison-Wesley, 1993.
[Bes04] Bessin, J., "The Business Value of Quality", IBM developerWorks, junio 15, 2004, disponible en www-128.ibm.com/developerworks/rational/library/4995.html.
[Bha06] Bhat, J., M. Gupta y S. Murthy, "Lessons from Offshore Outsourcing", *IEEE Software*, vol. 23, núm. 5, septiembre-octubre 2006.
[Bie94] Bieman, J. M. y L. M. Ott, "Measuring Functional Cohesion", *IEEE Trans. Software Engineering*, vol. SE-20, núm. 8, agosto 1994, pp. 308-320.
[Bin93] Binder, R., "Design for Reuse Is for Real", *American Programmer*, vol. 6, núm. 8, agosto 1993, pp. 30-37.
[Bin94a] Binder, R., "Testing Object-Oriented Systems: A Status Report", *American Programmer*, vol. 7, núm. 4, abril 1994, pp. 23-28.
[Bin94b] Binder, R. V., "Object-Oriented Software Testing", *Communications of the ACM*, vol. 37, núm. 9, septiembre 1994, p. 29.
[Bin99] Binder, R., *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, 1999.
[Bir98] Biró, M. y T. Remzsö, "Business Motivations for Software Process Improvement", ERCIM News, núm. 32, enero 1998, disponible en www.ercim.org/publication/Ercim_News/enw32/biro.html.
[Boe81] Boehm, B., *Software Engineering Economics*, Prentice Hall, 1981.
[Boe88] Boehm, B., "A Spiral Model for Software Development and Enhancement", *Computer*, vol. 21, núm. 5, mayo 1988, pp. 61-72.
[Boe89] Boehm, B. W., *Software Risk Management*, IEEE Computer Society Press, 1989.
[Boe96] Boehm, B., "Anchoring the Software Process", *IEEE Software*, vol. 13, núm. 4, julio 1996, pp. 73-82.
[Boe98] Boehm, B., "Using the WINWIN Spiral Model: A Case Study", *Computer*, vol. 31, núm. 7, julio 1998, pp. 33-44.
[Boe00] Boehm, B., *et al.*, *Software Cost Estimation in COCOMO II*, Prentice Hall, 2000.
[Boe01a] Boehm, B., "The Spiral Model as a Tool for Evolutionary Software Acquisition", *CrossTalk*, mayo 2001, disponible en www.stsc.hill.af.mil/crosstalk/2001/05/boehm.html.
[Boe01b] Boehm, B. y V. Basili, "Software Defect Reduction Top 10 List", *IEEE Computer*, vol. 34, núm. 1, enero 2001, pp. 135-137.
[Boe08] Boehm, B., "Making a Difference in the Software Century", *IEEE Computer*, vol. 41, núm. 3, marzo 2008, pp. 32-38.
[Boh66] Bohm, C. y G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules", *CACM*, vol. 9, núm. 5, mayo 1966, pp. 366-371.
[Boh00] Bohl, M. y M. Rynn, *Tools for Structured Design: An Introduction to Programming Logic*, 5a. ed., Prentice Hall, 2000.
[Boi04] Boiko, B., *Content Management Bible*, 2a. ed., Wiley, 2004.
[Bol02] Boldyreff, C., *et al.*, "Environments to Support Collaborative Software Engineering", 2002, descargable de www.cs.put.poznan.pl/dweiss/site/publications/download/csmre-paper.pdf.
[Boo94] Booch, G., *Object-Oriented Analysis and Design*, 2a. ed., Benjamin Cummings, 1994.
[Boo05] Booch, G., J. Rumbaugh y I. Jacobsen, *The Unified Modeling Language User Guide*. 2a. ed., Addison-Wesley, 2005.
[Boo06] Bootstrap-institute.com, 2006, www.cse.dcu.ie/espinode/directory/directory.html.
[Boo08] Booch, G., *Handbook of Software Architecture*, 2008, disponible en www.booch.com/architecture/systems.jsp.

[Bor01] Borchers, J., *A Pattern Approach to Interaction Design*, Wiley, 2001.
[Bos00] Bosch, J., *Design & Use of Software Architectures*, Addison-Wesley, 2000.
[Bra85] Bradley, J. H., "The Science and Art of Debugging", *Computerworld*, agosto 19, 1985, pp. 35-38.
[Bra94] Bradac, M., D. Perry y L. Votta, "Prototyping a Process Monitoring Experiment", *IEEE Trans. Software Engineering*, vol. 20, núm. 10, octubre 1994, pp. 774-784.
[Bre02] Breen, P., "Exposing the Fallacy of 'Good Enough' Software", informit.com, febrero 1, 2002, disponible en www.informit.com/articles/article.asp?p=25141&rl=1.
[Bro95] Brooks, F., *The Mythical Man-Month*, Silver Anniversary edition, Addison-Wesley, 1995.
[Bro96] Brown, A. W. y K. C. Wallnau, "Engineering of Component Based Systems", *Component-Based Software Engineering*, IEEE Computer Society Press, 1996, pp. 7-15.
[Bro01] Brown, B., *Oracle9i Web Development*, 2a. ed., McGraw-Hill, 2001.
[Bro03] Brooks, F, "Three Great Challenges for Half-Century-Old Computer Science", *JACM*, vol. 50, núm. 1, enero 2003, pp. 25-26.
[Bro06] Broy, M., "The 'Grand Challenge' in Informatics: Engineering Software Intensive Systems", *IEEE Computer*, vol. 39, núm. 10, octubre 2006, pp. 72-80.
[Buc99] Bucanac, C., "The V-Model", University of Karlskrona/Ronneby, enero 1999, descargable de www.bucanac.com/documents/The_V-Model.pdf.
[Bud96] Budd, T., *An Introduction to Object-Oriented Programming*, 2a. ed., Addison-Wesley, 1996.
[Bus96] Buschmann, F., *et al.*, *Pattern-Oriented Software Architecture*, Wiley, 1996.
[Bus07] Buschmann, F., *et al.*, *Pattern-Oriented Software Architecture, A System of Patterns*, Wiley, 2007.
[Cac02] Cachero, C., *et al.*, "Conceptual Navigation Analysis: A Device and Platform Independent Navigation Specification", *Proc. 2nd Intl. Workshop on Web-Oriented Technology*, junio 2002, descargable de www.dsic.upv.es/~west/iwwost02/papers/cachero.pdf.
[Cai03] Caine, Frarber y Gordon, Inc., *PDL/81*, 2003, disponible en www.cfg.com/pdl81/lpd.html.
[Car90] Card, D. N. y R. L. Glass, *Measuring Software Design Quality*, Prentice Hall, 1990.
[Cas89] Cashman, M., "Object Oriented Domain Analysis", *ACM Software Engineering Notes*, vol. 14, núm. 6, octubre 1989, p. 67.
[Cav78] Cavano, J. P. y J. A. McCall, "A Framework for the Measurement of Software Quality", *Proc. ACM Software Quality Assurance Workshop*, noviembre 1978, pp. 133-139.
[CCS02] CS3 Consulting Services, 2002, www.cs3inc.com/DSDM.htm.
[Cec06] Cechich, A., *et al.*, "Trends on COTS Component Identification", *Proc. Fifth Intl. Conf. on COTS-Based Software Systems*, IEEE, 2006.
[Cha89] Charette, R. N., *Software Engineering Risk Analysis and Management*, McGraw-Hill/ Intertext, 1989.
[Cha92] Charette, R. N., "Building Bridges over Intelligent Rivers", *American Programmer*, vol. 5, núm. 7, septiembre 1992, pp. 2-9.
[Cha93] de Champeaux, D., D. Lea y P. Faure, *Object-Oriented System Development*, Addison-Wesley, 1993.
[Cha03] Chakravarti, A., "Online Software Design Pattern Links", 2003, disponible en www.anupriyo.com/oopfm.shtml.
[Che77] Chen, P., *The Entity-Relationship Approach to Logical Database Design*, QED Information Systems, 1977.
[Chi94] Chidamber, S. R. y C. F. Kemerer, "A Metrics Suite for Object-Oriented Design", *IEEE Trans. Software Engineering*, vol. SE-20, núm. 6, junio 1994, pp. 476-493.
[Cho89] Choi, S. C. y W. Scacchi, "Assuring the Correctness of a Configured Software Description", *Proc. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, octubre 1989, pp. 66-75.
[Chu95] Churcher, N. I. y M. J. Shepperd, "Towards a Conceptual Framework for Object-Oriented Metrics", *ACM Software Engineering Notes*, vol. 20, núm. 2, abril 1995, pp. 69-76.
[Cig07] Cigital, Inc., "Case Study: Finding Defects Earlier Yields Enormous Savings", 2007, disponible en www.cigital.com/solutions/roi-cs2.php.
[Cla05] Clark, S. y E. Baniasaad, *Aspect-Oriented Analysis and Design*, Addison-Wesley, 2005.
[Cle95] Clements, P., "From Subroutines to Subsystems: Component Based Software Development", *American Programmer*, vol. 8, núm. 11, noviembre 1995.
[Cle03] Clements, P., R. Kazman y M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2003.
[Cle06] Clemmons, R., "Project Estimation with Use Case Points", *CrossTalk*, febrero 2006, p. 18-222, descargable de www.stsc.hill.af.mil/crosstalk/2006/02/0602Clemmons.pdf.
[CMM07] *Capability Maturity Model Integration (CMMI)*, Software Engineering Institute, 2007, disponible en www.sei.cmu.edu/cmmi/.
[CMM08] *People Capability Maturity Model Integration (People CMM)*, Software Engineering Institute, 2008, disponible en www.sei.cmu.edu/cmm-p/.
[Coa91] Coad, P. y E. Yourdon, *Object-Oriented Analysis*, 2a. ed., Prentice Hall, 1991.
[Coa99] Coad, P., E. Lefebvre y J. DeLuca, *Java Modeling in Color with UML*, Prentice Hall, 1999.
[Coc01a] Cockburn, A. y J. Highsmith, "Agile Software Development: The People Factor", *IEEE Computer*, vol. 34, núm. 11, noviembre 2001, pp. 131-133.
[Coc01b] Cockburn, A., *Writing Effective Use-Cases*, Addison-Wesley, 2001.
[Coc02] Cockburn, A., *Agile Software Development*, Addison-Wesley, 2002.

[Coc04] Cockburn, A., "What the Agile Toolbox Contains", *CrossTalk*, noviembre 2004, disponible en www.stsc.hill.af.mil/crosstalk/2004/11/0411Cockburn.html.
[Coc05] Cockburn, A., *Crystal Clear*, Addison-Wesley, 2005.
[Con96] Conradi, R., "Software Process Improvement: Why We Need SPIQ", NTNU, octubre 1996, descargable de www.idi.ntnu.no/grupper/su/publ/pdf/nik96-spiq.pdf.
[Con02] Conradi, R. y A. Fuggetta, "Improving Software Process Improvement", *IEEE Software*, julio-agosto 2002, pp. 2-9, descargable de http://citeseer.ist.psu.edu/conradi02improving.html.
[Con93] Constantine, L., "Work Organization: Paradigms for Project Management and Organization", *CACM*, vol. 36, núm. 10, octubre 1993, pp. 34-43.
[Con95] Constantine, L, "What DO Users Want? Engineering Usability in Software", *Windows Tech Journal*, diciembre 1995, disponible en www.forUse.com.
[Con03] Constantine, L. y L. Lockwood, *Software for Use*, Addison-Wesley, 1999; vea también www.foruse.com/.
[Cop05] Coplien, J., "Software Patterns", 2005, disponible en http://hillside.net/patterns/definition.html.
[Cor98] Corfman, R., "An Overview of Patterns", en *The Patterns Handbook*, SIGS Books, 1998.
[Cou00] Coulouris, G., J. Dollimore y T. Kindberg, *Distributed Systems: Concepts and Design*, 3a. ed., Addison-Wesley, 2000.
[Cox86] Cox, Brad, *Object-Oriented Programming*, Addison-Wesley, 1986.
[Cri92] Christel, M. G. y K. C. Kang, "Issues in Requirements Elicitation", Software Engineering Institute, CMU/SEI-92-TR-12 7, septiembre 1992.
[Cro79] Crosby, P., *Quality Is Free*, McGraw-Hill, 1979.
[Cro07] Cross, M. y M. Fisher, *Developer's Guide to Web Application Security*, Syngress Publishing, 2007.
[Cur86] Curritt, P. A., M. Dyer y H. D. Mills, "Certifying the Reliability of Software", *IEEE Trans, Software Engineering*, vol. SE-12, núm. 1, enero 1994.
[Cur88] Curtis, B., *et al.*, "A Field Study of the Software Design Process for Large Systems", *IEEE Trans. Software Engineering*, vol. SE-31, núm. 11, noviembre 1988, pp. 1268-1287.
[Cur01] Curtis, B., W. Hefley y S. Miller, *People Capability Maturity Model*, Addison-Wesley, 2001.
[CVS07] Concurrent Versions System, Ximbiot, http://ximbiot.com/cvs/wiki/index.php?title= Main_Page, 2007.
[DAC03} "An Overview of Model-Based Testing for Software", Data and Analysis Center for Software, CR/TA 12, junio 2003, descargable de www.goldpractices.com/dwnload/ practice/pdf/Model_Based_Testing.pdf.
[Dah72] Dahl, O., E. Dijkstra y C. Hoare, *Structured Programming*, Academic Press, 1972.
[Dar91] Dart, S., "Concepts in Configuration Management Systems", *Proc. Third International Workshop on Software Configuration Management*, ACM SIGSOFT, 1991, descargable de www.sei.cmu.edu/legacy/scm/abstracts/abscm_concepts.html.
[Dar99] Dart, S., "Change Management: Containing the Web Crisis", *Proc. Software Configuration Management Symposium*, Toulouse, Francia, 1999, disponible en www.perforce.com/perforce/ conf99/dart.html.
[Dar01] Dart, S., *Spectrum of Functionality in Configuration Management Systems*, Software Engineering Institute, 2001, disponible en www.sei.cmu.edu/legacy/scm/tech_rep/TR11_90/ TOC_TR11_90.html.
[Das05] Dasari, R., "Lean Software Development", a white paper, descargable de www .projectperfect.com.au/downloads/Info/info_lean_development.pdf, 2005.
[Dav90] Davenport, T. H. y J. E. Young, "The New Industrial Engineering: Information Technology and Business Process Redesign", *Sloan Management Review*, verano 1990, pp. 11-27.
[Dav93] Davis, A., *et al.*, "Identifying and Measuring Quality in a Software Requirements Specification", *Proc. First Intl. Software Metrics Symposium*, IEEE, Baltimore, MD, mayo 1993, pp. 141-152.
[Dav95a] Davis, M., "Process and Product: Dichotomy or Duality", *Software Engineering Notes*, ACM Press, vol. 20, núm. 2, abril, 1995, pp. 17-18.
[Dav95b] Davis, A., *201 Principles of Software Development*, McGraw-Hill, 1995.
[Day99] Dayani-Fard, H., *et al.*, "Legacy Software Systems: Issues, Progress, and Challenges", IBM Technical Report: TR-74.165-k, abril 1999, disponible en www.cas.ibm.com/toronto/ publications/TR-74.165/k/legacy.html.
[Dem86] Deming, W. E., *Out of the Crisis*, MIT Press, 1986.
[DeM79] DeMarco, T., *Structured Analysis and System Specification*, Prentice Hall, 1979.
[DeM95] DeMarco, T., *Why Does Software Cost So Much?* Dorset House, 1995.
[DeM95a] DeMarco, T., "Lean and Mean", *IEEE Software*, noviembre 1995, pp. 101-102.
[DeM98] DeMarco, T. y T. Lister, *Peopleware*, 2a. ed., Dorset House, 1998.
[DeM02] DeMarco, T. y B. Boehm, "The Agile Methods Fray", IEEE Computer, vol. 35, núm. 6, junio 2002, pp. 90-92.
[Den73] Dennis, J., "Modularity", en *Advanced Course on Software Engineering* (F. L. Bauer, ed.), Springer-Verlag, 1973, pp. 128-182.
[Dev01] Devedzik, V., "Software Patterns", en *Handbook of Software Engineering and Knowledge Engineering*, World Scientific Publishing Co., 2001.
[Dha95] Dhama, H., "Quantitative Metrics for Cohesion and Coupling in Software", *Journal of Systems and Software*, vol. 29, núm. 4, abril 1995.

[Dij65] Dijkstra, E., "Programming Considered as a Human Activity", en *Proc. 1965 IFIP Congress*, North-Holland Publishing Co., 1965.
[Dij72] Dijkstra, E., "The Humble Programmer", 1972 ACM Turing Award Lecture, *CACM*, vol. 15, núm. 10, octubre 1972, pp. 859-866.
[Dij76a] Dijkstra, E., "Structured Programming", en *Software Engineering, Concepts and Techniques*, (J. Buxton *et al.*, eds.), Van Nostrand-Reinhold, 1976.
[Dij76b] Dijkstra, E., *A Discipline of Programming*, Prentice Hall, 1976.
[Dij82] Dijksta, E., "On the Role of Scientific Thought", *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982.
[Dix99] Dix, A., "Design of User Interfaces for the Web", *Proc. User Interfaces to Data Systems Conference*, septiembre 1999, descargable de www.comp.lancs.ac.uk/computing/ users/dixa/topics/webarch/.
[Dob04] Dobb, F., *ISO 9001:2000 Quality Registration Step-by-Step*, 3a. ed., Butterworth- Heinemann, 2004.
[Don99] Donahue, G., S. Weinschenck y J. Nowicki, "Usability Is Good Business", Compuware Corp., julio 1999, disponible en www.compuware.com.
[Dre99] Dreilinger, S., "CVS Version Control for Web Site Projects", 1999, disponible en www.durak.org/cvswebsites/howto-cvs/howto-cvs.html.
[Dru75] Drucker, P., *Management*, W. H. Heinemann, 1975.
[Duc01] Ducatel, K., *et al.*, *Scenarios for Ambient Intelligence in 2010*, ISTAG-European Commission, 2001, descargable de ftp://ftp.cordis.europa.eu/pub/ist/docs/istagscenarios2010.pdf.
[Dun82] Dunn, R. y R. Ullman, *Quality Assurance for Computer Software*, McGraw-Hill, 1982.
[Dun01] Dunaway, D. y S. Masters, *CMM-Based Appraisal for Internal Process Improvement (CBA IPI Version 1,2 Method Description)*, Software Engineering Institute, 2001, descargable de www.sei.cmu.edu/publications/documents/01.reports/01tr033.html.
[Dun02] Dunn, W., *Practical Design of Safety-Critical Computer Systems*, William Dunn, 2002.
[Duy02] VanDuyne, D., J. Landay y J. Hong, *The Design of Sites*, Addison-Wesley, 2002.
[Dye92] Dyer, M., *The Cleanroom Approach to Quality Software Development*, Wiley, 1992.
[Edg95] Edgemon, J., "Right Stuff: How to Recognize It When Selecting a Project Manager", *Application Development Trends*, vol. 2, núm. 5, mayo 1995, pp. 37-42.
[Eji91] Ejiogu, L., *Software Engineering with Formal Metrics*, QED Publishing, 1991.
[Elr01] Elrad, T., R. Filman y A. Bader (eds.), "Aspect Oriented Programming", *Comm. ACM*, vol. 44, núm. 10, octubre 2001, número especial.
[Eri05] Ericson, C., *Hazard Analysis Techniques for System Safety*, Wiley-Interscience, 2005.
[Eri08] Erickson, T., *The Interaction Design Patterns Page*, mayo 2008, disponible en www.visi .com/~snowfall/InteractionPatterns.html.
[Eva04] Evans, E., *Domain Driven Design*, Addison-Wesley, 2003.
[Fag86] Fagan, M., "Advances in Software Inspections", *IEEE Trans. Software Engineering*, vol. 12, núm. 6, julio 1986.
[Fel89] Felican, L. y G. Zalateu, "Validating Halstead's Theory for Pascal Programs", *IEEE Trans. Software Engineering*, vol. SE-15, núm. 2, diciembre 1989, pp. 1630-1632.
[Fel07] Feller, J., *et al.* (eds.), *Perspectives on Free and Open Source Software*, The MIT Press, 2007.
[Fen91] Fenton, N., *Software Metrics*, Chapman y Hall, 1991.
[Fen94] Fenton, N., "Software Measurement: A Necessary Scientific Basis", *IEEE Trans. Software Engineering*, vol. SE-20, núm. 3, marzo 1994, pp. 199-206.
[Fer97] Ferguson, P., *et al.*, "Results of Applying the Personal Software Process", *IEEE Computer*, vol. 30, núm. 5, mayo 1997, pp. 24-31.
[Fer98] Ferdinandi, P. L., "Facilitating Communication", *IEEE Software*, septiembre 1998, pp. 92-96.
[Fer00] Fernandez, E. B. y X. Yuan, "Semantic Analysis Patterns", *Proceedings of the 19th Int. Conf. on Conceptual Modeling, ER2000*, Lecture Notes in Computer Science 1920, Springer, 2000, pp. 183-195. También disponible en www.cse.fau.edu/~ed/SAPpaper2.pdf.
[Fir93] Firesmith, D. G., *Object-Oriented Requirements Analysis and Logical Design*, Wiley, 1993.
[Fis06] Fisher, R. y D. Shapiro, *Beyond Reason: Using Emotions as You Negotiate*, Penguin, 2006.
[Fit54] Fitts, P., "The Information Capacity of the Human Motor System in Controlling the Amplitude of Movement", *Journal of Experimental Psychology*, vol. 47, 1954, pp. 381-391.
[Fle98] Fleming, Q. W. y J. M. Koppelman, "Earned Value Project Management", *CrossTalk*, vol. 11, núm. 7, julio 1998, p. 19.
[Fos06] Foster, E., "Quality Culprits", InfoWorld Grip Line Weblog, mayo 2, 2006, disponible en http://weblog.infoworld.com/gripeline/2006/05/02_a395.html.
[Fow97] Fowler, M., *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.
[Fow00] Fowler, M., *et al.*, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.
[Fow01] Fowler, M. y J. Highsmith, "The Agile Manifesto", *Software Development Magazine*, agosto 2001, www.sdmagazine.com/documents/s=844/sdm0108a/0108a.htm.
[Fow02] Fowler. M., "The New Methodology", junio 2002, www.martinfowler.com/articles/newMethodology.html#N8B.
[Fow03] Fowler, M., *et al.*, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.

[Fow04] Fowler, M., *UML Distilled*, 3a. ed., Addison-Wesley, 2004.
[Fra93] Frankl, P. G. y S. Weiss, "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow", *IEEE Trans. Software Engineering*, vol. SE-19, núm. 8, agosto 1993, pp. 770-787.
[Fra03] Francois, A., "Software Architecture for Immersipresence", IMSC Technical Report IMSC-03-001, University of Southern California, diciembre 2003, disponible en http://iris.usc.edu/ ~afrancoi/pdf/sai-tr.pdf.
[Fre80] Freeman, P., "The Context of Design", en *Software Design Techniques*, 3a. ed. (P. Freeman y A. Wasserman, eds.), IEEE Computer Society Press, 1980, pp. 2-4.
[Fre90] Freedman, D. P. y G. M. Weinberg, *Handbook of Walkthroughs, Inspections and Technical Reviews*, 3a. ed., Dorset House, 1990.
[Gag04] Gage, D. y J. McCormick, "We Did Nothing Wrong", *Baseline Magazine*, marzo 4, 2004, disponible en www.baselinemag.com/article2/0,1397,1544403,00.asp.
[Gai95] Gaines, B., "Modeling and Forecasting the Information Sciences", Technical Report, University of Calgary, Calgary, Alberta, septiembre 1995.
[Gam95] Gamma, E., *et al.*, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
[Gar84] Garvin, D., "What Does 'Product Quality' Really Mean?" *Sloan Management Review*, otoño 1984, pp. 25-45.
[Gar87] Garvin D., *"Competing on the Eight Dimensions of Quality", Harvard Business Review*, noviembre 1987, pp. 101-109. Resumen disponible en www.acm.org/crossroads/ xrds6-4/software.html.
[Gar95] Garlan, D. y M. Shaw, "An Introduction to Software Architecture", *Advances in Software Engineering and Knowledge Engineering*, vol. I (V. Ambriola and G. Tortora, eds.), World Scientific Publishing Company, 1995.
[Gar08] GartnerGroup, "Understanding Hype Cycles", 2008, disponible en www.gartner.com/ pages/story.php.id.8795.s.8.jsp.
[Gau89] Gause, D. C. y G. M. Weinberg, *Exploring Requirements: Quality Before Design*, Dorset House, 1989.
[Gey01] Geyer-Schulz, A. y M. Hahsler, "Software Engineering with Analysis Patterns", Technical Report 01/2001, Institut für Informationsverarbeitung und -wirtschaft, Wirschaftsuniversität Wien, noviembre 2001, descargable de wwwai.wu-wien.ac.at/~hahsler/ research/virlib_working2001/virlib/.
[Gil88] Gilb, T., *Principles of Software Project Management*, Addison-Wesley, 1988.
[Gil95] Gilb, T., "What We Fail to Do in Our Current Testing Culture", *Testing Techniques Newsletter* (online edition, ttn@soft.com), Software Research, enero 1995.
[Gil06] Gillis, D., "Pattern-Based Design", tehan + lax blog, septiembre 14, 2006, disponible en www.teehan-lax.com/blog/?p=96.
[Gla98] Glass, R., "Defining Quality Intuitively", *IEEE Software*, mayo 1998, pp. 103-104, 107.
[Gla00] Gladwell, M., *The Tipping Point*, Back Bay Books, 2002.
[Gli07] Glinz, M. y R. Wieringa, "Stakeholders in Requirements Engineering", *IEEE Software*, vol. 24, núm. 2, marzo-abril 2007, pp. 18-20.
[Glu94] Gluch, D., "A Construct for Describing Software Development Risks", CMU/SEI-94-TR-14, Software Engineering Institute, 1994.
[Gna99] Gnaho, C. y F. Larcher, "A User-Centered Methodology for Complex and Customizable Web Engineering", *Proc. 1st ICSE Workshop on Web Engineering*, ACM, Los Angeles, mayo 1999.
[Gon04] Gonzales, R., "Requirements Engineering", Sandia National Laboratories, presentación de diapositivas, disponible en www.incose.org/enchantment/docs/04AprRequirementsEngineering.pdf.
[Gor02] Gordon, B. y M. Gordon, *The Complete Guide to Digital Graphic Design*, Watson-Guptill, 2002.
[Gor06] Gorton, I., *Essential Software Architecture*, Springer, 2006.
[Gra87] Grady, R. B. y D. L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice Hall, 1987.
[Gra92] Grady, R. B., *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall, 1992.
[Gra99] Grable, R., *et al.*, "Metrics for Small Projects: Experiences at SED," *IEEE Software*, marzo 1999, pp. 21-29.
[Gra03] Gradecki, J. y N. Lesiecki, *Mastering AspectJ: Aspect-Oriented Programming in Java*, Wiley, 2003.
[Gru02] Grundy, J., "Aspect-Oriented Component Engineering", 2002, www.cs.auckland.ac.nz/~john-g/aspects.html.
[Gus89] Gustavsson, A., "Maintaining the Evolution of Software Objects in an Integrated Environment", *Proc. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, octubre 1989, pp. 114-117.
[Gut93] Guttag, J. V. y J. J. Horning, *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.
[Hac98] Hackos, J. y J. Redish, *User and Task Analysis for Interface Design*, Wiley, 1998.
[Hai02] Hailpern, B. y P. Santhanam, "Software Debugging, Testing and Verification", *IBM Systems Journal*, vol. 41, núm. 1, 2002, disponible en www.research.ibm.com/journal/sj/ 411/hailpern.html.
[Hal77] Halstead, M., *Elements of Software Science*, North-Holland, 1977.

[Hal90] Hall, A., "Seven Myths of Formal Methods", *IEEE Software*, septiembre 1990, pp. 11-20.
[Hal98] Hall, E. M., *Managing Risk: Methods for Software Systems Development*, Addison-Wesley, 1998.
[Ham90] Hammer, M., "Reengineer Work: Don't Automate, Obliterate", *Harvard Business Review*, julio-agosto 1990, pp. 104-112.
[Han95] Hanna, M., "Farewell to Waterfalls", *Software Magazine*, mayo 1995, pp. 38-46.
[Har98a] Harmon, P., "Navigating the Distributed Components Landscape", *Cutter IT Journal.*, vol. 11, núm. 2, diciembre 1998, pp. 4-11.
[Har98b] Harrison, R., S. J. Counsell y R. V. Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics", *IEEE Trans. Software Engineering*, vol. SE-24, núm. 6, junio 1998, pp. 491-496.
[Her00] Herrmann, D., *Software Safety and Reliability*, Wiley-IEEE Computer Society Press, 2000.
[Het84] Hetzel, W., *The Complete Guide to Software Testing*, QED Information Sciences, 1984.
[Het93] Hetzel, W., *Making Software Measurement Work*, QED Publishing, 1993.
[Hev93] Hevner, A. R. y H. D. Mills, "Box Structure Methods for System Development with Objects", *IBM Systems Journal*, vol. 31, núm. 2, febrero 1993, pp. 232-251.
[Hig95] Higuera, R. P., "Team Risk Management", *CrossTalk*, U.S. Dept. of Defense, enero 1995, pp. 2-4.
[Hig00] Highsmith, J., *Adaptive Software Development: An Evolutionary Approach to Managing Complex Systems*, Dorset House Publishing, 2000.
[Hig01] Highsmith, J. (ed.), "The Great Methodologies Debate: Part 1", *Cutter IT Journal.*, vol. 14, núm. 12, diciembre 2001.
[Hig02a] Highsmith, J. (ed.), "The Great Methodologies Debate: Part 2", *Cutter IT Journal.*, vol. 15, núm. 1, enero 2002.
[Hig02b] Highsmith, J., *Agile Software Development Ecosystems*, Addison-Wesley, 2002.
[Hil05] Hildreth, S., "Buggy Software: Up from a Low Quality Quagmire", *Computerworld*, julio 25, 2005, disponible en www.computerworld.com/developmenttopics/development/story/ 0,10801,103378,00.html.
[Hil08] Hillside.net, *Patterns Catalog*, 2008, disponible en http://hillside.net/patterns/ onlinepatterncatalog.htm.
[Hob06] Hoberman, S., *Data Modeling Made Simple*, Technics Publications, 2006.
[Hof00] Hofmeister, C., R. Nord y D. Soni, *Applied Software Architecture*, Addison-Wesley, 2000.
[Hof01] Hofmann, C., *et al.*, "Approaches to Software Architecture", 2001, descargable de http://citeseer.nj.nec.com/84015.html.
[Hol06] Holzner, S., *Design Patterns for Dummies*, For Dummies Publishers, 2006.
[Hoo96] Hooker, D., "Seven Principles of Software Development", septiembre 1996, disponible en http://c2.com/cgi/wikiSevenPrinciplesOfSoftwareDevelopment.
[Hop90] Hopper, M. D., "Rattling SABRE, New Ways to Compete on Information", *Harvard Business Review*, mayo-junio 1990.
[Hor03] Horch, J., *Practical Guide to Software Quality Management*, 2a. ed., Artech House, 2003.
[HPR02] Hypermedia Design Patterns Repository, 2002, disponible en www.designpattern .lu.unisi.ch/index.htm.
[Hum95] Humphrey, W., *A Discipline for Software Engineering*, Addison-Wesley, 1995.
[Hum96] Humphrey, W., "Using a Defined and Measured Personal Software Process", *IEEE Software*, vol. 13, núm. 3, mayo-junio 1996, pp. 77-88.
[Hum97] Humphrey, W., *Introduction to the Personal Software Process*, Addison-Wesley, 1997.
[Hum98] Humphrey, W., "The Three Dimensions of Process Improvement, Part III: The Team Process", *CrossTalk*, abril 1998, disponible en www.stsc.hill.af.mil/crosstalk/1998/apr/dimensions.asp.
[Hum00] Humphrey, W., *Introduction to the Team Software Process*, Addison-Wesley, 2000.
[Hun99] Hunt, A., D. Thomas y W. Cunningham, *The Pragmatic Programmer*, Addison-Wesley, 1999.
[Hur83] Hurley, R. B., *Decision Tables in Software Engineering*, Van Nostrand-Reinhold, 1983.
[Hya96] Hyatt, L. y L. Rosenberg, "A Software Quality Model and Metrics for Identifying Project Risks and Assessing Software Quality", NASA SATC, 1996, disponible en http://satc.gsfc.nasa.gov/support/STC_APR96/quality/stc_qual.html.
[IBM81] "Implementing Software Inspections", notas del curso, IBM Systems Sciences Institute, IBM Corporation, 1981.
[IBM03] IBM, *Web Services Globalization Model*, 2003, disponible en www.ibm.com/ developerworks/webservices/library/ws-global/.
[IEE93a] *IEEE Standards Collection: Software Engineering*, IEEE Standard 610.12-1990, IEEE, 1993.
[IEE93b] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, 1993.
[IEE00] IEEE Standard Association, IEEE-Std-1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems*, 2000, disponible en http://standards.ieee.org/ reading/ieee/std_public/description/se/1471-2000_desc.html.
[IFP01] *Function Point Counting Practices Manual*, Release 4.1.1, International Function Point Users Group, 2001, disponible en www.ifpug.org/publications/manual.htm.
[IFP05] Function Point Bibliography/Reference Library, International Function Point Users Group, 2005, disponible en www.ifpug.org/about/bibliography.htm.
[ISI08] iSixSigma, LLC, "New to Six Sigma: A Guide for Both Novice and Experiences Quality Practitioners", 2008, disponible en www.isixsigma.com/library/content/six-sigma-newbie.asp.

[ISO00] ISO 9001: 2000 Document Set, International Organization for Standards, 2000, www.iso.ch/iso/en/iso9000-14000/iso9000/iso9000index.html.
[ISO02] *Z Formal Specification Notation—Syntax, Type System and Semantics*, ISO/IEC 13568:2002, Intl. Standards Organization, 2002.
[ISO08] ISO SPICE, 2008, www.isospice.com/categories/SPICE-Project/.
[Ivo01] Ivory, M., R. Sinha y M. Hearst, "Empirically Validated Web Page Design Metrics", ACM *SIGCHI'01*, marzo 31-abril 4, 2001, disponible en http://webtango.berkeley.edu/papers/chi2001/.
[Jac75] Jackson, M. A., *Principles of Program Design*, Academic Press, 1975.
[Jac92] Jacobson, I., *Object-Oriented Software Engineering*, Addison-Wesley, 1992.
[Jac98] Jackman, M., "Homeopathic Remedies for Team Toxicity", *IEEE Software*, julio 1998, pp. 43-45.
[Jac99] Jacobson, I., G. Booch y J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.
[Jac02a] Jacobson, I., "A Resounding 'Yes' to Agile Processes—But Also More", *Cutter IT Journal*, vol. 15, núm. 1, enero 2002, pp. 18-24.
[Jac02b] Jacyntho, D., D. Schwabe y G. Rossi, "An Architecture for Structuring Complex Web Applications", 2002, disponible en www2002.org/CDROM/alternate/478/.
[Jac04] Jacobson, I. y P. Ng, *Aspect-Oriented Software Development*, Addison-Wesley, 2004.
[Jal04] Jalote, P., *et al.*, "Timeboxing: A Process Model for Iterative Software Development", *Journal of Systems and Software,*vol. 70, núm. 2, 2004, pp. 117-127. Disponible en www.cse .iitk.ac.in/users/jalote/papers/Timeboxing.pdf.
[Jay94] Jaychandra, Y., *Re-engineering the Networked Enterprise*, McGraw-Hill, 1994.
[Jec06] Jech, T., *Set Theory*, 3a. ed., Springer, 2006.
[Jon86] Jones, C., *Programming Productivity*, McGraw-Hill, 1986.
[Jon91] Jones, C., *Systematic Software Development Using VDM*, 2a. ed., Prentice Hall, 1991.
[Jon96] Jones, C., "How Software Estimation Tools Work", *American Programmer*, vol. 9, núm. 7, julio 1996, pp. 19-27.
[Jon98] Jones, C., *Estimating Software Costs*, McGraw-Hill, 1998.
[Jon04] Jones, C., "Software Project Management Practices: Failure Versus Success", *CrossTalk*, octubre 2004. Disponible en www.stsc.hill.af.mil/crossTalk/2004/10/0410Jones.html.
[Joy00] Joy, B., "The Future Doesn't Need Us", *Wired*, vol. 8, núm. 4, abril 2000.
[Kai02] Kaiser, J., "Elements of Effective Web Design", About, Inc., 2002, disponible en http://webdesign.about.com/library/weekly/aa091998.htm.
[Kal03] Kalman, S., *Web Security Field Guide*, Cisco Press, 2003.
[Kan93] Kaner, C., J. Falk y H. Q. Nguyen, *Testing Computer Software*, 2a. ed., Van Nostrand-Reinhold, 1993.
[Kan95] Kaner, C., "Lawyers, Lawsuits, and Quality Related Costs", 1995, disponible en www.badsoftware.com/plaintif.htm.
[Kan01] Kaner, C., "Pattern: Scenario Testing" (draft), 2001, disponible en www.testing.com/test-patterns/patterns/pattern-scenario-testing-kaner.html.
[Kar94] Karten, N., *Managing Expectations*, Dorset House, 1994.
[Kau95] Kauffman, S., *At Home in the Universe*, Oxford, 1995.
[Kaz98] Kazman, R., *et al.*, *The Architectural Tradeoff Analysis Method*, Software Engineering Institute, CMU/SEI-98-TR-008, julio 1998.
[Kaz03] Kazman, R. y A. Eden, "Defining the Terms Architecture, Design, and Implementation", *news@sei interactive*, Software Engineering Institute, vol. 6, núm. 1, 2003, disponible en www.sei.cmu.edu/news-at-sei/columns/the_architect/2003/1q03/architect-1q03.htm.
[Kei98] Keil, M., *et al.*, "A Framework for Identifying Software Project Risks", *CACM*, vol. 41, núm. 11, noviembre 1998, pp. 76-83.
[Kel00] Kelly, D. y R. Oshana, "Improving Software Quality Using Statistical Techniques, Information and Software Technology", Elsevier, vol. 42, agosto 2000, pp. 801-807, disponible en www.eng.auburn.edu/~kchang/comp6710/readings/Improving_Quality_with_Statistical_ Testing_InfoSoftTech_agosto2000.pdf.
[Ker78] Kernighan, B. y P. Plauger, *The Elements of Programming Style*, 2a. ed., McGraw-Hill, 1978.
[Ker05] Kerievsky, J., *Industrial XP: Making XP Work in Large Organizations*, Cutter Consortium, Executive Report, vol. 6., núm. 2, 2005, disponible en www.cutter.com/content-and-analysis/ resource-centers/agile-project-management/sample-our-research/apmr0502.html.
[Kim04] Kim, E., "A Manifesto for Collaborative Tools", *Dr. Dobb's Journal*, mayo 2004, disponible en www.blueoxen.com/papers/0000D/.
[Kir94] Kirani, S. y W. T. Tsai, "Specification and Verification of Object-Oriented Programs", Technical Report TR 94-64, Computer Science Department, University of Minnesota, diciembre 1994.
[Kiz05] Kizza, J., *Computer Network Security*, Springer, 2005.
[Knu98] Knuth, D., *The Art of Computer Programming*, tres volúmenes, Addison-Wesley, 1998.
[Kon02] Konrad, S. y B. Cheng, "Requirements Patterns for Embedded Systems", *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, IEEE, septiembre 2002, pp. 127-136, descargable de http://citeseer.ist.psu.edu/669258.html.

[Kra88] Krasner, G. y S. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80", *Journal of Object-Oriented Programming*, vol. 1, núm. 3, agosto-septiembre 1988, pp. 26-49.
[Kra95] Kraul, R. y L. Streeter, "Coordination in Software Development", *CACM*, vol. 38, núm. 3, marzo 1995, pp. 69-81.
[Kru05] Krutchen, P., "Software Design in a Postmodern Era", *IEEE Software*, vol. 22, núm. 2, marzo-abril 2005, pp. 16-18.
[Kru06] Kruchten, P., H. Obbink y J. Stafford (eds.), "Software Architectural" (número especial), *IEEE Software*, vol. 23, núm. 2, marzo-abril, 2006.
[Kur05] Kurzweil, R., *The Singularity Is Near*, Penguin Books, 2005.
[Kyb84] Kyburg, H. E., *Theory and Measurement*, Cambridge University Press, 1984.
[Laa00] Laakso, S., *et al.*, "Improved Scroll Bars", *CHI 2000 Conf. Proc.*, ACM, 2000, pp. 97-98, disponible en www.cs.helsinki.fi/u/salaakso/patterns/.
[Lai02] Laitenberger, A., "A Survey of Software Inspection Technologies", en *Handbook on Software Engineering and Knowledge Engineering*, World Scientific Publishing Company, 2002.
[Lam01] Lam, W., "Testing E-Commerce Systems: A Practical Guide", *IEEE IT Pro*, marzo-abril 2001, pp. 19-28.
[Lan01] Lange, M., "It's Testing Time! Patterns for Testing Software", junio 2001, descargable de www.testing.com/test-patterns/patterns/index.html.
[Lan02] Land, R., "A Brief Survey of Software Architecture", Technical Report, Dept. of Computer Engineering, Mälardalen University, Suecia, febrero 2002.
[Leh97a] Lehman, M. y L. Belady, *Program Evolution: Processes of Software Change*, Academic Press, 1997.
[Leh97b] Lehman, M., *et al.*, "Metrics and Laws of Software Evolution—The Nineties View", *Proceedings of the 4a. International Software Metrics Symposium (METRICS '97)*, IEEE, 1997, descargable de www.ece.utexas.edu/~perry/work/papers/feast1.pdf.
[Let01] Lethbridge, T. y R. Laganiere, *Object-Oriented Software Engineering: Practical Software Development Using UML and Java*, McGraw-Hill, 2001.
[Let03a] Lethbridge, T., Personal communication on domain analysis, mayo 2003.
[Let03b] Lethbridge, T., Personal communication on software metrics, junio 2003.
[Lev95] Leveson, N. G., *Safeware: System Safety and Computers*, Addison-Wesley, 1995.
[Lev01] Levinson, M., "Let's Stop Wasting $78 billion a Year", *CIO Magazine*, octubre 15, 2001, disponible en www.cio.com/archive/101501/wasting.html.
[Lew06] Lewicki, R., B. Barry y D. Saunders, *Essentials of Negotiation*, McGraw-Hill, 2006.
[Lie03] Lieberherr, K., "Demeter: Aspect-Oriented Programming", mayo 2003, disponible en www.ccs.neu.edu/home/lieber/LoD.html.
[Lin79] Linger, R., H. Mills y B. Witt, *Structured Programming*, Addison-Wesley, 1979.
[Lin88] Linger, R. M. y H. D. Mills, "A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility", *Proc. COMPSAC '88*, Chicago, octubre 1988.
[Lin94] Linger, R., "Cleanroom Process Model", *IEEE Software*, vol. 11, núm. 2, marzo 1994, pp. 50-58.
[Lis88] Liskov, B., "Data Abstraction and Hierarchy", *SIGPLAN Notices*, vol. 23, núm. 5, mayo 1988.
[Liu98] Liu, K., *et al.*, "Report on the First SEBPC Workshop on Legacy Systems", Durham University, febrero 1998, disponible en www.dur.ac.uk/CSM/SABA/legacy-wksp1/report.html.
[Lon02] Longstreet, D., "Fundamental of Function Point Analysis", Longstreet Consulting, Inc., 2002, disponible en www.ifpug.com/fpafund.htm.
[Lor94] Lorenz, M. y J. Kidd, *Object-Oriented Software Metrics*, Prentice Hall, 1994.
[Maa07] Maassen, O. y S. Stelting, "Creational Patterns: Creating Objects in an OO System", 2007, disponible en www.informit.com/articles/article.asp?p=26452&rl=1.
[Man81] Mantai, M., "The Effect of Programming Team Structures on Programming Tasks", *CACM*, vol. 24, núm. 3, marzo 1981, pp. 106-113.
[Man97] Mandel, T., *The Elements of User Interface Design*, Wiley, 1997.
[Mar94] Marick, B., *The Craft of Software Testing*, Prentice Hall, 1994.
[Mar00] Martin, R., "Design Principles and Design Patterns", descargable de www .objectmentor.com, 2000.
[Mar01] Marciniak, J. J. (ed.), *Encyclopedia of Software Engineering*, 2a. ed., Wiley, 2001.
[Mar02] Marick, B., "Software Testing Patterns", 2002, www.testing.com/test-patterns/ index.html.
[McC76] McCabe, T., "A Software Complexity Measure", *IEEE Trans. Software Engineering*, vol. SE-2, diciembre 1976, pp. 308-320.
[McC77] McCall, J., P. Richards y G. Walters, "Factors in Software Quality", tres volúmenes, NTIS AD-A049-014, 015, 055, noviembre 1977.
[McC94] McCabe, T. J. y A. H. Watson, "Software Complexity", *CrossTalk*, vol. 7, núm. 12, diciembre 1994, pp. 5-9.
[McC96] McConnell, S., "Best Practices: Daily Build and Smoke Test", *IEEE Software*, vol. 13, núm. 4, julio 1996, pp. 143-144.
[McC98] McConnell, S., *Software Project Survival Guide*, Microsoft Press, 1998.
[McC99] McConnell, S., "Software Engineering Principles", *IEEE Software*, vol. 16, núm. 2, marzo-abril 1999, disponible en www.stevemcconnell.com/ieeesoftware/eic04.htm.
[McC04] McConnell, S., *Code Complete*, Microsoft Press, 2004.

[McC05] McCrory, A., "Ten Technologies to Watch in 2006", SeachCIO.com, octubre 27, 2005, disponible en http://searchcio.techtarget.com/originalContent/0,289142,sid19_gci1137889,00.html.
[McDE93] McDermid, J. y P. Rook, "Software Development Process Models", en *Software Engineer's Reference Book*, CRC Press, 1993, pp. 15/26-15/28.
[McG91] McGlaughlin, R., "Some Notes on Program Design", *Software Engineering Notes*, vol. 16, núm. 4, octubre 1991, pp. 53-54.
[McG94] McGregor, J. D. y T. D. Korson, "Integrated Object-Oriented Testing and Development Processes", *Communications of the ACM*, vol. 37, núm. 9, septiembre, 1994, pp. 59-77.
[Men01] Mendes, E., N. Mosley y S. Counsell, "Estimating Design and Authoring Effort", *IEEE Multimedia*, vol. 8, núm. 1, enero-marzo 2001, pp. 50-57.
[Mer93] Merlo, E., *et al.*, "Reengineering User Interfaces", *IEEE Software*, enero 1993, pp. 64-73.
[Mic08] *Microsoft Accessibility Technology for Everyone*, 2008, disponible en www.microsoft.com/enable/.
[Mic04] Microsoft, "Prescriptive Architecture: Integration and Patterns", *MSDN*, mayo 2004, disponible en http://msdn2.microsoft.com/en-us/library/ms978700.aspx.
[Mic07] Microsoft, "Patterns and Practices", *MSDN*, 2007, disponible en http://msdn2.microsoft .com/en-us/library/ms998478.aspx.
[Mil72] Mills, H. D., "Mathematical Foundations for Structured Programming", Technical Report FSC 71-6012, IBM Corp., Federal Systems Division, Gaithersburg, MD, 1972.
[Mil77] Miller, E., "The Philosophy of Testing", en *Program Testing Techniques*, IEEE Computer Society Press, 1977, pp. 1-3.
[Mil87] Mills, H. D., M. Dyer y R. Linger, "Cleanroom Software Engineering", *IEEE Software*, septiembre 1987, pp. 19-25.
[Mil88] Mills, H. D., "Stepwise Refinement and Verification in Box Structured Systems", *Computer*, vol. 21, núm. 6, junio 1988, pp. 23-35.
[Mil00a] Miller, E., "WebSite Testing", 2000, disponible en www.soft.com/eValid/Technology/ White.Papers/website.testing.html.
[Mil00b] Mili, A. y R. Cowan, "Software Engineering Technology Watch", abril 6, 2000, disponible en www.serc.net/projects/TechWatch/NSF%20TechWatch%20Proposal.htm.
[Min95] Minoli, D., *Analyzing Outsourcing*, McGraw-Hill, 1995.
[Mon84] Monk, A. (ed.), *Fundamentals of Human-Computer Interaction*, Academic Press, 1984.
[Mor81] Moran, T. P., "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems", *Intl. Journal of Man-Machine Studies*, vol. 15, pp. 3-50.
[Mor05] Morales, A., "The Dream Team", *Dr. Dobbs Portal*, marzo 3, 2005, disponible en www.ddj .com/dept/global/184415303.
[Mus87] Musa, J. D., A. Iannino y K. Okumoto, *Engineering and Managing Software with Reliability Measures*, McGraw-Hill, 1987.
[Mus93] Musa, J., "Operational Profiles in Software Reliability Engineering", *IEEE Software*, marzo 1993, pp. 14-32.
[Mut03] Mutafelija, B. y H. Stromberg, *Systematic Process Improvement Using ISO 9001:2000 and CMMI*, Artech, 2003.
[Mye78] Myers, G., *Composite Structured Design*, Van Nostrand, 1978.
[Mye79] Myers, G., *The Art of Software Testing*, Wiley, 1979.
[NAS07] NASA, *Software Risk Checklist*, Form LeR-F0510.051, marzo 2007, descargable de http://osat-ext.grc.nasa.gov/rmo/spa/SoftwareRiskChecklist.doc.
[Nau69] Naur, P. y B. Randall (eds.), *Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee*, NATO, 1969.
[Ngu00] Nguyen, H., "Testing Web-Based Applications", *Software Testing and Quality Engineering*, mayo-junio 2000, disponible en www.stqemagazine.com.
[Ngu01] Nguyen, H., *Testing Applications on the Web*, Wiley, 2001.
[Ngu06] Nguyen, T., "Model-Based Version and Configuration Management for a Web Engineering Lifecycle", *Proc. 15th Intl. World Wide Web Conf.*, Edinburgo, Escocia, 2006, descargable de www2006.org/programme/item.php?id=4552.
[Nie92] Nierstrasz, O., S. Gibbs y D. Tsichritzis, "Component-Oriented Software Development," *CACM*, vol. 35, núm. 9, septiembre 1992, pp. 160-165.
[Nie94] Nielsen, J. y J. Levy, "Measuring Usability: Preference vs. Performance", *CACM*, vol. 37, núm. 4, abril 1994, pp. 65-75.
[Nie96] Nielsen, J. y A. Wagner, "User Interface Design for the WWW", *Proc. CHI '96 Conf. on Human Factors in Computing Systems*, ACM Press, 1996, pp. 330-331.
[Nie00] Nielsen, J., *Designing Web Usability*, New Riders Publishing, 2000.
[Nog00} Nogueira, J., C. Jones y Luqi, "Surfing the Edge of Chaos: Applications to Software Engineering", Command and Control Research and Technology Symposium, Naval Post Graduate School, Monterey, CA, junio 2000, descargable de www.dodccrp.org/ 2000CCRTS/cd/html/pdf_papers/Track_4/075.pdf.
[Nor70] Norden, P., "Useful Tools for Project Management" en *Management of Production*, M. K. Starr (ed.), Penguin Books, 1970.

[Nor86] Norman, D. A., "Cognitive Engineering", en *User Centered Systems Design*, Lawrence Earlbaum Associates, 1986.
[Nor88] Norman, D., *The Design of Everyday Things*, Doubleday, 1988.
[Nov04] Novotny, O., "Next Generation Tools for Object-Oriented Development", *The Architecture Journal*, enero 2005, disponible en http://msdn2.microsoft.com/en-us/library/aa480062.aspx.
[Noy02] Noyes, B., "Rugby, Anyone?" *Managing Development* (an online publication of Fawcette Technical Publications), junio 2002, www.fawcette.com/resources/managingdev/methodologies/scrum/.
[Off02] Offutt, J., "Quality Attributes of Web Software Applications", *IEEE Software*, marzo-abril 2002, pp. 25-32.
[Ols99] Olsina, L., *et al.*, "Specifying Quality Characteristics and Attributes for Web Sites", *Proc. 1st ICSE Workshop on Web Engineering*, ACM, Los Angeles, mayo 1999.
[Ols06] Olsen, G., "From COM to Common", *Component Technologies*, ACM, vol. 4, núm. 5, junio 2006, disponible en http://acmqueue.com/modules.php?name=Content&pa=showpage& pid=394.
[OMG03a] Object Management Group, *OMG Unified Modeling Language Specification*, version 1.5, marzo 2003, disponible en www.rational.com/uml/resources/documentation/.
[OMG03b] "Object Constraint Language Specification", en *Unified Modeling Language*, v2.0, Object Management Group, septiembre 2003, descargable de www.omg.org.
[Orf99] Orfali, R., D. Harkey y J. Edwards, *Client/Server Survival Guide*, 3a. ed., Wiley, 1999.
[Osb90] Osborne, W. M. y E. J. Chikofsky, "Fitting Pieces to the Maintenance Puzzle", *IEEE Software*, enero 1990, pp. 10-11.
[OSO08] OpenSource.org, 2008, disponible en www.opensource.org/.
[Pag85] Page-Jones, M., *Practical Project Management*, Dorset House, 1985, p. vii.
[Pal02] Palmer, S. y J. Felsing, *A Practical Guide to Feature Driven Development*, Prentice Hall, 2002.
[Par72] Parnas, D. L., "On Criteria to Be Used in Decomposing Systems into Modules", *CACM*, vol. 14, núm. 1, abril 1972, pp. 221-227.
[Par96a] Pardee, W., *To Satisfy and Delight Your Customer*, Dorset House, 1996.
[Par96b] Park, R. E., W. B. Goethert y W. A. Florac, *Goal Driven Software Measurement—A Guidebook*, CMU/SEI-96-BH-002, Software Engineering Institute, Carnegie Mellon University, agosto 1996.
[Pat07] Patton, J., "Understanding User Centricity", *IEEE Software*, vol. 24, núm. 6, noviembre-diciembre, 2007, pp. 9-11.
[Pau94] Paulish, D. y A. Carleton, "Case Studies of Software Process Improvement Measurement", *Computer*, vol. 27, núm. 9, septiembre 1994, pp. 50-57.
[PCM03] "Technologies to Watch", *PC Magazine*, julio 2003, disponible en www.pcmag.com/article2/0,4149, 1130591,00.asp.
[Per74] Persig, R., *Zen and the Art of Motorcycle Maintenance*, Bantam Books, 1974.
[Pet06] Pethokoukis, J., "Small Biz Watch: Future Business Trends", *U.S. News & World Report*, enero 20, 2006, disponible en www.usnews.com/usnews/biztech/articles/060120/ 20sbw.htm.
[Pha89] Phadke, M. S., *Quality Engineering Using Robust Design*, Prentice Hall, 1989.
[Pha97] Phadke, M. S., "Planning Efficient Software Tests", *CrossTalk*, vol. 10, núm. 10, octubre 1997, pp. 11-15.
[Phi98] Phillips, D., *The Software Project Manager's Handbook*, IEEE Computer Society Press, 1998.
[Phi02] Phillips, M., "CMMI V1.1 Tutorial.", abril 2002, disponible en www.sei.cmu.edu/cmmi/.
[Pol45] Polya, G., *How to Solve It*, Princeton University Press, 1945.
[Poo88] Poore, J. H. y H. D. Mills, "Bringing Software Under Statistical Quality Control", *Quality Progress*, noviembre 1988, pp. 52-55.
[Poo93] Poore, J. H., H. D. Mills y D. Mutchler, "Planning and Certifying Software System Reliability", *IEEE Software*, vol. 10, núm. 1, enero 1993, pp. 88-99.
[Pop03] Poppendieck, M. y T. Poppendieck, *Lean Software Development*, Addison-Wesley, 2003.
[Pop06a] Poppendeick, LLC, *Lean Software Development*, disponible en www.poppendieck.com/.
[Pop06b] Poppendieck, M. y T. Poppendieck, *Implementing Lean Software Development*, Addison-Wesley, 2006.
[Pop08] Popcorn, F., *Faith Popcorn's Brain Reserve*, 2008, disponible en www.faithpopcorn.com/.
[Pot04] Potter, M., *Set Theory and Its Philosophy: A Critical Introduction*, Oxford University Press, 2004.
[Pow98] Powell, T., *Web Site Engineering*, Prentice Hall, 1998.
[Pow02] Powell, T., *Web Design*, 2a. ed., McGraw-Hill/Osborne, 2002.
[Pre94] Premerlani, W. y M. Blaha, "An Approach for Reverse Engineering of Relational Databases", *CACM*, vol. 37, núm. 5, mayo 1994, pp. 42-49.
[Pre88] Pressman, R., *Making Software Engineering Happen*, Prentice Hall, 1988.
[Pre05] Pressman, R., *Adaptable Process Model*, revision 2.0, R. S. Pressman & Associates, 2005, disponible en www.rspa.com/apm/index.html.
[Pre08] Pressman, R. y D. Lowe, *Web Engineering: A Practitioner's Approach*, McGraw-Hill, 2008.
[Put78] Putnam, L., "A General Empirical Solution to the Macro Software Sizing and Estimation Problem", *IEEE Trans. Software Engineering*, vol. SE-4, núm. 4, julio 1978, pp. 345-361.
[Put92] Putnam, L. y W. Myers, *Measures for Excellence*, Yourdon Press, 1992.

[Put97a] Putnam, L. y W. Myers, "How Solved Is the Cost Estimation Problem?" *IEEE Software*, noviembre 1997, pp. 105-107.
[Put97b] Putnam, L. y W. Myers, *Industrial Strength Software: Effective Management Using Measurement*, IEEE Computer Society Press, 1997.
[Pyz03] Pyzdek, T., *The Six Sigma Handbook*, McGraw-Hill, 2003.
[QAI08] *A Software Engineering Curriculum*, QAI, 2008, información obtenida en www .qaieschool.com/innerpages/offer.asp.
[QSM02] "QSM Function Point Language Gearing Factors", Version 2.0, Quantitative Software Management, 2002, www.qsm.com/FPGearing.html.
[Rad02] Radice, R., *High-Quality Low Cost Software Inspections*, Paradoxicon Publishing, 2002.
[Rai06] Raiffa, H., *The Art and Science of Negotiation*, Belknap Press, 2005.
[Ree99] Reel, J. S., "Critical Success Factors in Software Projects", *IEEE Software*, mayo 1999, pp. 18-23.
[Ric01] Ricadel, A., "The State of Software Quality", *InformationWeek*, mayo 21, 2001, disponible en www.informationweek.com/838/quality.htm.
[Ric04] Rico, D., *ROI of Software Process Improvement*, J. Ross Publishing, 2004. Se puede encontrar un artículo resumido en http://davidfrico.com/rico03a.pdf.
[Roc94] Roche, J. M., "Software Metrics and Measurement Principles", *Software Engineering Notes*, ACM, vol. 19, núm. 1, enero 1994, pp. 76-85.
[Roc06] *Graphic Design That Works*, Rockport Publishers, 2006.
[Roe00] Roetzheim, W., "Estimating Internet Development", *Software Development*, agosto 2000, disponible en www.sdmagazine.com/documents/s=741/sdm0008d/0008d.htm.
[Roo96] Roos, J., "The Poised Organization: Navigating Effectively on Knowledge Landscapes", 1996, disponible en www.imd.ch/fac/roos/paper_po.html.
[Ros75] Ross, D., J. Goodenough y C. Irvine, "Software Engineering: Process, Principles and Goals", *IEEE Computer*, vol. 8, núm. 5, mayo 1975.
[Ros04] Rosenhainer, L., "Identifying Crosscutting Concerns in Requirements Specifications", 2004, disponible en http://trese.cs.utwente.nl/workshops/oopsla-early-aspects-2004/ Papers/Rosenhainer.pdf.
[Rou02] Rout, T (project manager), *SPICE: Software Process Assessment—Part 1: Concepts and Introductory Guide*, 2002, descargable de www.sqi.gu.edu.au/spice/suite/download .html.
[Roy70] Royce, W. W., "Managing the Development of Large Software Systems: Concepts and Techniques", *Proc. WESCON*, agosto 1970.
[Roz05] Rozanski, N. y E. Woods, *Software Systems Architecture*, Addison-Wesley, 2005.
[Rub88] Rubin, T., *User Interface Design for Computer Systems*, Halstead Press (Wiley), 1988.
[Rum91] Rumbaugh, J., *et al.*, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
[Sar06] Sarwate, A., "Hot or Not: Web Application Vulnerabilities", *SC Magazine*, diciembre 27, 2006, disponible en http://scmagazine.com/us/news/article/623765/hot-not-web-application-vulnerabilities.
[Sca00] Scacchi, W., "Understanding Software Process Redesign Using Modeling, Analysis, and Simulation", *Software Process Improvement and Practice*, Wiley, 2000, pp. 185-195, descargable de www.ics.uci.edu/~wscacchi/Papers/Software_Process_Redesign/ SPIP-ProSim99.pdf.
[Sce02] Sceppa, D., *Microsoft ADO.NET*, Microsoft Press, 2002.
[Sch95] Schwabe, D. y G. Rossi, "The Object-Oriented Hypermedia Design Model", *CACM*, vol. 38, núm. 8, agosto 1995, pp. 45-46.
[Sch96] Schorsch, T., "The Capability Im-Maturity Model", *CrossTalk*, noviembre 1996, disponible en www.stsc.hill.af.mil/crosstalk/1996/11/xt96d11h.asp.
[Sch98a] Schneider, G. y J. Winters, *Applying Use Cases*, Addison-Wesley, 1998.
[Sch98b] Schwabe, D. y G. Rossi, "Developing Hypermedia Applications Using OOHDM", *Proc. Workshop on Hypermedia Development Process, Methods and Models, Hypertext '98*, 1998, descargable de http://citeseer.nj.nec.com/schwabe98developing.html.
[Sch98c] Schulmeyer, G. C. y J. I. McManus (eds.), *Handbook of Software Quality Assurance*, 3a. ed., Prentice Hall, 1998.
[Sch99] Schneidewind, N., "Measuring and Evaluating Maintenance Process Using Reliability, Risk, and Test Metrics", *IEEE Trans. SE*, vol. 25, núm. 6, noviembre-diciembre 1999, pp. 768-781, descargable de www.dacs.dtic.mil/topics/reliability/IEEETrans.pdf.
[Sch01a] Schwabe, D., G. Rossi y Barbosa, S., "Systematic Hypermedia Application Design Using OOHDM", 2001, disponible en www-di.inf.puc-rio.br/~schwabe/HT96WWW/section1.html.
[Sch01b] Schwaber, K. y M. Beedle, *Agile Software Development with SCRUM*, Prentice Hall, 2001.
[Sch02] Schwaber, K., "Agile Processes and Self-Organization", Agile Alliance, 2002, www.aanpo.org/articles/index.
[Sch03] Schlickman, J., *ISO 9001: 2000 Quality Management System Design*, Artech House Publishers, 2003.
[Sch06] Schmidt, D., "Model-Driven Engineering", *IEEE Computer*, vol. 39, núm. 2, febrero 2006, pp. 25-31.
[SDS08] Spice Document Suite, "The SPICE and ISO Document Suite", ISO-Spice, 2008, disponible en www.isospice.com/articles/9/1/SPICE-Project/Page1.html.
[Sea93] Sears, A., "Layout Appropriateness: A Metric for Evaluating User Interface Widget Layout, *IEEE Trans. Software Engineering*, vol. SE-19, núm. 7, julio 1993, pp. 707-719.

[SEE03] The Software Engineering Ethics Research Institute, "UCITA Updates", 2003, disponible en http://seeri.etsu.edu/default.htm.
[SEI00] *SCAMPI, V1.0 Standard CMMI ®Assessment Method for Process Improvement: Method Description*, Software Engineering Institute, Technical Report CMU/SEI-2000-TR-009, descargable de www.sei.cmu.edu/publications/documents/00.reports/00tr009.html.
[SEI02] "Maintainability Index Technique for Measuring Program Maintainability", SEI, 2002, disponible en www.sei.cmu.edu/str/descriptions/mitmpm_body.html.
[SEI08] "The Ideal Model", Software Engineering Institute, 2008, disponible en www.sei.cmu .edu/ideal/.
[Sha95a] Shaw, M. y D. Garlan, "Formulations and Formalisms in Software Architecture", *Volume 1000—Lecture Notes in Computer Science*, Springer-Verlag, 1995.
[Sha95b] Shaw, M., *et al.*, "Abstractions for Software Architecture and Tools to Support Them", *IEEE Trans. Software Engineering*, vol. SE-21, núm. 4, abril 1995, pp. 314-335.
[Sha96] Shaw, M. y D. Garlan, *Software Architecture*, Prentice Hall, 1996.
[Sha05] Shalloway, A. y J. Trott, *Design Patterns Explained*, 2a. ed., Addison-Wesley, 2005.
[Shn80] Shneiderman, B., *Software Psychology*, Winthrop Publishers, 1980, p. 28.
[Shn04] Shneiderman, B. y C. Plaisant, *Designing the User Interface*, 4a. ed., Addison-Wesley, 2004.
[Sho83] Shooman, M. L., *Software Engineering*, McGraw-Hill, 1983.
[Sim05] Simsion, G. y G. Witt, *Data Modeling Essentials*, 3a. ed., Morgan Kaufman, 2005.
[Sin99] Singpurwalla, N. y S. Wilson, *Statistical Methods in Software Engineering: Reliability and Risk*, Springer-Verlag, 1999.
[Smi99] Smith, J., "The Estimation of Effort Based on Use Cases", Rational Software Corp., 1999, descargable de www.rational.com/media/whitepapers/finalTP171.PDF.
[Smi05] Smith, D, *Reliability, Maintainability and Risk*, 7a. ed., Butterworth-Heinemann, 2005.
[Sne95] Sneed, H., "Planning the Reengineering of Legacy Systems", *IEEE Software*, enero 1995, pp. 24-25.
[Sne03] Snee, R. y R. Hoerl, *Leading Six Sigma*, Prentice Hall, 2003.
[Sol99] van Solingen, R. y E. Berghout, *The Goal/Question/Metric Method*, McGraw-Hill, 1999.
[Som97] Somerville, I. y P. Sawyer, *Requirements Engineering*, Wiley, 1997.
[Som05] Somerville, I., "Integrating Requirements Engineering: A Tutorial", *IEEE Software*, vol. 22, núm. 1, enero-febrero 2005, pp. 16-23.
[SPI99] "SPICE: Software Process Assessment, Part 1: Concepts and Introduction", Version 1.0, ISO/IEC JTC1, 1999.
[Spl01] Splaine, S. y S. Jaskiel, *The Web Testing Handbook*, STQE Publishing, 2001.
[Spo02] Spolsky, J, "The Law of Leaky Abstractions", noviembre 2002, disponible en www .joelonsoftware.com/articles/LeakyAbstractions.html.
[Sri01] Sridhar, M. y N. Mandyam, "Effective Use of Data Models in Building Web Applications," 2001, disponible en www2002.org/CDROM/alternate/698/.
[SSO08] Software-Supportability.org, www.software-supportability.org/ 2008.
[Sta97] Stapleton, J., *DSDM—Dynamic System Development Method: The Method in Practice*, Addison-Wesley, 1997.
[Sta97b] Statz, J., D. Oxley y P. O'Toole, "Identifying and Managing Risks for Software Process Improvement", *CrossTalk*, abril 1997, disponible en www.stsc.hill.af.mil/crosstalk/1997/ 04/identifying.asp.
[Ste74] Stevens, W., G. Myers y L. Constantine, "Structured Design", *IBM Systems Journal*, vol. 13, núm. 2, 1974, pp. 115-139.
[Ste93] Stewart, T. A., "Reengineering: The Hot New Managing Tool", *Fortune*, agosto 23, 1993, pp. 41-48.
[Ste99] Stelzer, D. y W. Mellis, "Success Factors of Organizational Change in Software Process Improvement", *Software Process Improvement and Practice*, vol. 4, núm. 4, Wiley, 1999, descargable de www.systement-wicklung.uni-koeln.de/forschung/artikel/dokumente/successfactors.pdf.
[Ste03] Stephens, M. y D. Rosenberg, *Extreme Programming Refactored*, Apress, 2003.
[Sto05] Stone, D., *et al.*, *User Interface Design and Evaluation*, Morgan Kaufman, 2005.
[Tai89] Tai, K. C., "What to Do Beyond Branch Testing," *ACM Software Engineering Notes*, vol. 14, núm. 2, abril 1989, pp. 58-61.
[Tay90] Taylor, D., *Object-Oriented Technology: A Manager's Guide*, Addison-Wesley, 1990.
[Tha97] Thayer, R. H. y M. Dorfman, *Software Requirements Engineering*, 2a. ed., IEEE Computer Society Press, 1997.
[The01] Thelin, T., H. Petersson y C. Wohlin, "Sample Driven Inspections", *Proc. of Workshop on Inspection in Software Engineering (WISE'01)*, París, Francia, julio 2001, pp. 81-91, descargable de http://www.cas.mcmaster.ca/wise/wise01/ThelinPeterssonWohlin.pdf.
[Tho92] Thomsett, R., "The Indiana Jones School of Risk Management", *American Programmer*, vol. 5, núm. 7, septiembre 1992, pp. 10-18.
[Tic05] *TickIT*, 2005, www.tickit.org/.
[Tid02] Tidwell, J., "IU Patterns and Techniques", mayo 2002, disponible en http://time-tripper.com/uipatterns/index.html.
[Til93] Tillmann, G., *A Practical Guide to Logical Data Modeling*, McGraw-Hill, 1993.
[Til00] Tillman, H., "Evaluating Quality on the Net", Babson College, mayo 30, 2000, disponible en www.hopetillman.com/findqual.html#2.

[Tog01] Tognozzi, B., "First Principles", *askTOG*, 2001, disponible en www.asktog.com/basics/firstPrinciples.html.
[Tra95] Tracz, W., "Third International Conference on Software Reuse—Summary", *ACM Software Engineering Notes*, vol. 20, núm. 2, abril 1995, pp. 21-22.
[Tre03] Trivedi, R, *Professional Web Services Security*, Wrox Press, 2003.
[Tri05] Tricker, R. y B. Sherring-Lucas, *ISO 9001: 2000 In Brief*, 2a. ed., Butterworth-Heinemann, 2005.
[Tyr05] Tyree, J. y A. Akerman, "Architectural Decisions: Demystifying Architecture", *IEEE Software*, vol. 22, núm. 2, marzo-abril, 2005.
[Uem99] Uemura, T., S. Kusumoto y K. Inoue: "A Function Point Measurement Tool for UML Design Specifications", *Proc. of Sixth International Symposium on Software Metrics*, IEEE, noviembre 1999, pp. 62-69
[Ull97] Ullman, E., *Close to the Machine: Technophilia and Its Discontents*, City Lights Books, 2002.
[UML03] The UML Café, "Customers Don't Print Themselves", mayo 2003, disponible en www.theumlcafe.com/a0079.htm.
[Uni03] Unicode, Inc., *The Unicode Home Page*, 2003, disponible en www.unicode.org/.
[USA87] *Management Quality Insight*, AFCSP 800-14 (U.S. Air Force), enero 20, 1987.
[Vac06] Vacca, J., *Practical Internet Security*, Springer, 2006.
[Van89] Van Vleck, T., "Three Questions About Each Bug You Find", *ACM Software Engineering Notes*, vol. 14, núm. 5, julio 1989, pp. 62-63.
[Van02] Van Steen, M. y A. Tanenbaum, *Distributed Systems: Principles and Paradigms*, Prentice Hall, 2002.
[Ven03] Venners, B., "Design by Contract: A Conversation with Bertrand Meyer", *Artima Developer*, diciembre 8, 2003, disponible en www.artima.com/intv/contracts.html.
[Wal03] Wallace, D., I. Raggett y J. Aufgang, *Extreme Programming for Web Projects*, Addison-Wesley, 2003.
[War74] Warnier, J. D., *Logical Construction of Programs*, Van Nostrand-Reinhold, 1974.
[War07] Ward, M., "Using VoIP Software Building zBlocks—A Look at the Choices", TMNNet, 2007, disponible en www.tmcnet.com/voip/0605/featurearticle-using-voip-software-building-blocks.htm.
[Web05] Weber, S., *The Success of Open Source*, Harvard University Press, 2005.
[Wei86] Weinberg, G., *On Becoming a Technical Leader*, Dorset House, 1986.
[Wel99] Wells, D., "XP—Unit Tests", 1999, disponible en www.extremeprogramming.org/rules/unittests.html.
[Wel01] vanWelie, M., "Interaction Design Patterns", 2001, disponible en www.welie.com/ patterns/.
[Whi95] Whittle, B., "Models and Languages for Component Description and Reuse", *ACM Software Engineering Notes*, vol. 20, núm. 2, abril 1995, pp. 76-89.
[Whi97] Whitmire, S., *Object-Oriented Design Measurement*, Wiley, 1997.
[Wie02] Wiegers, K., *Peer Reviews in Software*, Addison-Wesley, 2002.
[Wie03] Wiegers, K., *Software Requirements*, 2a. ed., Microsoft Press, 2003.
[Wil93] Wilde, N. y R. Huitt, "Maintaining Object-Oriented Software", *IEEE Software*, enero 1993, pp. 75-80.
[Wil97] Williams, R. C, J. A. Walker y A. J. Dorofee, "Putting Risk Management into Practice", *IEEE Software*, mayo 1997, pp. 75-81.
[Wil99] Wilkens, T. T., "Earned Value, Clear and Simple", Primavera Systems, abril 1, 1999, p. 2.
[Wil00] Williams, L. y R. Kessler, "All I Really Need to Know about Pair Programming I Learned in Kindergarten", *CACM*, vol. 43, núm. 5, mayo 2000, disponible en http://collaboration.csc.ncsu.edu/laurie/Papers/Kindergarten.PDF.
[Wil05] Willoughby, M., "Q&A: Quality Software Means More Secure Software", *Computerworld*, marzo 21, 2005, disponible en www.computerworld.com/securitytopics/security/story/ 0,10801,91316,00.html.
[Win90] Wing, J. M., "A Specifier's Introduction to Formal Methods", *IEEE Computer*, vol. 23, núm. 9, septiembre 1990, pp. 8-24.
[Wir71] Wirth, N., "Program Development by Stepwise Refinement", *CACM*, vol. 14, núm. 4, 1971, pp. 221-227.
[Wir90] Wirfs-Brock, R., B. Wilkerson y L. Weiner, *Designing Object-Oriented Software*, Prentice Hall, 1990.
[WMT02] Web Mapping Testbed Tutorial., 2002, disponible en www.webmapping.org/ vcgdocuments/vcg-Tutorial/.
[Woh94] Wohlin, C. y P. Runeson, "Certification of Software Components", *IEEE Trans. Software Engineering*, vol. SE-20, núm. 6, junio 1994, pp. 494-499.
[Wor04] World Bank, *Digital Technology Risk Checklist*, 2004, descargable de www.moonv6.org/lists/att-0223/WWBANK_Technology_Risk_Checklist_Ver_6point1.pdf.
[W3C03] World Wide Web Consortium, *Web Content Accessibility Guidelines*, 2003, disponible en www.w3.org/TR/2003/WD-WCAG20-20030624/.
[Yac03] Yacoub, S., *et al.*, *Pattern-Oriented Analysis and Design*, Addison-Wesley, 2003.
[You75] Yourdon, E., *Techniques of Program Structure and Design*, Prentice Hall, 1975.
[You79] Yourdon, E. y L. Constantine, *Structured Design*, Prentice Hall, 1979.
[You95] Yourdon, E., "When Good Enough Is Best", *IEEE Software*, vol. 12, núm. 3, mayo 1995, pp. 79-81.
[You01] Young, R., *Effective Requirements Practices*, Addison-Wesley, 2001.
[Zah90] Zahniser, R. A., "Building Software in Groups", *American Programmer*, vol. 3, núms. 7-8, julio-agosto 1990.
[Zah94] Zahniser, R., "Timeboxing for Top Team Performance," *Software Development*, marzo 1994, pp. 35-38.

[Zha98] Zhao, J, "On Assessing the Complexity of Software Architectures", *Proc. Intl. Software Architecture Workshop*, ACM, Orlando, FL, 1998, pp. 163-167.
[Zha02] Zhao, H., "Fitt's Law: Modeling Movement Time in HCI", *Theories in Computer Human Interaction*, University of Maryland, octubre 2002, disponible en www.cs.umd.edu/class/fall2002/cmsc838s/tichi/fitts.html.
[Zul92] Zultner, R., "Quality Function Deployment for Software: Satisfying Customers", *American Programmer*, febrero 1992, pp. 28-41.
[Zus90] Zuse, H., *Software Complexity: Measures and Methods*, DeGruyter, 1990.
[Zus97] Zuse, H., *A Framework of Software Measurement*, DeGruyter, 1997.

D

E

R

S

T

U