

# EL SOFTWARE Y LA INGENIERÍA DE SOFTWARE

## CONCEPTOS CLAVE

actividades estructurales . . . .	12
actividades sombrilla . . . . .	12
características del software . . .	3
dominios de aplicación . . . . .	6
ingeniería de software . . . . .	10
mitos del software . . . . .	18
práctica . . . . .	15
principios . . . . .	16
proceso del software . . . . .	12
software heredado . . . . .	8
webapps . . . . .	9

Tenía la apariencia clásica de un alto ejecutivo de una compañía importante de software —a la mitad de los 40, con las sienes comenzando a encanecer, esbelto y atlético, con ojos que penetraban al observador mientras hablaba—. Pero lo que dijo me dejó anonadado. “El software ha muerto”.

Pestañeé con sorpresa y sonreí. “Bromeas, ¿verdad? El mundo es dirigido con software y tu empresa se ha beneficiado mucho de ello. ¡No ha muerto! Está vivo y en desarrollo.”

Movió su cabeza de manera enfática. “No, está muerto... al menos como lo conocimos.”

Me apoyé en el escritorio. “Continúa.”

Habló al tiempo que golpeaba en la mesa con énfasis. “El concepto antiguo del software —lo compras, lo posees y tu trabajo consiste en administrarlo— está llegando a su fin. Hoy día, con Web 2.0 y la computación ubicua cada vez más fuerte, vamos a ver una generación de software por completo diferente. Se distribuirá por internet y se verá exactamente como si estuviera instalado en el equipo de cómputo de cada usuario... pero se encontrará en un servidor remoto.”

Tuve que estar de acuerdo. “Entonces, tu vida será mucho más sencilla. Tus muchachos no tendrán que preocuparse por las cinco diferentes versiones de la misma App que utilizan decenas de miles de usuarios.”

Sonrió. “Absolutamente. Sólo la versión más reciente estará en nuestros servidores. Cuando hagamos un cambio o corrección, actualizaremos funcionalidad y contenido a cada usuario. Todos lo tendrán en forma instantánea...”

Hice una mueca. “Pero si cometes un error, todos lo tendrán también instantáneamente”.

Él se rió entre dientes. “Es verdad, por eso estamos redoblando nuestros esfuerzos para hacer una ingeniería de software aún mejor. El problema es que tenemos que hacerlo ‘rápido’ porque el mercado se ha acelerado en cada área de aplicación.”

## UNA MIRADA RÁPIDA

**¿Qué es?** El software de computadora es el producto que construyen los programadores profesionales y al que después le dan mantenimiento durante un largo tiempo. Incluye programas que se ejecutan en una computadora de cualquier tamaño y arquitectura, contenido que se presenta a medida de que se ejecutan los programas de cómputo e información descriptiva tanto en una copia dura como en formatos virtuales que engloban virtualmente a cualesquiera medios electrónicos. La ingeniería de software está formada por un proceso, un conjunto de métodos (prácticas) y un arreglo de herramientas que permite a los profesionales elaborar software de cómputo de alta calidad.

**¿Quién lo hace?** Los ingenieros de software elaboran y dan mantenimiento al software, y virtualmente cada persona lo emplea en el mundo industrializado, ya sea en forma directa o indirecta.

**¿Por qué es importante?** El software es importante porque afecta a casi todos los aspectos de nuestras vidas y ha invadido nuestro comercio, cultura y actividades cotidia-

nas. La ingeniería de software es importante porque nos permite construir sistemas complejos en un tiempo razonable y con alta calidad.

**¿Cuáles son los pasos?** El software de computadora se construye del mismo modo que cualquier producto exitoso, con la aplicación de un proceso ágil y adaptable para obtener un resultado de mucha calidad, que satisfaga las necesidades de las personas que usarán el producto. En estos pasos se aplica el enfoque de la ingeniería de software.

**¿Cuál es el producto final?** Desde el punto de vista de un ingeniero de software, el producto final es el conjunto de programas, contenido (datos) y otros productos terminados que constituyen el software de computadora. Pero desde la perspectiva del usuario, el producto final es la información resultante que de algún modo hace mejor al mundo en el que vive.

**¿Cómo me aseguro de que lo hice bien?** Lea el resto de este libro, seleccione aquellas ideas que sean aplicables al software que usted hace y aplíquelas a su trabajo.

Me recargué en la espalda y coloqué mis manos en mi nuca. “Ya sabes lo que se dice... puedes tenerlo rápido o bien hecho o barato. Escoge dos de estas características...”

“Elijo rápido y bien hecho”, dijo mientras comenzaba a levantarse.

También me incorporé. “Entonces realmente necesitas ingeniería de software.”

“Ya lo sé”, dijo mientras salía. “El problema es que tenemos que llegar a convencer a otra generación más de técnicos de que así es...”

¿Está muerto *realmente* el software? Si lo estuviera, usted no estaría leyendo este libro...

El software de computadora sigue siendo la tecnología más importante en la escena mundial. Y también es un ejemplo magnífico de la ley de las consecuencias inesperadas. Hace 50 años, nadie hubiera podido predecir que el software se convertiría en una tecnología indispensable para los negocios, ciencias e ingeniería, ni que permitiría la creación de tecnologías nuevas (por ejemplo, ingeniería genética y nanotecnología), la ampliación de tecnologías ya existentes (telecomunicaciones) y el cambio radical de tecnologías antiguas (la industria de la impresión); tampoco que el software sería la fuerza que impulsaría la revolución de las computadoras personales, que productos de software empacados se comprarían en los supermercados, que el software evolucionaría poco a poco de un producto a un servicio cuando compañías de software “sobre pedido” proporcionaran funcionalidad justo a tiempo a través de un navegador web, que una compañía de software sería más grande y tendría más influencia que casi todas las empresas de la era industrial, que una vasta red llamada internet sería operada con software y evolucionaría y cambiaría todo, desde la investigación en bibliotecas y la compra de productos para el consumidor hasta el discurso político y los hábitos de encuentro de los adultos jóvenes (y no tan jóvenes).

Nadie pudo prever que habría software incrustado en sistemas de toda clase: de transporte, médicos, de telecomunicaciones, militares, industriales, de entretenimiento, en máquinas de oficina... la lista es casi infinita. Y si usted cree en la ley de las consecuencias inesperadas, hay muchos efectos que aún no podemos predecir.

Nadie podía anticipar que millones de programas de computadora tendrían que ser corregidos, adaptados y mejorados a medida que transcurriera el tiempo. Ni que la carga de ejecutar estas actividades de “mantenimiento” absorbería más personas y recursos que todo el trabajo aplicado a la creación de software nuevo.

Conforme ha aumentado la importancia del software, la comunidad de programadores ha tratado continuamente de desarrollar tecnologías que hagan más fácil, rápida y barata la elaboración de programas de cómputo de alta calidad. Algunas de estas tecnologías se dirigen a un dominio específico de aplicaciones (por ejemplo, diseño e implantación de un sitio web), otras se centran en un dominio tecnológico (sistemas orientados a objetos o programación orientada a aspectos), otros más tienen una base amplia (sistemas operativos, como Linux). Sin embargo, todavía falta por desarrollarse una tecnología de software que haga todo esto, y hay pocas probabilidades de que surja una en el futuro. A pesar de ello, las personas basan sus trabajos, confort, seguridad, diversiones, decisiones y sus propias vidas en software de computadora. Más vale que esté bien hecho.

Este libro presenta una estructura que puede ser utilizada por aquellos que hacen software de cómputo —personas que deben hacerlo bien—. La estructura incluye un proceso, un conjunto de métodos y unas herramientas que llamamos *ingeniería de software*.

**Cita:**

“Las ideas y los descubrimientos tecnológicos son los motores que impulsan el crecimiento económico.”

Wall Street Journal

## 1.1 LA NATURALEZA DEL SOFTWARE

En la actualidad, el software tiene un papel dual. Es un producto y al mismo tiempo es el vehículo para entregar un producto. En su forma de producto, brinda el potencial de cómputo incorporado en el hardware de cómputo o, con más amplitud, en una red de computadoras a las

## PUNTO CLAVE

El software es tanto un producto como un vehículo para entregar un producto.

### Cita:

“El software es un lugar donde se siembran sueños y se cosechan pesadillas, una ciénega abstracta y mística en la que terribles demonios luchan contra panaceas mágicas, un mundo de hombres lobo y balas de plata.”

Brad J. Cox

que se accede por medio de un hardware local. Ya sea que resida en un teléfono móvil u opere en el interior de una computadora central, el software es un transformador de información—produce, administra, adquiere, modifica, despliega o transmite información que puede ser tan simple como un solo bit o tan compleja como una presentación con multimedios generada a partir de datos obtenidos de decenas de fuentes independientes—. Como vehículo utilizado para distribuir el producto, el software actúa como la base para el control de la computadora (sistemas operativos), para la comunicación de información (redes) y para la creación y control de otros programas (herramientas y ambientes de software).

El software distribuye el producto más importante de nuestro tiempo: *información*. Transforma los datos personales (por ejemplo, las transacciones financieras de un individuo) de modo que puedan ser más útiles en un contexto local, administra la información de negocios para mejorar la competitividad, provee una vía para las redes mundiales de información (la internet) y brinda los medios para obtener información en todas sus formas.

En el último medio siglo, el papel del software de cómputo ha sufrido un cambio significativo. Las notables mejoras en el funcionamiento del hardware, los profundos cambios en las arquitecturas de computadora, el gran incremento en la memoria y capacidad de almacenamiento, y una amplia variedad de opciones de entradas y salidas exóticas han propiciado la existencia de sistemas basados en computadora más sofisticados y complejos. Cuando un sistema tiene éxito, la sofisticación y complejidad producen resultados deslumbrantes, pero también plantean problemas enormes para aquellos que deben construir sistemas complejos.

En la actualidad, la enorme industria del software se ha convertido en un factor dominante en las economías del mundo industrializado. Equipos de especialistas de software, cada uno centrado en una parte de la tecnología que se requiere para llegar a una aplicación compleja, han reemplazado al programador solitario de los primeros tiempos. A pesar de ello, las preguntas que se hacía aquel programador son las mismas que surgen cuando se construyen sistemas modernos basados en computadora:<sup>1</sup>

- ¿Por qué se requiere tanto tiempo para terminar el software?
- ¿Por qué son tan altos los costos de desarrollo?
- ¿Por qué no podemos detectar todos los errores antes de entregar el software a nuestros clientes?
- ¿Por qué dedicamos tanto tiempo y esfuerzo a mantener los programas existentes?
- ¿Por qué seguimos con dificultades para medir el avance mientras se desarrolla y mantiene el software?

Éstas y muchas otras preguntas, denotan la preocupación sobre el software y la manera en que se desarrolla, preocupación que ha llevado a la adopción de la práctica de la ingeniería del software.

### 1.1.1 Definición de software

En la actualidad, la mayoría de profesionales y muchos usuarios tienen la fuerte sensación de que entienden el software. Pero, ¿es así?

La descripción que daría un libro de texto sobre software sería más o menos así:

**El software es:** 1) instrucciones (programas de cómputo) que cuando se ejecutan proporcionan las características, función y desempeño buscados; 2) estructuras de datos que permiten que los progra-

## ? ¿Cómo se define software?

<sup>1</sup> En un excelente libro de ensayos sobre el negocio del software, Tom DeMarco [DeM95] defiende el punto de vista contrario. Dice: “En lugar de preguntar por qué el software cuesta tanto, necesitamos comenzar a preguntar: ¿Qué hemos hecho para hacer posible que el software actual cueste tan poco? La respuesta a esa pregunta nos ayudará a continuar el extraordinario nivel de logro que siempre ha distinguido a la industria del software.”

mas manipulen en forma adecuada la información, y 3) información descriptiva tanto en papel como en formas virtuales que describen la operación y uso de los programas.

No hay duda de que podrían darse definiciones más completas.

Pero es probable que una definición más formal no mejore de manera apreciable nuestra comprensión. Para asimilar lo anterior, es importante examinar las características del software que lo hacen diferente de otros objetos que construyen los seres humanos. El software es elemento de un sistema lógico y no de uno físico. Por tanto, tiene características que difieren considerablemente de las del hardware:

1. *El software se desarrolla o modifica con intelecto; no se manufactura en el sentido clásico.*

Aunque hay algunas similitudes entre el desarrollo de software y la fabricación de hardware, las dos actividades son diferentes en lo fundamental. En ambas, la alta calidad se logra a través de un buen diseño, pero la fase de manufactura del hardware introduce problemas de calidad que no existen (o que se corrigen con facilidad) en el software. Ambas actividades dependen de personas, pero la relación entre los individuos dedicados y el trabajo logrado es diferente por completo (véase el capítulo 24). Las dos actividades requieren la construcción de un "producto", pero los enfoques son distintos. Los costos del software se concentran en la ingeniería. Esto significa que los proyectos de software no pueden administrarse como si fueran proyectos de manufactura.

2. *El software no se "desgasta".*

La figura 1.1 ilustra la tasa de falla del hardware como función del tiempo. La relación, que es frecuente llamar "curva de tina", indica que el hardware presenta una tasa de fallas relativamente elevada en una etapa temprana de su vida (fallas que con frecuencia son atribuibles a defectos de diseño o manufactura); los defectos se corrigen y la tasa de fallas se abate a un nivel estable (muy bajo, por fortuna) durante cierto tiempo. No obstante, conforme pasa el tiempo, la tasa de fallas aumenta de nuevo a medida que los componentes del hardware resienten los efectos acumulativos de suciedad, vibración, abuso, temperaturas extremas y muchos otros inconvenientes ambientales. En pocas palabras, el hardware comienza a *desgastarse*.

El software no es susceptible a los problemas ambientales que hacen que el hardware se desgaste. Por tanto, en teoría, la curva de la tasa de fallas adopta la forma de la "curva idealizada" que se aprecia en la figura 1.2. Los defectos ocultos ocasionarán ta-

### PUNTO CLAVE

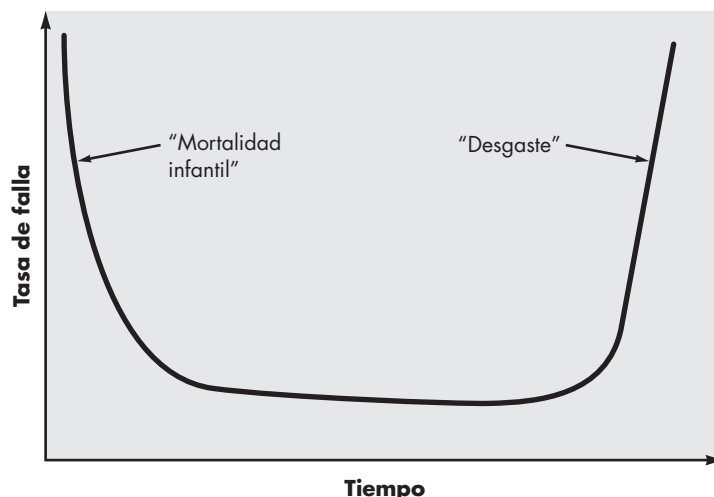
El software se modifica con intelecto, no se manufactura.

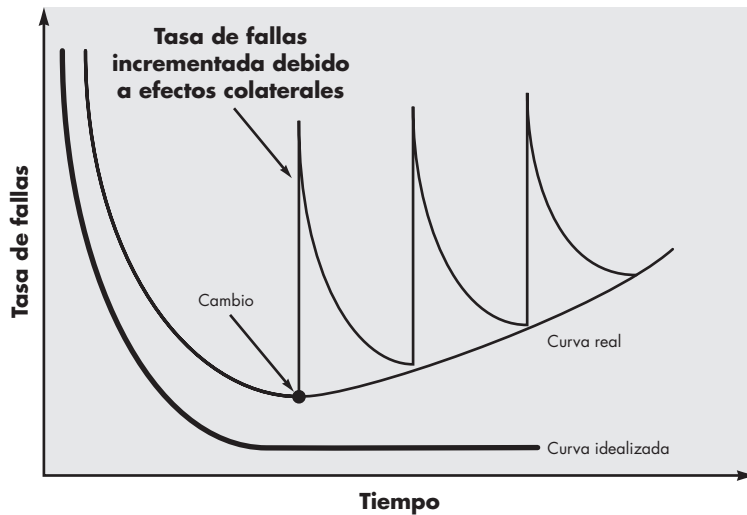
### PUNTO CLAVE

El software no se desgasta, pero sí se deteriora.

**FIGURA 1.1**

Curva de fallas del hardware



**FIGURA 1.2****Curvas de falla del software**

Si quiere reducir el deterioro del software, tendrá que mejorar su diseño (capítulos 8 a 13).



Los métodos de la ingeniería de software llevan a reducir la magnitud de los picos y de la pendiente de la curva real en la figura 1.2.

Las elevadas tasas de fallas al comienzo de la vida de un programa. Sin embargo, éstas se corrigen y la curva se aplanan, como se indica. La curva idealizada es una gran simplificación de los modelos reales de las fallas del software. Aun así, la implicación está clara: el software no se desgasta, ¡pero sí se *deteriora*!

Esta contradicción aparente se entiende mejor si se considera la curva real en la figura 1.2. Durante su vida,<sup>2</sup> el software sufrirá cambios. Es probable que cuando éstos se realicen, se introduzcan errores que ocasionen que la curva de tasa de fallas tenga aumentos súbitos, como se ilustra en la “curva real” (véase la figura 1.2). Antes de que la curva vuelva a su tasa de fallas original de estado estable, surge la solicitud de otro cambio que hace que la curva se dispare otra vez. Poco a poco, el nivel mínimo de la tasa de fallas comienza a aumentar: el software se está deteriorando como consecuencia del cambio.

Otro aspecto del desgaste ilustra la diferencia entre el hardware y el software. Cuando un componente del hardware se desgasta es sustituido por una refacción. En cambio, no hay refacciones para el software. Cada falla de éste indica un error en el diseño o en el proceso que tradujo el diseño a código ejecutable por la máquina. Entonces, las tareas de mantenimiento del software, que incluyen la satisfacción de peticiones de cambios, involucran una complejidad considerablemente mayor que el mantenimiento del hardware.

3. Aunque la industria se mueve hacia la construcción basada en componentes, la mayor parte del software se construye para un uso individualizado.

A medida que evoluciona una disciplina de ingeniería, se crea un conjunto de componentes estandarizados para el diseño. Los tornillos estándar y los circuitos integrados preconstruidos son sólo dos de los miles de componentes estándar que utilizan los ingenieros mecánicos y eléctricos conforme diseñan nuevos sistemas. Los componentes reutilizables han sido creados para que el ingeniero pueda concentrarse en los elementos verdaderamente innovadores de un diseño; es decir, en las partes de éste que representan algo nuevo. En el mundo del hardware, volver a usar componentes es una parte

**Cita:**

“Las ideas son los ladrillos con los que se construyen las ideas.”

Jason Zebehazy

<sup>2</sup> En realidad, los distintos participantes solicitan cambios desde el momento en que comienza el desarrollo y mucho antes de que se disponga de la primera versión.

natural del proceso de ingeniería. En el del software, es algo que apenas ha empezado a hacerse a gran escala.

Un componente de software debe diseñarse e implementarse de modo que pueda volverse a usar en muchos programas diferentes. Los modernos componentes reutilizables incorporan tanto los datos como el procesamiento que se les aplica, lo que permite que el ingeniero de software cree nuevas aplicaciones a partir de partes susceptibles de volverse a usar.<sup>3</sup> Por ejemplo, las actuales interfaces interactivas de usuario se construyen con componentes reutilizables que permiten la creación de ventanas gráficas, menús desplegables y una amplia variedad de mecanismos de interacción. Las estructuras de datos y el detalle de procesamiento que se requieren para construir la interfaz están contenidos en una librería de componentes reusables para tal fin.

### 1.1.2 Dominios de aplicación del software

Actualmente, hay siete grandes categorías de software de computadora que plantean retos continuos a los ingenieros de software:

**Software de sistemas:** conjunto de programas escritos para dar servicio a otros programas. Determinado software de sistemas (por ejemplo, compiladores, editores y herramientas para administrar archivos) procesa estructuras de información complejas pero deterministas.<sup>4</sup> Otras aplicaciones de sistemas (por ejemplo, componentes de sistemas operativos, manejadores, software de redes, procesadores de telecomunicaciones) procesan sobre todo datos indeterminados. En cualquier caso, el área de software de sistemas se caracteriza por: gran interacción con el hardware de la computadora, uso intensivo por parte de usuarios múltiples, operación concurrente que requiere la secuenciación, recursos compartidos y administración de un proceso sofisticado, estructuras complejas de datos e interfaces externas múltiples.

**Software de aplicación:** programas aislados que resuelven una necesidad específica de negocios. Las aplicaciones en esta área procesan datos comerciales o técnicos en una forma que facilita las operaciones de negocios o la toma de decisiones administrativas o técnicas. Además de las aplicaciones convencionales de procesamiento de datos, el software de aplicación se usa para controlar funciones de negocios en tiempo real (por ejemplo, procesamiento de transacciones en punto de venta, control de procesos de manufactura en tiempo real).

**Software de ingeniería y ciencias:** se ha caracterizado por algoritmos “devoradores de números”. Las aplicaciones van de la astronomía a la vulcanología, del análisis de tensiones en automóviles a la dinámica orbital del transbordador espacial, y de la biología molecular a la manufactura automatizada. Sin embargo, las aplicaciones modernas dentro del área de la ingeniería y las ciencias están abandonando los algoritmos numéricos convencionales. El diseño asistido por computadora, la simulación de sistemas y otras aplicaciones interactivas, han comenzado a hacerse en tiempo real e incluso han tomado características del software de sistemas.

**Software incrustado:** reside dentro de un producto o sistema y se usa para implementar y controlar características y funciones para el usuario final y para el sistema en sí. El software incrustado ejecuta funciones limitadas y particulares (por ejemplo, control del tablero de un horno de microondas) o provee una capacidad significativa de funcionamiento y control

#### WebRef

En la dirección [shareware.cnet.com](http://shareware.cnet.com) se encuentra una de las librerías más completas de software compartido y libre.

<sup>3</sup> El desarrollo basado en componentes se estudia en el capítulo 10.

<sup>4</sup> El software es *determinista* si es posible predecir el orden y momento de las entradas, el procesamiento y las salidas. El software es *no determinista* si no pueden predecirse el orden y momento en que ocurren éstos.

(funciones digitales en un automóvil, como el control del combustible, del tablero de control y de los sistemas de frenado).

**Software de línea de productos:** es diseñado para proporcionar una capacidad específica para uso de muchos consumidores diferentes. El software de línea de productos se centra en algún mercado limitado y particular (por ejemplo, control del inventario de productos) o se dirige a mercados masivos de consumidores (procesamiento de textos, hojas de cálculo, gráficas por computadora, multimedia, entretenimiento, administración de base de datos y aplicaciones para finanzas personales o de negocios).

**Aplicaciones web:** llamadas “webapps”, esta categoría de software centrado en redes agrupa una amplia gama de aplicaciones. En su forma más sencilla, las *webapps* son poco más que un conjunto de archivos de hipertexto vinculados que presentan información con uso de texto y gráficas limitadas. Sin embargo, desde que surgió Web 2.0, las *webapps* están evolucionando hacia ambientes de cómputo sofisticados que no sólo proveen características aisladas, funciones de cómputo y contenido para el usuario final, sino que también están integradas con bases de datos corporativas y aplicaciones de negocios.

**Software de inteligencia artificial:** hace uso de algoritmos no numéricos para resolver problemas complejos que no son fáciles de tratar computacionalmente o con el análisis directo. Las aplicaciones en esta área incluyen robótica, sistemas expertos, reconocimiento de patrones (imagen y voz), redes neurales artificiales, demostración de teoremas y juegos.

Son millones de ingenieros de software en todo el mundo los que trabajan duro en proyectos de software en una o más de estas categorías. En ciertos casos se elaboran sistemas nuevos, pero en muchos otros se corrigen, adaptan y mejoran aplicaciones ya existentes. No es raro que una joven ingeniera de software trabaje en un programa de mayor edad que la de ella... Las generaciones pasadas de los trabajadores del software dejaron un legado en cada una de las categorías mencionadas. Por fortuna, la herencia que dejará la actual generación aligerará la carga de los futuros ingenieros de software. Aun así, nuevos desafíos (capítulo 31) han aparecido en el horizonte.

**Computación en un mundo abierto:** el rápido crecimiento de las redes inalámbricas quizá lleve pronto a la computación verdaderamente ubicua y distribuida. El reto para los ingenieros de software será desarrollar software de sistemas y aplicación que permita a dispositivos móviles, computadoras personales y sistemas empresariales comunicarse a través de redes enormes.

**Construcción de redes:** la red mundial (World Wide Web) se está convirtiendo con rapidez tanto en un motor de computación como en un proveedor de contenido. El desafío para los ingenieros de software es hacer arquitecturas sencillas (por ejemplo, planeación financiera personal y aplicaciones sofisticadas que proporcionen un beneficio a mercados objetivo de usuarios finales en todo el mundo).

**Fuente abierta:** tendencia creciente que da como resultado la distribución de código fuente para aplicaciones de sistemas (por ejemplo, sistemas operativos, bases de datos y ambientes de desarrollo) de modo que mucha gente pueda contribuir a su desarrollo. El desafío para los ingenieros de software es elaborar código fuente que sea autodescriptivo, y también, lo que es más importante, desarrollar técnicas que permitirán tanto a los consumidores como a los desarrolladores saber cuáles son los cambios hechos y cómo se manifiestan dentro del software.

Es indudable que cada uno de estos nuevos retos obedecerá a la ley de las consecuencias imprevistas y tendrá efectos (para hombres de negocios, ingenieros de software y usuarios finales) que hoy no pueden predecirse. Sin embargo, los ingenieros de software pueden prepararse de-

**Cita:**

“No hay computadora que tenga sentido común.”

Marvin Minsky

**Cita:**

“No siempre puedes predecir, pero siempre puedes prepararte.”

Anónimo

sarrollando un proceso que sea ágil y suficientemente adaptable para que acepte los cambios profundos en la tecnología y las reglas de los negocios que seguramente surgirán en la década siguiente.

### 1.1.3 Software heredado

Cientos de miles de programas de cómputo caen en uno de los siete dominios amplios de aplicación que se estudiaron en la subsección anterior. Algunos de ellos son software muy nuevo, disponible para ciertos individuos, industria y gobierno. Pero otros programas son más viejos, en ciertos casos *muy* viejos.

Estos programas antiguos —que es frecuente denominar *software heredado*— han sido centro de atención y preocupación continuas desde la década de 1960. Dayani-Fard y sus colegas [Day99] describen el software heredado de la manera siguiente:

Los sistemas de software heredado [...] fueron desarrollados hace varias décadas y han sido modificados de manera continua para que satisfagan los cambios en los requerimientos de los negocios y plataformas de computación. La proliferación de tales sistemas es causa de dolores de cabeza para las organizaciones grandes, a las que resulta costoso mantenerlos y riesgoso hacerlos evolucionar.

Liu y sus colegas [Liu98] amplían esta descripción al hacer notar que “muchos sistemas heredados continúan siendo un apoyo para las funciones básicas del negocio y son ‘indispensables’ para éste”. Además, el software heredado se caracteriza por su longevidad e importancia crítica para el negocio.

Desafortunadamente, en ocasiones hay otra característica presente en el software heredado: *mala calidad*.<sup>5</sup> Hay veces en las que los sistemas heredados tienen diseños que no son susceptibles de extenderse, código confuso, documentación mala o inexistente, casos y resultados de pruebas que nunca se archivaron, una historia de los cambios mal administrada... la lista es muy larga. A pesar de esto, dichos sistemas dan apoyo a las “funciones básicas del negocio y son indispensables para éste”. ¿Qué hacer?

La única respuesta razonable es: *hacer nada*, al menos hasta que el sistema heredado tenga un cambio significativo. Si el software heredado satisface las necesidades de sus usuarios y corre de manera confiable, entonces no falla ni necesita repararse. Sin embargo, conforme pase el tiempo será frecuente que los sistemas de software evolucionen por una o varias de las siguientes razones:

- El software debe adaptarse para que cumpla las necesidades de los nuevos ambientes del cómputo y de la tecnología.
- El software debe ser mejorado para implementar nuevos requerimientos del negocio.
- El software debe ampliarse para que sea operable con otros sistemas o bases de datos modernos.
- La arquitectura del software debe rediseñarse para hacerla viable dentro de un ambiente de redes.

Cuando ocurren estos modos de evolución, debe hacerse la reingeniería del sistema heredado (capítulo 29) para que sea viable en el futuro. La meta de la ingeniería de software moderna es “desarrollar metodologías que se basen en el concepto de evolución; es decir, el concepto de que los sistemas de software cambian continuamente, que los nuevos sistemas de software se



**¿Qué hago si encuentro un sistema heredado de mala calidad?**



**¿Qué tipos de cambios se hacen a los sistemas heredados?**



*Todo ingeniero de software debe reconocer que el cambio es natural. No trate de evitarlo.*

<sup>5</sup> En este caso, la calidad se juzga con base en el pensamiento moderno de la ingeniería de software, criterio algo injusto, ya que algunos conceptos y principios de la ingeniería de software moderna tal vez no hayan sido bien entendidos en la época en que se desarrolló el software heredado.

desarrollan a partir de los antiguos y [...] que todo debe operar entre sí y cooperar con cada uno de los demás” [Day99].

## 1.2 LA NATURALEZA ÚNICA DE LAS WEBAPPS

### Cita:

“Cuando veamos cualquier tipo de estabilización, la web se habrá convertido en algo completamente diferente.”

Louis Monier

En los primeros días de la Red Mundial (entre 1990 y 1995), los *sitios web* consistían en poco más que un conjunto de archivos de hipertexto vinculados que presentaban la información con el empleo de texto y gráficas limitadas. Al pasar el tiempo, el aumento de HTML por medio de herramientas de desarrollo (XML, Java) permitió a los ingenieros de la web brindar capacidad de cómputo junto con contenido de información. Habían nacido los *sistemas y aplicaciones basados en la web*<sup>6</sup> (denominó a éstas en forma colectiva como *webapps*). En la actualidad, las *webapps* se han convertido en herramientas sofisticadas de cómputo que no sólo proporcionan funciones aisladas al usuario final, sino que también se han integrado con bases de datos corporativas y aplicaciones de negocios.

Como se dijo en la sección 1.1.2, las *webapps* son una de varias categorías distintas de software. No obstante, podría argumentarse que las *webapps* son diferentes. Powell [Pow98] sugiere que los sistemas y aplicaciones basados en web “involucran una mezcla entre las publicaciones impresas y el desarrollo de software, entre la mercadotecnia y la computación, entre las comunicaciones internas y las relaciones exteriores, y entre el arte y la tecnología”. La gran mayoría de *webapps* presenta los siguientes atributos:

? ¿Qué característica diferencia las *webapps* de otro software?

**Uso intensivo de redes.** Una *webapp* reside en una red y debe atender las necesidades de una comunidad diversa de clientes. La red permite acceso y comunicación mundiales (por ejemplo, internet) o tiene acceso y comunicación limitados (por ejemplo, una intranet corporativa).

**Concurrencia.** A la *webapp* puede acceder un gran número de usuarios a la vez. En muchos casos, los patrones de uso entre los usuarios finales varían mucho.

**Carga impredecible.** El número de usuarios de la *webapp* cambia en varios órdenes de magnitud de un día a otro. El lunes tal vez la utilicen cien personas, el jueves quizá 10 000 usen el sistema.

**Rendimiento.** Si un usuario de la *webapp* debe esperar demasiado (para entrar, para el procesamiento por parte del servidor, para el formado y despliegue del lado del cliente), él o ella quizá decidan irse a otra parte.

**Disponibilidad.** Aunque no es razonable esperar una disponibilidad de 100%, es frecuente que los usuarios de *webapps* populares demanden acceso las 24 horas de los 365 días del año. Los usuarios en Australia o Asia quizá demanden acceso en horas en las que las aplicaciones internas de software tradicionales en Norteamérica no estén en línea por razones de mantenimiento.

**Orientadas a los datos.** La función principal de muchas *webapp* es el uso de hipermedios para presentar al usuario final contenido en forma de texto, gráficas, audio y video. Además, las *webapps* se utilizan en forma común para acceder a información que existe en bases de datos que no son parte integral del ambiente basado en web (por ejemplo, comercio electrónico o aplicaciones financieras).

<sup>6</sup> En el contexto de este libro, el término *aplicación web* (*webapp*) agrupa todo, desde una simple página web que ayude al consumidor a calcular el pago del arrendamiento de un automóvil hasta un sitio web integral que proporcione servicios completos de viaje para gente de negocios y vacacionistas. En esta categoría se incluyen sitios web completos, funcionalidad especializada dentro de sitios web y aplicaciones de procesamiento de información que residen en internet o en una intranet o extranet.

**Contenido sensible.** La calidad y naturaleza estética del contenido constituye un rasgo importante de la calidad de una *webapp*.

**Evolución continua.** A diferencia del software de aplicación convencional que evoluciona a lo largo de una serie de etapas planeadas y separadas cronológicamente, las aplicaciones web evolucionan en forma continua. No es raro que ciertas *webapp* (específicamente su contenido) se actualicen minuto a minuto o que su contenido se calcule en cada solicitud.

**Inmediatez.** Aunque la *inmediatez* —necesidad apremiante de que el software llegue con rapidez al mercado— es una característica en muchos dominios de aplicación, es frecuente que las *webapps* tengan plazos de algunos días o semanas para llegar al mercado.<sup>7</sup>

**Seguridad.** Debido a que las *webapps* se encuentran disponibles con el acceso a una red, es difícil o imposible limitar la población de usuarios finales que pueden acceder a la aplicación. Con el fin de proteger el contenido sensible y brindar modos seguros de transmisión de los datos, deben implementarse medidas estrictas de seguridad a través de la infraestructura de apoyo de una *webapp* y dentro de la aplicación misma.

**Estética.** Parte innegable del atractivo de una *webapp* es su apariencia y percepción. Cuando se ha diseñado una aplicación para comercializar o vender productos o ideas, la estética tiene tanto que ver con el éxito como el diseño técnico.

Podría argumentarse que otras categorías de aplicaciones estudiadas en la sección 1.1.2 muestran algunos de los atributos mencionados. Sin embargo, las *webapps* casi siempre poseen todos ellos.

### 1.3 INGENIERÍA DE SOFTWARE

Con objeto de elaborar software listo para enfrentar los retos del siglo XXI, el lector debe aceptar algunas realidades sencillas:

#### PUNTO CLAVE

Entender el problema antes de dar una solución.

- El software se ha incrustado profundamente en casi todos los aspectos de nuestras vidas y, como consecuencia, el número de personas que tienen interés en las características y funciones que brinda una aplicación específica<sup>8</sup> ha crecido en forma notable. Cuando ha de construirse una aplicación nueva o sistema incrustado, deben escucharse muchas opiniones. Y en ocasiones parece que cada una de ellas tiene una idea un poco distinta de cuáles características y funciones debiera tener el software. *Se concluye que debe hacerse un esfuerzo concertado para entender el problema antes de desarrollar una aplicación de software.*
- Los requerimientos de la tecnología de la información que demandan los individuos, negocios y gobiernos se hacen más complejos con cada año que pasa. En la actualidad, grandes equipos de personas crean programas de cómputo que antes eran elaborados por un solo individuo. El software sofisticado, que alguna vez se implementó en un ambiente de cómputo predecible y autocontenido, hoy en día se halla incrustado en el interior de todo, desde la electrónica de consumo hasta dispositivos médicos o sistemas de armamento. La complejidad de estos nuevos sistemas y productos basados en computadora demanda atención cuidadosa a las interacciones de todos los elementos del sistema. *Se concluye que el diseño se ha vuelto una actividad crucial.*

#### PUNTO CLAVE

El diseño es una actividad crucial de la ingeniería de software.

<sup>7</sup> Con las herramientas modernas es posible producir páginas web sofisticadas en unas cuantas horas.

<sup>8</sup> En una parte posterior de este libro, llamaré a estas personas “participantes”.

### PUNTO CLAVE

Tanto la calidad como la facilidad de recibir mantenimiento son resultado de un buen diseño.

- Los individuos, negocios y gobiernos dependen cada vez más del software para tomar decisiones estratégicas y tácticas, así como para sus operaciones y control cotidianos. Si el software falla, las personas y empresas grandes pueden experimentar desde un inconveniente menor hasta fallas catastróficas. *Se concluye que el software debe tener alta calidad.*
- A medida que aumenta el valor percibido de una aplicación específica se incrementa la probabilidad de que su base de usuarios y longevidad también crezcan. Conforme se extiende su base de usuarios y el tiempo de uso, las demandas para adaptarla y mejorarla también crecerán. *Se concluye que el software debe tener facilidad para recibir mantenimiento.*

Estas realidades simples llevan a una conclusión: *debe hacerse ingeniería con el software en todas sus formas y a través de todos sus dominios de aplicación.* Y esto conduce al tema de este libro: *la ingeniería de software.*

Aunque cientos de autores han desarrollado definiciones personales de la ingeniería de software, la propuesta por Fritz Bauer [Nau69] en la conferencia fundamental sobre el tema todavía sirve como base para el análisis:

[La ingeniería de software es] el establecimiento y uso de principios fundamentales de la ingeniería con objeto de desarrollar en forma económica software que sea confiable y que trabaje con eficiencia en máquinas reales.

El lector se sentirá tentado de ampliar esta definición.<sup>9</sup> Dice poco sobre los aspectos técnicos de la calidad del software; no habla directamente de la necesidad de satisfacer a los consumidores ni de entregar el producto a tiempo; omite mencionar la importancia de la medición y la metrología; no establece la importancia de un proceso eficaz. No obstante, la definición de Bauer proporciona una base. ¿Cuáles son los “principios fundamentales de la ingeniería” que pueden aplicarse al desarrollo del software de computadora? ¿Cómo se desarrolla software “en forma económica” y que sea “confiable”? ¿Qué se requiere para crear programas de cómputo que trabajen con “eficiencia”, no en una sino en muchas “máquinas reales” diferentes? Éstas son las preguntas que siguen siendo un reto para los ingenieros de software.

El IEEE [IEEE93a] ha desarrollado una definición más completa, como sigue:

La ingeniería de software es: 1) La aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software; es decir, la aplicación de la ingeniería al software. 2) El estudio de enfoques según el punto 1.

Aun así, el enfoque “sistemático, disciplinado y cuantificable” aplicado por un equipo de software podría ser algo burdo para otro. Se necesita disciplina, pero también adaptabilidad y agilidad.

La ingeniería de software es una tecnología con varias capas. Como se aprecia en la figura 1.3, cualquier enfoque de ingeniería (incluso la de software) debe basarse en un compromiso organizacional con la calidad. La administración total de la calidad, Six Sigma y otras filosofías similares<sup>10</sup> alimentan la cultura de mejora continua, y es esta cultura la que lleva en última instancia al desarrollo de enfoques cada vez más eficaces de la ingeniería de software. El fundamento en el que se apoya la ingeniería de software es el compromiso con la calidad.

El fundamento para la ingeniería de software es la capa *proceso*. El proceso de ingeniería de software es el aglutinante que une las capas de la tecnología y permite el desarrollo racional y

### Cita:

“Más que una disciplina o cuerpo de conocimientos, ingeniería es un verbo, una palabra de acción, una forma de abordar un problema.”

Scott Whitmir

### ? ¿Cómo se define la ingeniería de software?

### PUNTO CLAVE

La ingeniería de software incluye un proceso, métodos y herramientas para administrar y hacer ingeniería con el software.

<sup>9</sup> Consulte muchas otras definiciones en [www.answers.com/topic/software-engineering#wp\\_note-13](http://www.answers.com/topic/software-engineering#wp_note-13).

<sup>10</sup> En el capítulo 14 y toda la parte 3 del libro se estudia la administración de la calidad y los enfoques relacionados con ésta.

**FIGURA 1.3**

Capas de la ingeniería de software

**WebRef**

*CrossTalk* es un periódico que da información práctica sobre procesos, métodos y herramientas. Se encuentra en [www.stsc.hill.af.mil](http://www.stsc.hill.af.mil)

oportuno del software de cómputo. El proceso define una estructura que debe establecerse para la obtención eficaz de tecnología de ingeniería de software. El proceso de software forma la base para el control de la administración de proyectos de software, y establece el contexto en el que se aplican métodos técnicos, se generan productos del trabajo (modelos, documentos, datos, reportes, formatos, etc.), se establecen puntos de referencia, se asegura la calidad y se administra el cambio de manera apropiada.

Los *métodos* de la ingeniería de software proporcionan la experiencia técnica para elaborar software. Incluyen un conjunto amplio de tareas, como comunicación, análisis de los requerimientos, modelación del diseño, construcción del programa, pruebas y apoyo. Los métodos de la ingeniería de software se basan en un conjunto de principios fundamentales que gobiernan cada área de la tecnología e incluyen actividades de modelación y otras técnicas descriptivas.

Las *herramientas* de la ingeniería de software proporcionan un apoyo automatizado o semiautomatizado para el proceso y los métodos. Cuando se integran las herramientas de modo que la información creada por una pueda ser utilizada por otra, queda establecido un sistema llamado *ingeniería de software asistido por computadora* que apoya el desarrollo de software.

## 1.4 EL PROCESO DEL SOFTWARE

**?** ¿Cuáles son los elementos de un proceso de software?

**Cita:**

"Un proceso define quién hace qué, cuándo y cómo, para alcanzar cierto objetivo."

Ivar Jacobson, Grady Booch y James Rumbaugh

Un *proceso* es un conjunto de actividades, acciones y tareas que se ejecutan cuando va a crearse algún producto del trabajo. Una *actividad* busca lograr un objetivo amplio (por ejemplo, comunicación con los participantes) y se desarrolla sin importar el dominio de la aplicación, tamaño del proyecto, complejidad del esfuerzo o grado de rigor con el que se usará la ingeniería de software. Una *acción* (diseño de la arquitectura) es un conjunto de tareas que producen un producto importante del trabajo (por ejemplo, un modelo del diseño de la arquitectura). Una *tarea* se centra en un objetivo pequeño pero bien definido (por ejemplo, realizar una prueba unitaria) que produce un resultado tangible.

En el contexto de la ingeniería de software, un proceso *no* es una prescripción rígida de cómo elaborar software de cómputo. Por el contrario, es un enfoque adaptable que permite que las personas que hacen el trabajo (el equipo de software) busquen y elijan el conjunto apropiado de acciones y tareas para el trabajo. Se busca siempre entregar el software en forma oportuna y con calidad suficiente para satisfacer a quienes patrocinaron su creación y a aquellos que lo usarán.

La *estructura del proceso* establece el fundamento para el proceso completo de la ingeniería de software por medio de la identificación de un número pequeño de *actividades estructurales* que sean aplicables a todos los proyectos de software, sin importar su tamaño o complejidad. Además, la estructura del proceso incluye un conjunto de *actividades sombrilla* que son aplicables a través de todo el proceso del software. Una estructura de proceso general para la ingeniería de software consta de cinco actividades:

**?** ¿Cuáles son las cinco actividades estructurales del proceso?

**Cita:**

"Einstein afirmaba que debía haber una explicación sencilla de la naturaleza porque Dios no es caprichoso o arbitrario. Al ingeniero de software no lo conforta una fe parecida. Gran parte de la complejidad que debe doblegar es de origen arbitrario."

Fred Brooks

**Comunicación.** Antes de que comience cualquier trabajo técnico, tiene importancia crítica comunicarse y colaborar con el cliente (y con otros participantes).<sup>11</sup> Se busca entender los objetivos de los participantes respecto del proyecto, y reunir los requerimientos que ayuden a definir las características y funciones del software.

**Planeación.** Cualquier viaje complicado se simplifica si existe un mapa. Un proyecto de software es un viaje difícil, y la actividad de planeación crea un "mapa" que guía al equipo mientras viaja. El mapa —llamado *plan del proyecto de software*— define el trabajo de ingeniería de software al describir las tareas técnicas por realizar, los riesgos probables, los recursos que se requieren, los productos del trabajo que se obtendrán y una programación de las actividades.

**Modelado.** Ya sea usted diseñador de paisaje, constructor de puentes, ingeniero aeronáutico, carpintero o arquitecto, a diario trabaja con modelos. Crea un "bosquejo" del objeto por hacer a fin de entender el panorama general —cómo se verá arquitectónicamente, cómo ajustan entre sí las partes constituyentes y muchas características más—. Si se requiere, refina el bosquejo con más y más detalles en un esfuerzo por comprender mejor el problema y cómo resolverlo. Un ingeniero de software hace lo mismo al crear modelos a fin de entender mejor los requerimientos del software y el diseño que los satisfará.

**Construcción.** Esta actividad combina la generación de código (ya sea manual o automatizada) y las pruebas que se requieren para descubrir errores en éste.

**Despliegue.** El software (como entidad completa o como un incremento parcialmente terminado) se entrega al consumidor que lo evalúa y que le da retroalimentación, misma que se basa en dicha evaluación.

Estas cinco actividades estructurales genéricas se usan durante el desarrollo de programas pequeños y sencillos, en la creación de aplicaciones web grandes y en la ingeniería de sistemas enormes y complejos basados en computadoras. Los detalles del proceso de software serán distintos en cada caso, pero las actividades estructurales son las mismas.

Para muchos proyectos de software, las actividades estructurales se aplican en forma iterativa a medida que avanza el proyecto. Es decir, la **comunicación**, la **planeación**, el **modelado**, la **construcción** y el **despliegue** se ejecutan a través de cierto número de repeticiones del proyecto. Cada iteración produce un *incremento del software* que da a los participantes un subconjunto de características y funcionalidad generales del software. Conforme se produce cada incremento, el software se hace más y más completo.

Las actividades estructurales del proceso de ingeniería de software son complementadas por cierto número de *actividades sombrilla*. En general, las actividades sombrilla se aplican a lo largo de un proyecto de software y ayudan al equipo que lo lleva a cabo a administrar y controlar el avance, la calidad, el cambio y el riesgo. Es común que las actividades sombrilla sean las siguientes:

**Seguimiento y control del proyecto de software:** permite que el equipo de software evalúe el progreso comparándolo con el plan del proyecto y tome cualquier acción necesaria para apegarse a la programación de actividades.

**Administración del riesgo:** evalúa los riesgos que puedan afectar el resultado del proyecto o la calidad del producto.

**PUNTO CLAVE**

Las actividades sombrilla ocurren a lo largo del proceso de software y se centran sobre todo en la administración, el seguimiento y el control del proyecto.

<sup>11</sup> Un *participante* es cualquier persona que tenga algo que ver en el resultado exitoso del proyecto —gerentes del negocio, usuarios finales, ingenieros de software, personal de apoyo, etc.—. Rob Thomset dice en broma que "un participante es una persona que blande una estaca grande y aguda [...] Si no vez más lejos que los participantes, ya sabes dónde terminará la estaca". (N. del T.: Esta nota es un juego de palabras: *stake* significa estaca y también *parte*, y *stakeholder* es el que blande una estaca, pero también un *participante*.)

**Aseguramiento de la calidad del software:** define y ejecuta las actividades requeridas para garantizar la calidad del software.

**Revisiones técnicas:** evalúa los productos del trabajo de la ingeniería de software a fin de descubrir y eliminar errores antes de que se propaguen a la siguiente actividad.

**Medición:** define y reúne mediciones del proceso, proyecto y producto para ayudar al equipo a entregar el software que satisfaga las necesidades de los participantes; puede usarse junto con todas las demás actividades estructurales y sombrilla.

**Administración de la configuración del software:** administra los efectos del cambio a lo largo del proceso del software.

**Administración de la reutilización:** define criterios para volver a usar el producto del trabajo (incluso los componentes del software) y establece mecanismos para obtener componentes reutilizables.

**Preparación y producción del producto del trabajo:** agrupa las actividades requeridas para crear productos del trabajo, tales como modelos, documentos, registros, formatos y listas.

Cada una de estas actividades sombrilla se analiza en detalle más adelante.

Ya se dijo en esta sección que el proceso de ingeniería de software no es una prescripción rígida que deba seguir en forma dogmática el equipo que lo crea. Al contrario, debe ser ágil y adaptable (al problema, al proyecto, al equipo y a la cultura organizacional). Por tanto, un proceso adoptado para un proyecto puede ser significativamente distinto de otro adoptado para otro proyecto. Entre las diferencias se encuentran las siguientes:

- Flujo general de las actividades, acciones y tareas, así como de las interdependencias entre ellas
- Grado en el que las acciones y tareas están definidas dentro de cada actividad estructural
- Grado en el que se identifican y requieren los productos del trabajo
- Forma en la que se aplican las actividades de aseguramiento de la calidad
- Manera en la que se realizan las actividades de seguimiento y control del proyecto
- Grado general de detalle y rigor con el que se describe el proceso
- Grado con el que el cliente y otros participantes se involucran con el proyecto
- Nivel de autonomía que se da al equipo de software
- Grado con el que son prescritos la organización y los roles del equipo

En la parte 1 de este libro, se examinará el proceso de software con mucho detalle. Los *modelos de proceso prescriptivo* (capítulo 2) enfatizan la definición, la identificación y la aplicación detalladas de las actividades y tareas del proceso. Su objetivo es mejorar la calidad del sistema, desarrollar proyectos más manejables, hacer más predecibles las fechas de entrega y los costos, y guiar a los equipos de ingenieros de software cuando realizan el trabajo que se requiere para construir un sistema. Desafortunadamente, ha habido casos en los que estos objetivos no se han logrado. Si los modelos prescriptivos se aplican en forma dogmática y sin adaptación, pueden incrementar el nivel de burocracia asociada con el desarrollo de sistemas basados en computadora y crear inadvertidamente dificultades para todos los participantes.

Los *modelos de proceso ágil* (capítulo 3) ponen el énfasis en la “agilidad” del proyecto y siguen un conjunto de principios que conducen a un enfoque más informal (pero no menos efectivo, dicen sus defensores) del proceso de software. Por lo general, se dice que estos modelos del proceso son “ágiles” porque acentúan la maniobrabilidad y la adaptabilidad. Son apropiados para muchos tipos de proyectos y son útiles en particular cuando se hace ingeniería sobre aplicaciones web.

### PUNTO CLAVE

La adaptación del proceso de software es esencial para el éxito del proyecto.

### ? ¿Qué diferencias existen entre los modelos del proceso?

#### Cita:

“Siento que una receta es sólo un tema que una cocinera inteligente ejecuta con una variación en cada ocasión.”

Madame Benoit

### ? ¿Qué caracteriza a un proceso “ágil”?

## 1.5 LA PRÁCTICA DE LA INGENIERÍA DE SOFTWARE

### WebRef

En la dirección [www.literateprogramming.com](http://www.literateprogramming.com) se encuentran varias citas provocativas sobre la práctica de la ingeniería de software.



Podría decirse que el enfoque de Polya es simple sentido común. Es verdad. Pero es sorprendente la frecuencia con la que el sentido común es poco común en el mundo del software.

En la sección 1.4 se introdujo un modelo general de proceso de software compuesto de un conjunto de actividades que establecen una estructura para la práctica de la ingeniería de software. Las actividades estructurales generales —**comunicación, planeación, modelado, construcción y despliegue**— y las actividades sombrilla establecen el esqueleto de la arquitectura para el trabajo de ingeniería de software. Pero, ¿cómo entra aquí la práctica de la ingeniería de software? En las secciones que siguen, el lector obtendrá la comprensión básica de los conceptos y principios generales que se aplican a las actividades estructurales.<sup>12</sup>

### 1.5.1 La esencia de la práctica

En un libro clásico, *How to Solve It*, escrito antes de que existieran las computadoras modernas, George Polya [Pol45] describió la esencia de la solución de problemas y, en consecuencia, la esencia de la práctica de la ingeniería de software:

1. *Entender el problema* (comunicación y análisis).
2. *Planear la solución* (modelado y diseño del software).
3. *Ejecutar el plan* (generación del código).
4. *Examinar la exactitud del resultado* (probar y asegurar la calidad).

En el contexto de la ingeniería de software, estas etapas de sentido común conducen a una serie de preguntas esenciales [adaptado de Pol45]:

**Entender el problema.** En ocasiones es difícil de admitir, pero la mayor parte de nosotros adoptamos una actitud de orgullo desmedido cuando se nos presenta un problema. Escuchamos por unos segundos y después pensamos: *Claro, sí, entiendo, resolvamos esto*. Desafortunadamente, entender no siempre es fácil. Es conveniente dedicar un poco de tiempo a responder algunas preguntas sencillas:

- *¿Quiénes tienen que ver con la solución del problema?* Es decir, ¿quiénes son los participantes?
- *¿Cuáles son las incógnitas?* ¿Cuáles datos, funciones y características se requieren para resolver el problema en forma apropiada?
- *¿Puede fraccionarse el problema?* ¿Es posible representarlo con problemas más pequeños que sean más fáciles de entender?
- *¿Es posible representar gráficamente el problema?* ¿Puede crearse un modelo de análisis?

**Planear la solución.** Ahora entiende el problema (o es lo que piensa) y no puede esperar para escribir el código. Antes de hacerlo, cálmese un poco y haga un pequeño diseño:

- *¿Ha visto antes problemas similares?* ¿Hay patrones reconocibles en una solución potencial? ¿Hay algún software existente que implemente los datos, funciones y características que se requieren?
- *¿Ha resuelto un problema similar?* Si es así, ¿son reutilizables los elementos de la solución?
- *¿Pueden definirse problemas más pequeños?* Si así fuera, ¿hay soluciones evidentes para éstos?

### Cita:

“En la solución de cualquier problema hay un grano de descubrimiento.”

George Polya

<sup>12</sup> El lector debería volver a consultar las secciones de este capítulo a medida que en el libro se describan en específico los métodos y las actividades sombrilla de la ingeniería de software.

- ¿Es capaz de representar una solución en una forma que lleve a su implementación eficaz?  
¿Es posible crear un modelo del diseño?

**Ejecutar el plan.** El diseño que creó sirve como un mapa de carreteras para el sistema que quiere construir. Puede haber desviaciones inesperadas y es posible que descubra un camino mejor a medida que avanza, pero el “plan” le permitirá proceder sin que se pierda.

- ¿Se ajusta la solución al plan? ¿El código fuente puede apegarse al modelo del diseño?
- ¿Es probable que cada parte componente de la solución sea correcta? ¿El diseño y código se han revisado o, mejor aún, se han hecho pruebas respecto de la corrección del algoritmo?

**Examinar el resultado.** No se puede estar seguro de que la solución sea perfecta, pero sí de que se ha diseñado un número suficiente de pruebas para descubrir tantos errores como sea posible.

- ¿Puede probarse cada parte componente de la solución? ¿Se ha implementado una estrategia razonable para hacer pruebas?
- ¿La solución produce resultados que se apegan a los datos, funciones y características que se requieren? ¿El software se ha validado contra todos los requerimientos de los participantes?

No debiera sorprender que gran parte de este enfoque tenga que ver con el sentido común. En realidad, es razonable afirmar que un enfoque de sentido común para la ingeniería de software hará que nunca se extravié.

### 1.5.2 Principios generales

El diccionario define la palabra *principio* como “una ley importante o suposición que subyace y se requiere en un sistema de pensamiento”. En este libro se analizarán principios en muchos niveles distintos de abstracción. Algunos se centran en la ingeniería de software como un todo, otros consideran una actividad estructural general específica (por ejemplo, **comunicación**), y otros más se centran en acciones de la ingeniería de software (por ejemplo, diseño de la arquitectura) o en tareas técnicas (escribir un escenario para el uso). Sin importar su nivel de enfoque, los principios lo ayudarán a establecer un conjunto de herramientas mentales para una práctica sólida de la ingeniería de software. Ésa es la razón de que sean importantes.

David Hooker [Hoo96] propuso siete principios que se centran en la práctica de la ingeniería de software como un todo. Se reproducen en los párrafos siguientes:<sup>13</sup>

#### Primer principio: *La razón de que exista todo*

Un sistema de software existe por una razón: *dar valor a sus usuarios*. Todas las decisiones deben tomarse teniendo esto en mente. Antes de especificar un requerimiento del sistema, antes de notar la funcionalidad de una parte de él, antes de determinar las plataformas del hardware o desarrollar procesos, plantéese preguntas tales como: “¿Esto agrega valor real al sistema?” Si la respuesta es “no”, entonces no lo haga. Todos los demás principios apoyan a éste.

#### Segundo principio: *MSE (Mantenlo sencillo, estúpido...)*

El diseño de software no es un proceso caprichoso. Hay muchos factores por considerar en cualquier actividad de diseño. *Todo diseño debe ser tan simple como sea posible, pero no más.*



*Antes de comenzar un proyecto de software, asegúrese de que el software tenga un propósito para el negocio y que los usuarios perciben valor en él.*

<sup>13</sup> Reproducido con permiso del autor [Hoo96]. Hooker define algunos patrones para estos principios en <http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment>.

**Cita:**

"Hay cierta majestad en la sencillez, que es con mucho todo lo que adorna al ingenio."

Papa Alejandro  
(1688-1744)

**PUNTO  
CLAVE**

Si el software tiene valor, cambiará durante su vida útil. Por esa razón, debe construirse de forma que sea fácil darle mantenimiento.

Esto facilita conseguir un sistema que sea comprendido más fácilmente y que sea susceptible de recibir mantenimiento, lo que no quiere decir que en nombre de la simplicidad deban descartarse características o hasta rasgos internos. En realidad, los diseños más elegantes por lo general son los más simples. Simple tampoco significa "rápido y sucio". La verdad es que con frecuencia se requiere mucha reflexión y trabajo con iteraciones múltiples para poder simplificar. La recompensa es un software más fácil de mantener y menos propenso al error.

### **Tercer principio: Mantener la visión**

Una *visión clara es esencial para el éxito de un proyecto de software*. Sin ella, casi infaliblemente el proyecto terminará siendo un ser "con dos [o más mentes]". Sin integridad conceptual, un sistema está amenazado de convertirse en una urdimbre de diseños incompatibles unidos por tornillos del tipo equivocado [...] Comprometer la visión de la arquitectura de un sistema de software debilita y, finalmente hará que colapsen incluso los sistemas bien diseñados. Tener un arquitecto que pueda para mantener la visión y que obligue a su cumplimiento garantiza un proyecto de software muy exitoso.

### **Cuarto principio: Otros consumirán lo que usted produce**

Rara vez se construye en el vacío un sistema de software con fortaleza industrial. En un modo u otro, alguien más lo usará, mantendrá, documentará o, de alguna forma, dependerá de su capacidad para entender el sistema. Así que *siempre establezca especificaciones, diseñe e implemente con la seguridad de que alguien más tendrá que entender lo que usted haga*. La audiencia para cualquier producto de desarrollo de software es potencialmente grande. Elabore especificaciones con la mirada puesta en los usuarios. Diseñe con los implementadores en mente. Codifique pensando en aquellos que deben dar mantenimiento y ampliar el sistema. Alguien debe depurar el código que usted escriba, y eso lo hace usuario de su código. Hacer su trabajo más fácil agrega valor al sistema.

### **Quinto principio: Ábrase al futuro**

Un sistema con larga vida útil tiene más valor. En los ambientes de cómputo actuales, donde las especificaciones cambian de un momento a otro y las plataformas de hardware quedan obsoletas con sólo unos meses de edad, es común que la vida útil del software se mida en meses y no en años. Sin embargo, los sistemas de software con verdadera "fortaleza industrial" deben durar mucho más tiempo. Para tener éxito en esto, los sistemas deben ser fáciles de adaptar a éstos y otros cambios. Los sistemas que lo logran son los que se diseñaron para ello desde el principio. *Nunca diseñe sobre algo iniciado*. Siempre pregunte: "¿qué pasa si...?" y prepárese para todas las respuestas posibles mediante la creación de sistemas que resuelvan el problema general, no sólo uno específico.<sup>14</sup> Es muy posible que esto lleve a volver a usar un sistema completo.

### **Sexto principio: Planee por anticipado la reutilización**

La reutilización ahorra tiempo y esfuerzo.<sup>15</sup> Al desarrollar un sistema de software, lograr un alto nivel de reutilización es quizá la meta más difícil de lograr. La reutilización del código y de los diseños se ha reconocido como uno de los mayores beneficios de usar tecnologías orientadas a objetos. Sin embargo, la recuperación de esta inversión no es automática. Para reforzar las posibilidades de la reutilización que da la programación orientada a objetos [o la

<sup>14</sup> Es peligroso llevar este consejo a los extremos. Diseñar para resolver "el problema general" en ocasiones requiere compromisos de rendimiento y puede volver ineficientes las soluciones específicas.

<sup>15</sup> Aunque esto es verdad para aquellos que reutilizan software en proyectos futuros, volver a usar puede ser caro para quienes deben diseñar y elaborar componentes reutilizables. Los estudios indican que diseñar y construir componentes reutilizables llega a costar entre 25 y 200% más que el software buscado. En ciertos casos no se justifica la diferencia de costos.

convencional], se requiere reflexión y planeación. Hay muchas técnicas para incluir la reutilización en cada nivel del proceso de desarrollo del sistema... *La planeación anticipada en busca de la reutilización disminuye el costo e incrementa el valor tanto de los componentes reutilizables como de los sistemas en los que se incorpora.*

#### Séptimo principio: ¡Piense!

Este último principio es tal vez el que más se pasa por alto. *Pensar en todo con claridad antes de emprender la acción casi siempre produce mejores resultados.* Cuando se piensa en algo es más probable que se haga bien. Asimismo, también se gana conocimiento al pensar cómo volver a hacerlo bien. Si usted piensa en algo y aun así lo hace mal, eso se convierte en una experiencia valiosa. Un efecto colateral de pensar es aprender a reconocer cuando no se sabe algo, punto en el que se puede investigar la respuesta. Cuando en un sistema se han puesto pensamientos claros, el valor se manifiesta. La aplicación de los primeros seis principios requiere pensar con intensidad, por lo que las recompensas potenciales son enormes.

Si todo ingeniero y equipo de software tan sólo siguiera los siete principios de Hooker, se eliminarían muchas de las dificultades que se experimentan al construir sistemas complejos basados en computadora.

## 1.6 MITOS DEL SOFTWARE

### Cita:

"En ausencia de estándares significativos, una industria nueva como la del software depende sólo del folklore."

Tom DeMarco

### WebRef

La Software Project Managers Network (Red de Gerentes de Proyectos de Software), en [www.spmn.com](http://www.spmn.com), lo ayuda a eliminar éstos y otros mitos.

Los mitos del software —creencias erróneas sobre éste y sobre el proceso que se utiliza para obtenerlo— se remontan a los primeros días de la computación. Los mitos tienen cierto número de atributos que los hacen insidiosos. Por ejemplo, parecen enunciados razonables de hechos (a veces contienen elementos de verdad), tienen una sensación intuitiva y es frecuente que los manifiesten profesionales experimentados que "conocen la historia".

En la actualidad, la mayoría de profesionales de la ingeniería de software reconocen los mitos como lo que son: actitudes equivocadas que han ocasionado serios problemas a los administradores y a los trabajadores por igual. Sin embargo, las actitudes y hábitos antiguos son difíciles de modificar, y persisten algunos remanentes de los mitos del software.

**Mitos de la administración.** Los gerentes que tienen responsabilidades en el software, como los de otras disciplinas, con frecuencia se hallan bajo presión para cumplir el presupuesto, mantener la programación de actividades sin desvíos y mejorar la calidad. Así como la persona que se ahoga se agarra de un clavo ardiente, no es raro que un gerente de software sostenga la creencia en un mito del software si eso disminuye la presión a que está sujeto (incluso de manera temporal).

**Mito:** *Tenemos un libro lleno de estándares y procedimientos para elaborar software. ¿No le dará a mi personal todo lo que necesita saber?*

**Realidad:** Tal vez exista el libro de estándares, pero ¿se utiliza? ¿Saben de su existencia los trabajadores del software? ¿Refleja la práctica moderna de la ingeniería de software? ¿Es completo? ¿Es adaptable? ¿Está dirigido a mejorar la entrega a tiempo y también se centra en la calidad? En muchos casos, la respuesta a todas estas preguntas es "no".

**Mito:** *Si nos atrasamos, podemos agregar más programadores y ponernos al corriente (en ocasiones, a esto se le llama "concepto de la horda de mongoles").*

**Realidad:** El desarrollo del software no es un proceso mecánico similar a la manufactura. En palabras de Brooks [Bro95]: "agregar personal a un proyecto de software atrasado lo atrasará más". Al principio, esta afirmación parece ir contra la intuición. Sin embargo, a medida que se agregan personas, las que ya se

encontraban trabajando deben dedicar tiempo para enseñar a los recién llegados, lo que disminuye la cantidad de tiempo dedicada al esfuerzo de desarrollo productivo. Pueden agregarse individuos, pero sólo en forma planeada y bien coordinada.

**Mito:** *Si decido subcontratar el proyecto de software a un tercero, puedo descansar y dejar que esa compañía lo elabore.*

**Realidad:** Si una organización no comprende cómo administrar y controlar proyectos de software internamente, de manera invariable tendrá dificultades cuando subcontrate proyectos de software.

**Mitos del cliente.** El cliente que requiere software de computadora puede ser la persona en el escritorio de al lado, un grupo técnico en el piso inferior, el departamento de mercadotecnia y ventas, o una compañía externa que solicita software mediante un contrato. En muchos casos, el cliente sostiene mitos sobre el software porque los gerentes y profesionales de éste hacen poco para corregir la mala información. Los mitos generan falsas expectativas (por parte del cliente) y, en última instancia, la insatisfacción con el desarrollador.



*Trabaje muy duro para entender qué es lo que tiene que hacer antes de empezar. Quizás no pueda desarrollarlo a detalle, pero entre más sepa, menor será el riesgo que tome.*

**Mito:** *Para comenzar a escribir programas, es suficiente el enunciado general de los objetivos —podremos entrar en detalles más adelante.*

**Realidad:** Aunque no siempre es posible tener el enunciado exhaustivo y estable de los requerimientos, un “planteamiento de objetivos” ambiguo es una receta para el desastre. Los requerimientos que no son ambiguos (que por lo general se obtienen en forma iterativa) se desarrollan sólo por medio de una comunicación eficaz y continua entre el cliente y el desarrollador.

**Mito:** *Los requerimientos del software cambian continuamente, pero el cambio se asimila con facilidad debido a que el software es flexible.*

**Realidad:** Es verdad que los requerimientos del software cambian, pero el efecto que los cambios tienen varía según la época en la que se introducen. Cuando se solicitan al principio cambios en los requerimientos (antes de que haya comenzado el diseño o la elaboración de código), el efecto sobre el costo es relativamente pequeño.<sup>16</sup> Sin embargo, conforme pasa el tiempo, el costo aumenta con rapidez: los recursos ya se han comprometido, se ha establecido la estructura del diseño y el cambio ocasiona perturbaciones que exigen recursos adicionales y modificaciones importantes del diseño.



*Siempre que piense que no hay tiempo para la ingeniería de software, pregúntese: “¿tendremos tiempo de hacerlo otra vez?”.*

**Mitos del profesional.** Los mitos que aún sostienen los trabajadores del software han sido alimentados por más de 50 años de cultura de programación. Durante los primeros días, la programación se veía como una forma del arte. Es difícil que mueran los hábitos y actitudes arraigados.

**Mito:** *Una vez que escribimos el programa y hacemos que funcione, nuestro trabajo ha terminado.*

**Realidad:** Alguien dijo alguna vez que “entre más pronto se comience a ‘escribir el código’, más tiempo tomará hacer que funcione”. Los datos de la industria indican que entre 60 y 80% de todo el esfuerzo dedicado al software ocurrirá después de entregarlo al cliente por primera vez.

**Mito:** *Hasta que no se haga “correr” el programa, no hay manera de evaluar su calidad.*

<sup>16</sup> Muchos ingenieros de software han adoptado un enfoque “ágil” que asimila los cambios en forma gradual y creciente, con lo que controlan su efecto y costo. Los métodos ágiles se estudian en el capítulo 3.

- Realidad:** Uno de los mecanismos más eficaces de asegurar la calidad del software puede aplicarse desde la concepción del proyecto: *la revisión técnica*. Las revisiones del software (descritas en el capítulo 15) son un “filtro de la calidad” que se ha revelado más eficaz que las pruebas para encontrar ciertas clases de defectos de software.
- Mito:** *El único producto del trabajo que se entrega en un proyecto exitoso es el programa que funciona.*
- Realidad:** Un programa que funciona sólo es una parte de una configuración de software que incluye muchos elementos. Son varios los productos terminados (modelos, documentos, planes) que proporcionan la base de la ingeniería exitosa y, lo más importante, que guían el apoyo para el software.
- Mito:** *La ingeniería de software hará que generemos documentación voluminosa e innecesaria, e invariablemente nos retrasará.*
- Realidad:** La ingeniería de software no consiste en producir documentos. Se trata de crear un producto de calidad. La mejor calidad conduce a menos repeticiones, lo que da como resultado tiempos de entrega más cortos.

Muchos profesionales del software reconocen la falacia de los mitos mencionados. Es lamentable que las actitudes y métodos habituales nutran la administración y las prácticas técnicas deficientes, aun cuando la realidad dicta un enfoque mejor. El primer paso hacia la formulación de soluciones prácticas para la ingeniería de software es el reconocimiento de las realidades en este campo.

## 1.7 CÓMO COMIENZA TODO

Todo proyecto de software se desencadena por alguna necesidad de negocios: la de corregir un defecto en una aplicación existente, la de adaptar un “sistema heredado” a un ambiente de negocios cambiante, la de ampliar las funciones y características de una aplicación ya existente o la necesidad de crear un producto, servicio o sistema nuevo.

Al comenzar un proyecto de software, es frecuente que las necesidades del negocio se expresen de manera informal como parte de una simple conversación. La plática que se presenta en el recuadro que sigue es muy común.

### CASAsegura<sup>17</sup>



#### *Cómo se inicia un proyecto*

**La escena:** Sala de juntas en CPI Corporation, empresa (ficticia) que manufactura productos de consumo para uso doméstico y comercial.

**Participantes:** Mal Golden, alto directivo de desarrollo de productos; Lisa Pérez, gerente comercial; Lee Warren, gerente de ingeniería; Joe Camalleri, VP ejecutivo, desarrollo de negocios.

#### **La conversación:**

**Joe:** Oye, Lee, ¿qué es eso que oí acerca de que tu gente va a desarrollar no sé qué? ¿Una caja inalámbrica universal general?

**Lee:** Es sensacional... más o menos del tamaño de una caja de cerillos pequeña... podemos conectarla a sensores de todo tipo, una cámara digital... a cualquier cosa. Usa el protocolo 802.11g inalámbrico. Permite el acceso a la salida de dispositivos sin cables. Pensamos que llevará a toda una nueva generación de productos.

**Joe:** ¿Estás de acuerdo, Mal?

**Mal:** Sí. En realidad, con las ventas tan planas que hemos tenido este año necesitamos algo nuevo. Lisa y yo hemos hecho algo de investigación del mercado y pensamos que tenemos una línea de productos que podría ser algo grande.

<sup>17</sup> El proyecto *CasaSegura* se usará en todo el libro para ilustrar los entretelones de un equipo de proyecto que elabora un producto de software. La compañía, el proyecto y las personas son ficticias, pero las situaciones y problemas son reales.

**Joe:** ¿Cuán grande... tanto como el renglón de utilidades?

**Mal (que evita el compromiso directo):** Cuéntale nuestra idea, Lisa.

**Lisa:** Es toda una nueva generación que hemos llamado “productos para la administración del hogar”. Le dimos el nombre de *CasaSegura*. Usan la nueva interfaz inalámbrica, proporcionan a los dueños de viviendas o pequeños negocios un sistema controlado por su PC —seguridad del hogar, vigilancia, control de aparatos y equipos—, tú sabes, apaga el aire acondicionado cuando sales de casa, esa clase de cosas.

**Lee (dando un brinco):** La oficina de ingeniería hizo un estudio de factibilidad técnica de esta idea, Joe. Es algo realizable con un

costo bajo de manufactura. La mayor parte del hardware es de línea. Queda pendiente el software, pero no es algo que no podamos hacer.

**Joe:** Interesante. Pero pregunté sobre las utilidades.

**Mal:** Las PC han penetrado a 70 por ciento de los hogares de Estados Unidos. Si lo vendemos en el precio correcto, podría ser una aplicación sensacional. Nadie tiene nuestra caja inalámbrica... somos dueños. Nos adelantaremos dos años a la competencia. ¿Las ganancias? Quizá tanto como 30 a 40 millones de dólares en el segundo año.

**Joe (sonriente):** Llevemos esto al siguiente nivel. Estoy interesado.

Con excepción de una referencia casual, el software no se mencionó en la conversación. Y, sin embargo, es lo que hará triunfar o fracasar la línea de productos *CasaSegura*. El esfuerzo de ingeniería tendrá éxito sólo si también lo tiene el software de *CasaSegura*. El mercado aceptará el producto sólo si el software incrustado en éste satisface las necesidades del cliente (aún no establecidas). En muchos de los capítulos siguientes continuaremos el avance de la ingeniería del software en *CasaSegura*.

## 1.8 RESUMEN

El software es un elemento clave en la evolución de sistemas y productos basados en computadoras, y una de las tecnologías más importantes en todo el mundo. En los últimos 50 años, el software ha pasado de ser la solución de un problema especializado y herramienta de análisis de la información a una industria en sí misma. No obstante, aún hay problemas para desarrollar software de alta calidad a tiempo y dentro del presupuesto asignado.

El software —programas, datos e información descriptiva— se dirige a una gama amplia de tecnología y campos de aplicación. El software heredado sigue planteando retos especiales a quienes deben darle mantenimiento.

Los sistemas y aplicaciones basados en web han evolucionado de simples conjuntos de contenido de información a sistemas sofisticados que presentan una funcionalidad compleja y contenido en multimedios. Aunque dichas *webapps* tienen características y requerimientos únicos, son software.

La ingeniería de software incluye procesos, métodos y herramientas que permiten elaborar a tiempo y con calidad sistemas complejos basados en computadoras. El proceso de software incorpora cinco actividades estructurales: comunicación, planeación, modelado, construcción y despliegue que son aplicables a todos los proyectos de software. La práctica de la ingeniería de software es una actividad para resolver problemas, que sigue un conjunto de principios fundamentales.

Muchos mitos del software todavía hacen que administradores y trabajadores se equivoquen, aun cuando ha aumentado nuestro conocimiento colectivo del software y las tecnologías requeridas para elaborarlo. Conforme el lector aprenda más sobre ingeniería de software, comenzará a entender por qué deben rebatirse estos mitos cada vez que surjan.

## PROBLEMAS Y PUNTOS POR EVALUAR

**1.1.** Dé al menos cinco ejemplos de la forma en que se aplica la ley de las consecuencias imprevistas al software de cómputo.

- 1.2. Diga algunos ejemplos (tanto positivos como negativos) que indiquen el efecto del software en nuestra sociedad.
- 1.3. Desarrolle sus propias respuestas a las cinco preguntas planteadas al principio de la sección 1.1. Analícelas con sus compañeros estudiantes.
- 1.4. Muchas aplicaciones modernas cambian con frecuencia, antes de que se presenten al usuario final y después de que la primera versión ha entrado en uso. Sugiera algunos modos de elaborar software para detener el deterioro que produce el cambio.
- 1.5. Considere las siete categorías de software presentadas en la sección 1.1.2. ¿Piensa que puede aplicarse a cada una el mismo enfoque de ingeniería de software? Explique su respuesta.
- 1.6. La figura 1.3 muestra las tres capas de la ingeniería de software arriba de otra llamada “compromiso con la calidad”. Esto implica un programa de calidad organizacional como el enfoque de la administración total de la calidad. Haga un poco de investigación y desarrolle los lineamientos de los elementos clave de un programa para la administración de la calidad.
- 1.7. ¿Es aplicable la ingeniería de software cuando se elaboran *webapps*? Si es así, ¿cómo puede modificarse para que asimile las características únicas de éstas?
- 1.8. A medida que el software gana ubicuidad, los riesgos para el público (debidos a programas defectuosos) se convierten en motivo de preocupación significativa. Desarrolle un escenario catastrófico pero realista en el que la falla de un programa de cómputo pudiera ocasionar un gran daño (económico o humano).
- 1.9. Describa con sus propias palabras una estructura de proceso. Cuando se dice que las actividades estructurales son aplicables a todos los proyectos, ¿significa que se realizan las mismas tareas en todos los proyectos sin que importe su tamaño y complejidad? Explique su respuesta.
- 1.10. Las actividades sombilla ocurren a través de todo el proceso del software. ¿Piensa usted que son aplicables por igual a través del proceso, o que algunas se concentran en una o más actividades estructurales?
- 1.11. Agregue dos mitos adicionales a la lista presentada en la sección 1.6. También diga la realidad que acompaña al mito.

## LECTURAS ADICIONALES Y FUENTES DE INFORMACIÓN<sup>18</sup>

Hay literalmente miles de libros escritos sobre software de cómputo. La gran mayoría analiza lenguajes de programación o aplicaciones de software, pero algunos estudian al software en sí mismo. Pressman y Herron (*Software Shock*, Dorset House, 1991) presentaron un estudio temprano (dirigido a las personas comunes) sobre el software y la forma en la que lo elaboran los profesionales. El libro de Negroponte que se convirtió en un éxito de ventas (*Being Digital*, Alfred A. Knopf, Inc., 1995) describe el panorama de la computación y su efecto general en el siglo xxi. DeMarco (*Why Does Software Cost So Much?*, Dorset House, 1995) ha producido varios ensayos amenos y profundos sobre el software y el proceso con el que se elabora.

Minasi (*The Software Conspiracy: Why Software Companies Put out Faulty Products, How They Can Hurt You, and What You Can Do*, McGraw-Hill, 2000) afirma que el “flagelo moderno” de los errores en el software puede eliminarse y sugiere formas de lograrlo. Compaine (*Digital Divide: Facing A Crisis or Creating a Myth*, MIT Press, 2001) asegura que la “división” entre aquellos que tienen acceso a recursos de la información (por ejemplo, la web) y los que no lo tienen se está estrechando conforme avanzamos en la primera década de este siglo. Los libros escritos por Greenfield (*Everyware: The Dawning Age of Ubiquitous Computing*, New Riders Publishing, 2006) y Loke (*Context-Aware Pervasive Systems: Architectures for a New Breed of Applications*, Auerbach, 2006) introducen el concepto de software de “mundo abierto” y predicen un ambiente inalámbrico en el que el software deba adaptarse a los requerimientos que surjan en tiempo real.

<sup>18</sup> La sección de “Lecturas adicionales y fuentes de información” que se presenta al final de cada capítulo expone un panorama breve de fuentes impresas que ayudan a aumentar la comprensión de los principales temas presentados. El autor ha creado un sitio web para apoyar al libro *Ingeniería de software: enfoque del profesional* en [www.mhhe.com/compsci/pressman](http://www.mhhe.com/compsci/pressman). Entre los muchos temas que se abordan en dicho sitio, se encuentran desde los recursos de la ingeniería de software capítulo por capítulo hasta información basada en web que complementa el material presentado. Como parte de esos recursos se halla un vínculo hacia Amazon.com para cada libro citado en esta sección.

El estado actual de la ingeniería y del proceso de software se determina mejor a partir de publicaciones tales como *IEEE Software*, *IEEE Computer*, *CrossTalk* y *IEEE Transactions on Software Engineering*. Publicaciones periódicas como *Application Development Trends* y *Cutter IT Journal* con frecuencia contienen artículos sobre temas de ingeniería de software. La disciplina se “resume” cada año en *Proceeding of the International Conference on Software Engineering*, patrocinada por IEEE y ACM, y se analiza a profundidad en revistas tales como *ACM Transactions on Software Engineering and Methodology*, *ACM Software Engineering Notes* y *Annals of Software Engineering*. Hay decenas de miles de sitios web dedicados a la ingeniería y al proceso de software.

En años recientes se han publicado muchos libros que abordan el proceso y la ingeniería de software. Algunos presentan un panorama de todo el proceso, mientras otros profundizan en algunos temas importantes y omiten otros. Entre los más populares (¡además del que tiene usted en sus manos!) se encuentran los siguientes:

- Abran, A., and J. Moore, *SWEBOK: Guide to the Software Engineering Body of Knowledge*, IEEE, 2002.
- Andersson, E., et al., *Software Engineering for Internet Applications*, The MIT Press, 2006.
- Christensen, M., and R. Thayer, *A Project Manager's Guide to Software Engineering Best Practices*, IEEE-CS Press (Wiley), 2002.
- Glass, R., *Fact and Fallacies of Software Engineering*, Addison-Wesley, 2002.
- Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach*, 2d ed., Addison-Wesley, 2008.
- Jalote, P., *An Integrated Approach to Software Engineering*, Springer, 2006.
- Pfleeger, S., *Software Engineering: Theory and Practice*, 3d ed., Prentice-Hall, 2005.
- Schach, S., *Object-Oriented and Classical Software Engineering*, 7th ed., McGraw-Hill, 2006.
- Sommerville, I., *Software Engineering*, 8th ed., Addison-Wesley, 2006.
- Tsui, F., and O. Karam, *Essentials of Software Engineering*, Jones & Bartlett Publishers, 2006.

En las últimas décadas, son muchos los estándares para la ingeniería de software que han sido publicados por IEEE, ISO y sus organizaciones. Moore (*The Road Map to Software Engineering: A Standards-Based Guide*, Wiley-IEEE Computer Society Press, 2006) proporciona una revisión útil de los estándares relevantes y la forma en la que se aplican a proyectos reales.

En internet se encuentra disponible una amplia variedad de fuentes acerca de la ingeniería y el proceso de software. Una lista actualizada de referencias en la Red Mundial que son útiles para el proceso de software se encuentra en el sitio web del libro, en la dirección **[www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm](http://www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm)**.



## EL PROCESO DEL SOFTWARE

**E**n esta parte de la obra, aprenderá sobre el proceso que genera una estructura para la práctica de la ingeniería de software. En los capítulos que siguen se abordan preguntas como las siguientes:

- ¿Qué es el proceso del software?
- ¿Cuáles son las actividades estructurales generales que están presentes en todo proceso del software?
- ¿Cómo se modelan los procesos y cuáles son los patrones del proceso?
- ¿Cuáles son los modelos prescriptivos del proceso y cuáles son sus fortalezas y debilidades?
- ¿Por qué la *agilidad* es un imperativo en la ingeniería de software moderna?
- ¿Qué es un desarrollo ágil del software y en qué se diferencia de los modelos más tradicionales del proceso?

Una vez respondidas estas preguntas, el lector estará mejor preparado para entender el contexto en el que se aplica la práctica de la ingeniería de software.

## CONCEPTOS CLAVE

conjunto de tareas.....	29
desarrollo basado en componentes .....	43
modelo de métodos formales.....	44
modelo general de proceso...	27
modelos concurrentes .....	40
modelos de proceso evolutivo .....	36
modelos de proceso incremental.....	35
modelos de proceso prescriptivo .....	33
patrones del proceso.....	29
proceso del equipo de software .....	49
proceso personal del software.....	48
proceso unificado .....	45

**E**n un libro fascinante que expone el punto de vista de un economista sobre el software y su ingeniería, Howard Baetjer, Jr. [Bae98] comenta acerca del proceso del software.

Debido a que el software, como todo capital, es conocimiento incorporado y a que el conocimiento originalmente se halla disperso, tácito, latente e incompleto en gran medida, el desarrollo de software es un proceso de aprendizaje social. El proceso es un diálogo en el que el conocimiento que debe convertirse en software se reúne e incorpora en éste. El proceso genera interacción entre usuarios y diseñadores, entre usuarios y herramientas cambiantes, y entre diseñadores y herramientas en evolución [tecnología]. Es un proceso que se repite y en el que la herramienta que evoluciona sirve por sí misma como medio para la comunicación: con cada nueva ronda del diálogo se genera más conocimiento útil a partir de las personas involucradas.

En realidad, la elaboración de software de computadora es un proceso reiterativo de aprendizaje social, y el resultado, algo que Baetjer llamaría “capital de software”, es la reunión de conocimiento recabado, depurado y organizado a medida que se realiza el proceso.

Pero desde el punto de vista técnico, ¿qué es exactamente un proceso del software? En el contexto de este libro, se define *proceso del software* como una estructura para las actividades, acciones y tareas que se requieren a fin de construir software de alta calidad. ¿“Proceso” es sinónimo de “ingeniería de software”? La respuesta es “sí y no”. Un proceso del software define el enfoque adoptado mientras se hace ingeniería sobre el software. Pero la ingeniería de software también incluye tecnologías que pueblan el proceso: métodos técnicos y herramientas automatizadas.

Más importante aún, la ingeniería de software es llevada a cabo por personas creativas y preparadas que deben adaptar un proceso maduro de software a fin de que resulte apropiado para los productos que construyen y para las demandas de su mercado.

UNA  
MIRADA  
RÁPIDA

**¿Qué es?** Cuando se trabaja en la construcción de un producto o sistema, es importante ejecutar una serie de pasos predecibles —el mapa de carreteras que lo ayuda a obtener a tiempo un resultado de alta calidad—. El mapa que se sigue se llama “proceso del software”.

**¿Quién lo hace?** Los ingenieros de software y sus gerentes adaptan el proceso a sus necesidades y luego lo siguen. Además, las personas que solicitaron el software tienen un papel en el proceso de definición, elaboración y prueba.

**¿Por qué es importante?** Porque da estabilidad, control y organización a una actividad que puede volverse caótica si se descontrola. Sin embargo, un enfoque moderno de ingeniería de software debe ser “ágil”. Debe incluir sólo aquellas actividades, controles y productos del trabajo que sean apropiados para el equipo del proyecto y para el producto que se busca obtener.

**¿Cuáles son los pasos?** En un nivel detallado, el proceso que se adopte depende del software que se esté elaborando. Un proceso puede ser apropiado para crear software destinado a un sistema de control electrónico de un aeroplano, mientras que para la creación de un sitio web será necesario un proceso completamente distinto.

**¿Cuál es el producto final?** Desde el punto de vista de un ingeniero de software, los productos del trabajo son los programas, documentos y datos que se producen como consecuencia de las actividades y tareas definidas por el proceso.

**¿Cómo me aseguro de que lo hice bien?** Hay cierto número de mecanismos de evaluación del proceso del software que permiten que las organizaciones determinen la “madurez” de su proceso. Sin embargo, la calidad, oportunidad y viabilidad a largo plazo del producto que se elabora son los mejores indicadores de la eficacia del proceso que se utiliza.

## 2.1 UN MODELO GENERAL DE PROCESO

En el capítulo 1 se definió un proceso como la colección de actividades de trabajo, acciones y tareas que se realizan cuando va a crearse algún producto terminado. Cada una de las actividades, acciones y tareas se encuentra dentro de una estructura o modelo que define su relación tanto con el proceso como entre sí.

En la figura 2.1 se representa el proceso del software de manera esquemática. En dicha figura, cada actividad estructural está formada por un conjunto de acciones de ingeniería de software y cada una de éstas se encuentra definida por un *conjunto de tareas* que identifica las tareas del trabajo que deben realizarse, los productos del trabajo que se producirán, los puntos de aseguramiento de la calidad que se requieren y los puntos de referencia que se utilizarán para evaluar el avance.

Como se dijo en el capítulo 1, una estructura general para la ingeniería de software define cinco actividades estructurales: **comunicación, planeación, modelado, construcción y despliegue**. Además, a lo largo de todo el proceso se aplica un conjunto de actividades som-

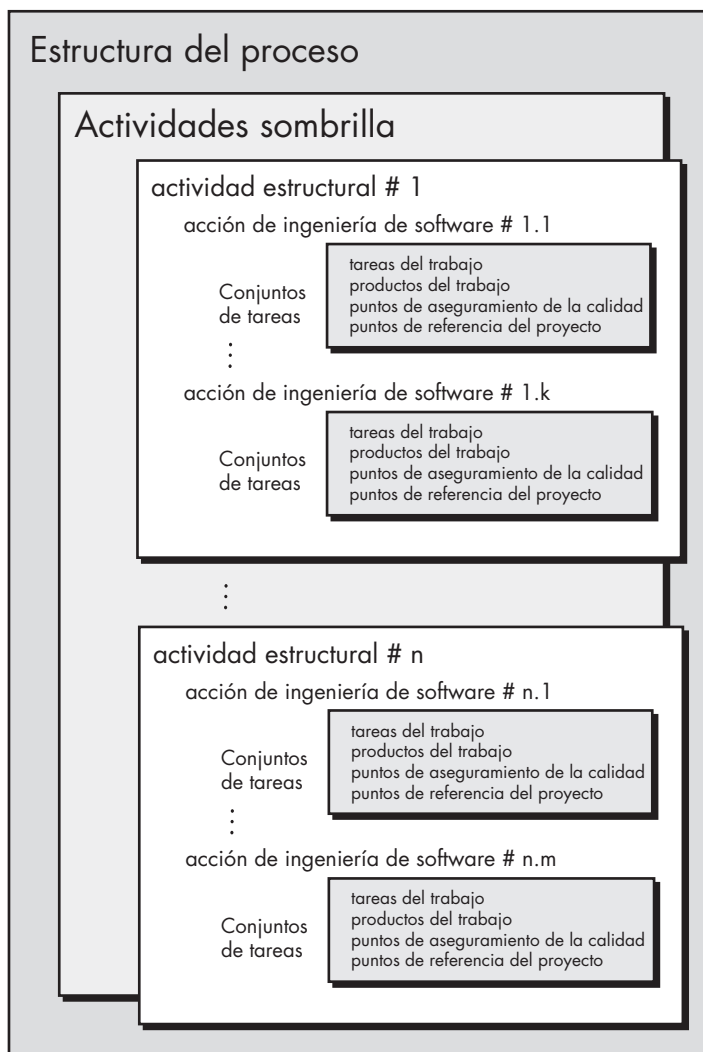
### PUNTO CLAVE

La jerarquía del trabajo técnico dentro del proceso del software es: actividades, acciones que contiene y tareas constituyentes.

FIGURA 2.1

Estructura de un proceso del software

### Proceso del software



**Cita:**

"Pensamos que los desarrolladores de software pierden de vista una verdad fundamental: la mayor parte de organizaciones no saben lo que hacen. Piensan que lo saben, pero no es así."

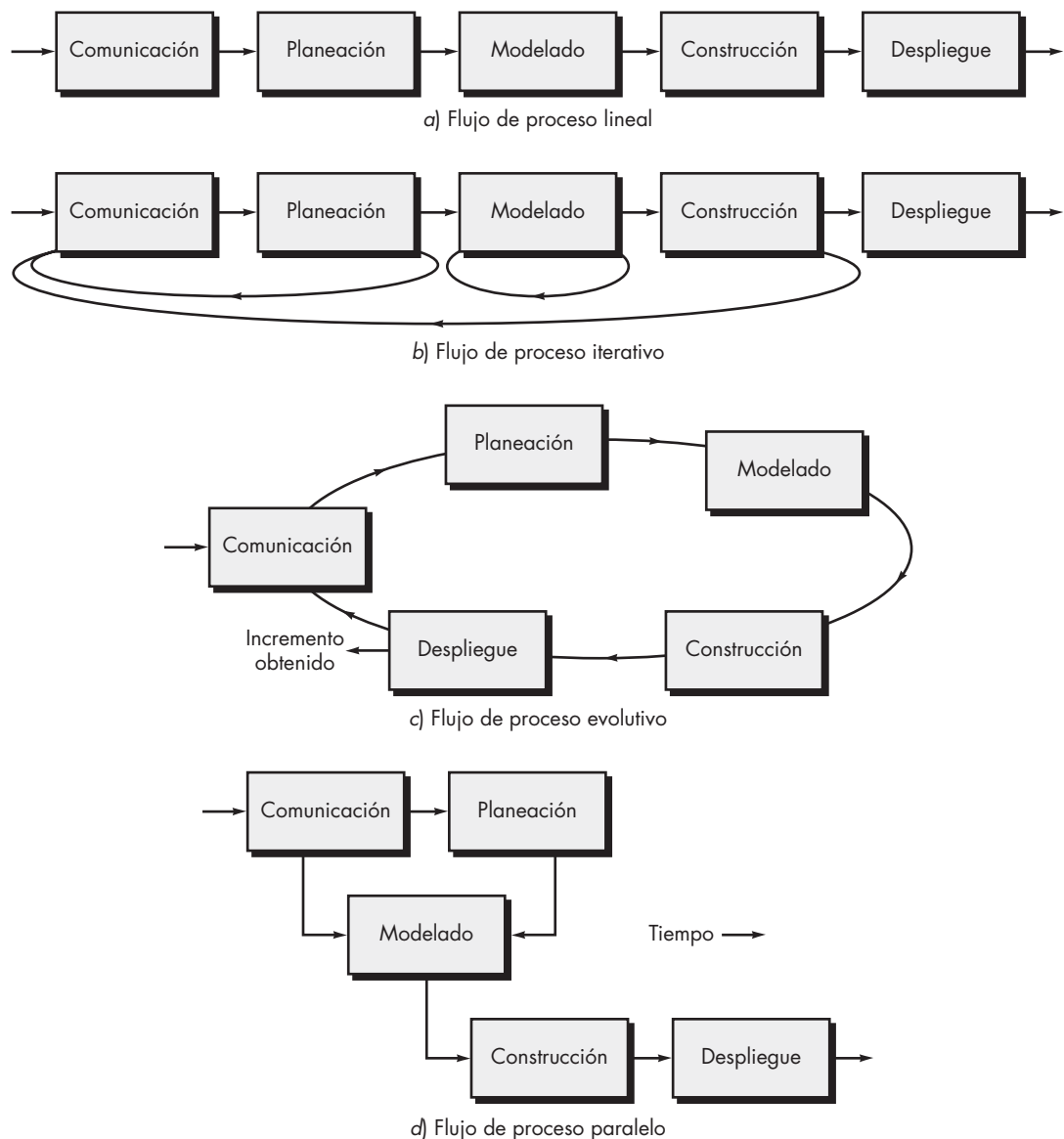
Tom DeMarco

brilla: seguimiento y control del proyecto, administración de riesgos, aseguramiento de la calidad, administración de la configuración, revisiones técnicas, entre otras.

El lector debe observar que aún no se menciona un aspecto importante del proceso del software. En la figura 2.2 se ilustra dicho aspecto —llamado *flujo del proceso*— y se describe la manera en que están organizadas las actividades estructurales y las acciones y tareas que ocurren dentro de cada una con respecto de la secuencia y el tiempo.

Un *flujo de proceso lineal* ejecuta cada una de las cinco actividades estructurales en secuencia, comenzando por la comunicación y terminando con el despliegue (véase la figura 2.2a). Un *flujo de proceso iterativo* repite una o más de las actividades antes de pasar a la siguiente (véase la figura 2.2b). Un *flujo de proceso evolutivo* realiza las actividades en forma "circular". A través de las cinco actividades, cada circuito lleva a una versión más completa del software (véase la figura 2.2c). Un *flujo de proceso paralelo* (véase la figura 2.2d) ejecuta una o más actividades en

**FIGURA 2.2** Flujo del proceso



paralelo con otras (por ejemplo, el modelado de un aspecto del software tal vez se ejecute en paralelo con la construcción de otro aspecto del software).

### 2.1.1 Definición de actividad estructural

Aunque en el capítulo 1 se describieron cinco actividades estructurales y se dio una definición básica de cada una, un equipo de software necesitará mucha más información antes de poder ejecutar de manera apropiada cualquiera de ellas como parte del proceso del software. Por tanto, surge una pregunta clave: *¿qué acciones son apropiadas para una actividad estructural, dados la naturaleza del problema por resolver, las características de las personas que hacen el trabajo y los participantes que patrocinan el proyecto?*

**?** ¿Cómo se transforma una actividad estructural cuando cambia la naturaleza del proyecto?

Para un proyecto de software pequeño solicitado por una persona (en una ubicación remota) con requerimientos sencillos y directos, la actividad de comunicación tal vez no incluya algo más que una llamada telefónica con el participante apropiado. Entonces, la única acción necesaria es una *conversación telefónica*, y las tareas del trabajo (el *conjunto de tareas*) que engloba son las siguientes:

1. Hacer contacto con el participante por vía telefónica.
2. Analizar los requerimientos y tomar notas.
3. Organizar las notas por escrito en una formulación breve de los requerimientos.
4. Enviar correo electrónico al participante para que revise y apruebe.

Si el proyecto fuera considerablemente más complejo, con muchos participantes y cada uno con un distinto conjunto de requerimientos (a veces en conflicto), la actividad de comunicación puede tener seis acciones distintas (descritas en el capítulo 5): *concepción, indagación, elaboración, negociación, especificación y validación*. Cada una de estas acciones de la ingeniería del software tendrá muchas tareas de trabajo y un número grande de diferentes productos finales.

#### **PUNTO CLAVE**

Diferentes proyectos demandan diferentes conjuntos de tareas. El equipo de software elige el conjunto de tareas con base en las características del problema y el proyecto.

### 2.1.2 Identificación de un conjunto de tareas

En relación con la figura 2.1, cada acción de la ingeniería de software (por ejemplo, *obtención*, asociada a la actividad de comunicación) se representa por cierto número de distintos *conjuntos de tareas*, cada uno de los cuales es una colección de tareas de trabajo de la ingeniería de software, relacionadas con productos del trabajo, puntos de aseguramiento de la calidad y puntos de referencia del proyecto. Debe escogerse el conjunto de tareas que se adapte mejor a las necesidades del proyecto y a las características del equipo. Esto implica que una acción de la ingeniería de software puede adaptarse a las necesidades específicas del proyecto de software y a las características del equipo del proyecto.

### 2.1.3 Patrones del proceso

Cada equipo de software se enfrenta a problemas conforme avanza en el proceso del software. Si se demostrara que existen soluciones fáciles para dichos problemas, sería útil para el equipo abordarlos y resolverlos rápidamente. Un *patrón del proceso*<sup>1</sup> describe un problema relacionado con el proceso que se encuentra durante el trabajo de ingeniería de software, identifica el ambiente en el que surge el problema y sugiere una o más soluciones para el mismo. Dicho de manera general, un patrón de proceso da un formato [Amb98]: un método consistente para describir soluciones del problema en el contexto del proceso del software. Al combinar patrones, un equipo de software resuelve problemas y construye el proceso que mejor satisfaga las necesidades de un proyecto.

**?** ¿Qué es un patrón del proceso?

<sup>1</sup> En el capítulo 12 se hace el análisis detallado de los patrones.

## INFORMACIÓN

**Conjunto de tareas**

Un conjunto de tareas define el trabajo real por efectuar a fin de cumplir los objetivos de una acción de ingeniería de software. Por ejemplo, la *indagación* (mejor conocida como “recabar los requerimientos”) es una acción importante de la ingeniería de software que ocurre durante la actividad de comunicación. La meta al recabar los requerimientos es entender lo que los distintos participantes desean del software que se va a elaborar.

Para un proyecto pequeño y relativamente sencillo, el conjunto de tareas para la indagación de requerimientos tendrá un aspecto parecido al siguiente:

1. Elaborar la lista de participantes del proyecto.
2. Invitar a todos los participantes a una reunión informal.
3. Pedir a cada participante que haga una relación de las características y funciones que requiere.
4. Analizar los requerimientos y construir la lista definitiva.
5. Ordenar los requerimientos según su prioridad.
6. Identificar las áreas de incertidumbre.

Para un proyecto de software más grande y complejo se requerirá de un conjunto de tareas diferente que quizá esté constituido por las siguientes tareas de trabajo:

1. Hacer la lista de participantes del proyecto.
2. Entrevistar a cada participante por separado a fin de determinar los deseos y necesidades generales.

3. Formar la lista preliminar de las funciones y características con base en las aportaciones del participante.
4. Programar una serie de reuniones para facilitar la elaboración de las especificaciones de la aplicación.
5. Celebrar las reuniones.
6. Producir en cada reunión escenarios informales de usuario.
7. Afinar los escenarios del usuario con base en la retroalimentación de los participantes.
8. Formar una lista revisada de los requerimientos de los participantes.
9. Usar técnicas de despliegue de la función de calidad para asignar prioridades a los requerimientos.
10. Agrupar los requerimientos de modo que puedan entregarse en forma paulatina y creciente.
11. Resaltar las limitantes y restricciones que se introducirán al sistema.
12. Analizar métodos para validar el sistema.

Los dos conjuntos de tareas mencionados sirven para “recabar los requerimientos”, pero son muy distintos en profundidad y formalidad. El equipo de software elige el conjunto de tareas que le permita alcanzar la meta de cada acción con calidad y agilidad.

**Cita:**

“La repetición de patrones es algo muy diferente de la repetición de las partes. En realidad, las distintas partes serán únicas porque los patrones son los mismos.”

Christopher Alexander

Los patrones se definen en cualquier nivel de abstracción.<sup>2</sup> En ciertos casos, un patrón puede usarse para describir un problema (y su solución) asociado con un modelo completo del proceso (por ejemplo, hacer prototipos). En otras situaciones, los patrones se utilizan para describir un problema (y su solución) asociado con una actividad estructural (por ejemplo, **planeación**) o una acción dentro de una actividad estructural (estimación de proyectos).

Ambler [Amb98] ha propuesto un formato para describir un patrón del proceso:

**Nombre del patrón.** El patrón recibe un nombre significativo que lo describe en el contexto del proceso del software (por ejemplo, **RevisionesTécnicas**).

**Fuerzas.** El ambiente en el que se encuentra el patrón y los aspectos que hacen visible el problema y afectan su solución.

**Tipo.** Se especifica el tipo de patrón. Ambler [Amb98] sugiere tres tipos:

1. *Patrón de etapa:* define un problema asociado con una actividad estructural para el proceso. Como una actividad estructural incluye múltiples acciones y tareas del trabajo, un patrón de la etapa incorpora múltiples patrones de la tarea (véase a continuación) que son relevantes para la etapa (actividad estructural). Un ejemplo de patrón de etapa sería **EstablecerComunicación**. Este patrón incorporaría el patrón de tarea **RecabarRequerimientos** y otros más.
2. *Patrón de tarea:* define un problema asociado con una acción o tarea de trabajo de la ingeniería de software y que es relevante para el éxito de la práctica de ingeniería de software (por ejemplo, **RecabarRequerimientos** es un patrón de tarea).

2 Los patrones son aplicables a muchas actividades de la ingeniería de software. El análisis, el diseño y la prueba de patrones se estudian en los capítulos 7, 9, 10, 12 y 14. Los patrones y “antipatrones” para las actividades de administración de proyectos se analizan en la parte 4 del libro.

3. *Patrón de fase*: define la secuencia de las actividades estructurales que ocurren dentro del proceso, aun cuando el flujo general de las actividades sea de naturaleza iterativa. Un ejemplo de patrón de fase es **ModeloEspiral** o **Prototipos**.<sup>3</sup>

**Contexto inicial.** Describe las condiciones en las que se aplica el patrón. Antes de iniciar el patrón: 1) ¿Qué actividades organizacionales o relacionadas con el equipo han ocurrido? 2) ¿Cuál es el estado de entrada para el proceso? 3) ¿Qué información de ingeniería de software o del proyecto ya existe?

Por ejemplo, el patrón **Planeación** (patrón de etapa) requiere que: 1) los clientes y los ingenieros de software hayan establecido una comunicación colaboradora; 2) haya terminado con éxito cierto número de patrones de tarea [especificado] para el patrón **Comunicación**; y 3) se conozcan el alcance del proyecto, los requerimientos básicos del negocio y las restricciones del proyecto.

**Problema.** El problema específico que debe resolver el patrón.

**Solución.** Describe cómo implementar con éxito el patrón. Esta sección describe la forma en la que se modifica el estado inicial del proceso (que existe antes de implementar el patrón) como consecuencia de la iniciación del patrón. También describe cómo se transforma la información sobre la ingeniería de software o sobre el proyecto, disponible antes de que inicie el patrón, como consecuencia de la ejecución exitosa del patrón.

**Contexto resultante.** Describe las condiciones que resultarán una vez que se haya implementado con éxito el patrón: 1) ¿Qué actividades organizacionales o relacionadas con el equipo deben haber ocurrido? 2) ¿Cuál es el estado de salida del proceso? 3) ¿Qué información sobre la ingeniería de software o sobre el proyecto se ha desarrollado?

**Patrones relacionados.** Proporciona una lista de todos los patrones de proceso directamente relacionados con éste. Puede representarse como una jerarquía o en alguna forma de diagrama. Por ejemplo, el patrón de etapa **Comunicación** incluye los patrones de tarea: **EquipoDelProyecto**, **LineamientosDeColaboración**, **DefiniciónDeAlcances**, **RecabarRequerimientos**, **DescripciónDeRestricciones** y **CreaciónDeEscenarios**.

**Usos y ejemplos conocidos.** Indica las instancias específicas en las que es aplicable el patrón. Por ejemplo, **Comunicación** es obligatoria al principio de todo proyecto de software, es recomendable a lo largo del proyecto y de nuevo obligatoria una vez alcanzada la actividad de despliegue.

Los patrones de proceso dan un mecanismo efectivo para enfrentar problemas asociados con cualquier proceso del software. Los patrones permiten desarrollar una descripción jerárquica del proceso, que comienza en un nivel alto de abstracción (un patrón de fase). Después se mejora la descripción como un conjunto de patrones de etapa que describe las actividades estructurales y se mejora aún más en forma jerárquica en patrones de tarea más detallados para cada patrón de etapa. Una vez desarrollados los patrones de proceso, pueden reutilizarse para la definición de variantes del proceso, es decir, un equipo de software puede definir un modelo de proceso específico con el empleo de los patrones como bloques constituyentes del modelo del proceso.

#### WebRef

En la dirección [www.ambyssoft.com/processPatternsPage.html](http://www.ambyssoft.com/processPatternsPage.html) se encuentran recursos amplios sobre los patrones de proceso.

## 2.2 EVALUACIÓN Y MEJORA DEL PROCESO

La existencia de un proceso del software no es garantía de que el software se entregue a tiempo, que satisfaga las necesidades de los consumidores o que tenga las características técnicas que

<sup>3</sup> Estos patrones de fase se estudian en la sección 2.3.3.

## INFORMACIÓN

**Ejemplo de patrón de proceso**

El siguiente patrón de proceso abreviado describe un enfoque aplicable en el caso en el que los participantes tienen una idea general de lo que debe hacerse, pero no están seguros de los requerimientos específicos de software.

**Nombre del patrón. Requerimientos Poco Claros**

**Intención.** Este patrón describe un enfoque para construir un modelo (un prototipo) que los participantes pueden evaluar en forma iterativa, en un esfuerzo por identificar o solidificar los requerimientos de software.

**Tipo.** Patrón de fase.

**Contexto inicial.** Antes de iniciar este patrón deben cumplirse las siguientes condiciones: 1) se ha identificado a los participantes; 2) se ha establecido un modo de comunicación entre los participantes y el equipo de software; 3) los participantes han identificado el problema general de software que se va a resolver; 4) se ha obtenido el entendimiento inicial del alcance del proyecto, los requerimientos básicos del negocio y las restricciones del proyecto.

**Problema.** Los requerimientos son confusos o inexistentes, pero hay un reconocimiento claro de que existe un problema por resolver y que

debe hacerse con una solución de software. Los participantes no están seguros de lo que quieren, es decir, no pueden describir con detalle los requerimientos del software.

**Solución.** Aquí se presentaría una descripción del proceso prototipo, que se describirá más adelante, en la sección 2.3.3.

**Contexto resultante.** Los participantes aprueban un prototipo de software que identifica los requerimientos básicos (por ejemplo, modos de interacción, características computacionales, funciones de procesamiento). Después de esto, 1) el prototipo quizá evolucione a través de una serie de incrementos para convertirse en el software de producción, o 2) tal vez se descarte el prototipo y el software de producción se elabore con el empleo de otro proceso de patrón.

**Patrones relacionados.** Los patrones siguientes están relacionados con este patrón: **Comunicación Con Clientes, Diseño Iterativo, Desarrollo Iterativo, Evaluación Del Cliente, Obtención De Requerimientos.**

**Usos y ejemplos conocidos.** Cuando los requerimientos sean inciertos, es recomendable hacer prototipos.

**PUNTO CLAVE**

La evaluación busca entender el estado actual del proceso del software con el objeto de mejorarlo.

conducirán a características de calidad de largo plazo (véanse los capítulos 14 y 16). Los patrones de proceso deben acoplarse con una práctica sólida de ingeniería de software (véase la parte 2 del libro). Además, el proceso en sí puede evaluarse para garantizar que cumple con ciertos criterios de proceso básicos que se haya demostrado que son esenciales para el éxito de la ingeniería de software.<sup>4</sup>

En las últimas décadas se han propuesto numerosos enfoques para la evaluación y mejora de un proceso del software:

**?** ¿De qué técnicas formales se dispone para evaluar el proceso del software?

**Cita:**

"Las organizaciones de software tienen deficiencias significativas en su capacidad de capitalizar las experiencias obtenidas de los proyectos terminados."

NASA

**Método de evaluación del estándar CMMI para el proceso de mejora (SCAMPI,** por sus siglas en inglés): proporciona un modelo de cinco fases para evaluar el proceso: inicio, diagnóstico, establecimiento, actuación y aprendizaje. El método SCAMPI emplea el SEI CMMI como la base de la evaluación [SEI00].

**Evaluación basada en CMM para la mejora del proceso interno (CBA IPI,** por sus siglas en inglés): proporciona una técnica de diagnóstico para evaluar la madurez relativa de una organización de software; usa el SEI CMM como la base de la evaluación [Dun01].

**SPICE (ISO/IEC 15504):** estándar que define un conjunto de requerimientos para la evaluación del proceso del software. El objetivo del estándar es ayudar a las organizaciones a desarrollar una evaluación objetiva de cualquier proceso del software definido [ISO08].

**ISO9001:2000 para software:** estándar genérico que se aplica a cualquier organización que desee mejorar la calidad general de los productos, sistemas o servicios que proporciona. Por tanto, el estándar es directamente aplicable a las organizaciones y compañías de software [Ant06].

En el capítulo 30 se presenta un análisis detallado de los métodos de evaluación del software y del proceso de mejora.

<sup>4</sup> En la publicación CMMI [CMM07] del SEI, se describen con muchos detalles las características de un proceso del software y los criterios para un proceso exitoso.

## 2.3 MODELOS DE PROCESO PRESCRIPTIVO

Los modelos de proceso prescriptivo fueron propuestos originalmente para poner orden en el caos del desarrollo de software. La historia indica que estos modelos tradicionales han dado cierta estructura útil al trabajo de ingeniería de software y que constituyen un mapa razonablemente eficaz para los equipos de software. Sin embargo, el trabajo de ingeniería de software y el producto que genera siguen “al borde del caos”.

En un artículo intrigante sobre la extraña relación entre el orden y el caos en el mundo del software, Nogueira y sus colegas [Nog00] afirman lo siguiente:

**Cita:**

“Si el proceso está bien, los resultados cuidarán de sí mismos.”

Takashi Osada

El borde del caos se define como “el estado natural entre el orden y el caos, un compromiso grande entre la estructura y la sorpresa” [Kau95]. El borde del caos se visualiza como un estado inestable y parcialmente estructurado [...] Es inestable debido a que se ve atraído constantemente hacia el caos o hacia el orden absoluto.

Tenemos la tendencia de pensar que el orden es el estado ideal de la naturaleza. Esto podría ser un error [...] las investigaciones apoyan la teoría de que la operación que se aleja del equilibrio genera creatividad, procesos autoorganizados y rendimientos crecientes [Roo96]. El orden absoluto significa ausencia de variabilidad, que podría ser una ventaja en los ambientes impredecibles. El cambio ocurre cuando hay cierta estructura que permita que el cambio pueda organizarse, pero que no sea tan rígida como para que no pueda suceder. Por otro lado, demasiado caos hace imposible la coordinación y la coherencia. La falta de estructura no siempre significa desorden.

Las implicaciones filosóficas de este argumento son significativas para la ingeniería de software. Si los modelos de proceso prescriptivo<sup>5</sup> buscan generar estructura y orden, ¿son inapropiados para el mundo del software, que se basa en el cambio? Pero si rechazamos los modelos de proceso tradicional (y el orden que implican) y los reemplazamos con algo menos estructurado, ¿hacemos imposible la coordinación y coherencia en el trabajo de software?

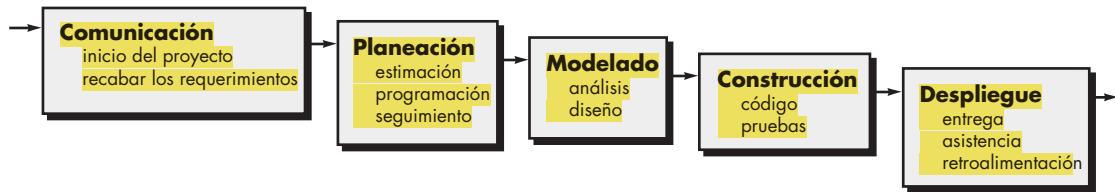
No hay respuestas fáciles para estas preguntas, pero existen alternativas disponibles para los ingenieros de software. En las secciones que siguen se estudia el enfoque de proceso prescriptivo en el que los temas dominantes son el orden y la consistencia del proyecto. El autor los llama “prescriptivos” porque prescriben un conjunto de elementos del proceso: actividades estructurales, acciones de ingeniería de software, tareas, productos del trabajo, aseguramiento de la calidad y mecanismos de control del cambio para cada proyecto. Cada modelo del proceso también prescribe un flujo del proceso (también llamado *flujo de trabajo*), es decir, la manera en la que los elementos del proceso se relacionan entre sí.

Todos los modelos del proceso del software pueden incluir las actividades estructurales generales descritas en el capítulo 1, pero cada una pone distinto énfasis en ellas y define en forma diferente el flujo de proceso que invoca cada actividad estructural (así como acciones y tareas de ingeniería de software).

### 2.3.1 Modelo de la cascada

Hay veces en las que los requerimientos para cierto problema se comprenden bien: cuando el trabajo desde la **comunicación** hasta el **despliegue** fluye en forma razonablemente lineal. Esta situación se encuentra en ocasiones cuando deben hacerse adaptaciones o mejoras bien definidas a un sistema ya existente (por ejemplo, una adaptación para software de contabilidad que es obligatorio hacer debido a cambios en las regulaciones gubernamentales). También ocurre en cierto número limitado de nuevos esfuerzos de desarrollo, pero sólo cuando los requerimientos están bien definidos y tienen una estabilidad razonable.

<sup>5</sup> Los modelos de proceso prescriptivo en ocasiones son denominados modelos de proceso “tradicional”.

**FIGURA 2.3** Modelo de la cascada

El *modelo de la cascada*, a veces llamado *ciclo de vida clásico*, sugiere un enfoque sistemático y secuencial<sup>6</sup> para el desarrollo del software, que comienza con la especificación de los requerimientos por parte del cliente y avanza a través de planeación, modelado, construcción y despliegue, para concluir con el apoyo del software terminado (véase la figura 2.3).

Una variante de la representación del modelo de la cascada se denomina *modelo en V*. En la figura 2.4 se ilustra el modelo en V [Buc99], donde se aprecia la relación entre las acciones para el aseguramiento de la calidad y aquellas asociadas con la comunicación, modelado y construcción temprana. A medida que el equipo de software avanza hacia abajo desde el lado izquierdo de la V, los requerimientos básicos del problema mejoran hacia representaciones técnicas cada vez más detalladas del problema y de su solución. Una vez que se ha generado el código, el equipo sube por el lado derecho de la V, y en esencia ejecuta una serie de pruebas (acciones para asegurar la calidad) que validan cada uno de los modelos creados cuando el equipo fue hacia abajo por el lado izquierdo.<sup>7</sup> En realidad, no hay diferencias fundamentales entre el ciclo de vida clásico y el modelo en V. Este último proporciona una forma de visualizar el modo de aplicación de las acciones de verificación y validación al trabajo de ingeniería inicial.

El modelo de la cascada es el paradigma más antiguo de la ingeniería de software. Sin embargo, en las últimas tres décadas, las críticas hechas al modelo han ocasionado que incluso sus defensores más obstinados cuestionen su eficacia [Han95]. Entre los problemas que en ocasiones surgen al aplicar el modelo de la cascada se encuentran los siguientes:

1. Es raro que los proyectos reales sigan el flujo secuencial propuesto por el modelo. Aunque el modelo lineal acepta repeticiones, lo hace en forma indirecta. Como resultado, los cambios generan confusión conforme el equipo del proyecto avanza.
2. A menudo, es difícil para el cliente enunciar en forma explícita todos los requerimientos. El modelo de la cascada necesita que se haga y tiene dificultades para aceptar la incertidumbre natural que existe al principio de muchos proyectos.
3. El cliente debe tener paciencia. No se dispondrá de una versión funcional del(de los) programa(s) hasta que el proyecto esté muy avanzado. Un error grande sería desastroso si se detectara hasta revisar el programa en funcionamiento.

En un análisis interesante de proyectos reales, Bradac [Bra94] encontró que la naturaleza lineal del ciclo de vida clásico llega a “estados de bloqueo” en los que ciertos miembros del equipo de proyecto deben esperar a otros a fin de terminar tareas interdependientes. En realidad, ¡el tiempo de espera llega a superar al dedicado al trabajo productivo! Los estados de bloqueo tienden a ocurrir más al principio y al final de un proceso secuencial lineal.

Hoy en día, el trabajo de software es acelerado y está sujeto a una corriente sin fin de cambios (en las características, funciones y contenido de información). El modelo de la cascada suele ser

### PUNTO CLAVE

El modelo en V ilustra la forma en la que se asocian las acciones de verificación y validación con las primeras acciones de ingeniería.

**?** ¿Por qué a veces falla el modelo de la cascada?

### Cita:

“Con demasiada frecuencia, el trabajo de software sigue la primera ley del ciclismo: no importa hacia dónde te dirijas, vas hacia arriba y contra el viento.”

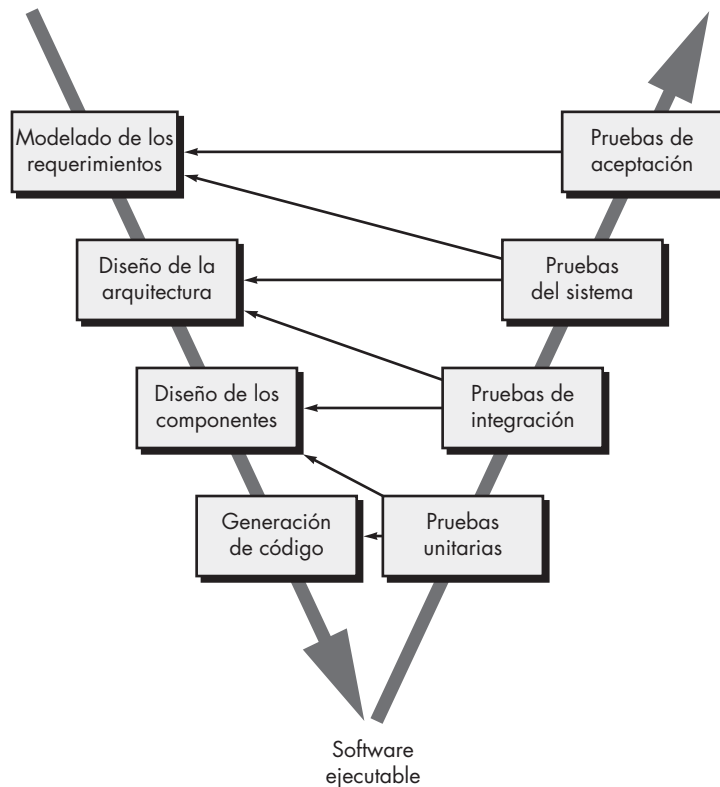
Anónimo

6 Aunque el modelo de la cascada propuesto originalmente por Winston Royce [Roy70] prevé los “bucles de retroalimentación”, la gran mayoría de organizaciones que aplican este modelo de proceso lo tratan como si fuera estrictamente lineal.

7 En la parte 3 del libro se estudian con detalle las acciones de aseguramiento de la calidad.

FIGURA 2.4

El modelo en V



inapropiado para ese tipo de labor. No obstante, sirve como un modelo de proceso útil en situaciones en las que los requerimientos son fijos y el trabajo avanza en forma lineal hacia el final.

### 2.3.2 Modelos de proceso incremental

Hay muchas situaciones en las que los requerimientos iniciales del software están razonablemente bien definidos, pero el alcance general del esfuerzo de desarrollo imposibilita un proceso lineal. Además, tal vez haya una necesidad imperiosa de dar rápidamente cierta funcionalidad limitada de software a los usuarios y aumentarla en las entregas posteriores de software. En tales casos, se elige un modelo de proceso diseñado para producir el software en incrementos.

El modelo *incremental* combina elementos de los flujos de proceso lineal y paralelo estudiados en la sección 2.1. En relación con la figura 2.5, el modelo incremental aplica secuencias lineales en forma escalonada a medida que avanza el calendario de actividades. Cada secuencia lineal produce “incrementos” de software susceptibles de entregarse [McD93] de manera parecida a los incrementos producidos en un flujo de proceso evolutivo (sección 2.3.3).

Por ejemplo, un software para procesar textos que se elabore con el paradigma incremental quizá entregue en el primer incremento las funciones básicas de administración de archivos, edición y producción del documento; en el segundo dará herramientas más sofisticadas de edición y producción de documentos; en el tercero habrá separación de palabras y revisión de la ortografía; y en el cuarto se proporcionará la capacidad para dar formato avanzado a las páginas. Debe observarse que el flujo de proceso para cualquier incremento puede incorporar el paradigma del prototipo.

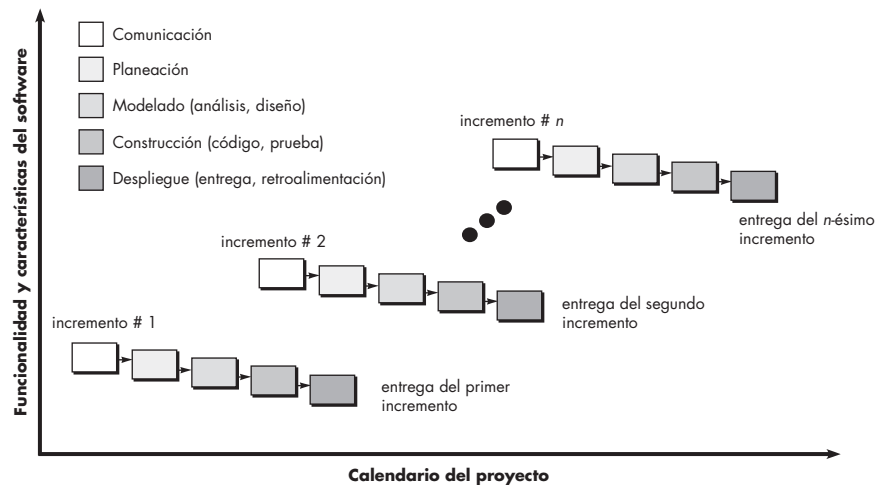
Cuando se utiliza un modelo incremental, es frecuente que el primer incremento sea el *producto fundamental*. Es decir, se abordan los requerimientos básicos, pero no se proporcionan muchas características suplementarias (algunas conocidas y otras no). El cliente usa el producto fundamental (o lo somete a una evaluación detallada). Como resultado del uso y/o evaluación,

#### PUNTO CLAVE

El modelo incremental ejecuta una serie de avances, llamados incrementos, que en forma progresiva dan más funcionalidad al cliente conforme se le entrega cada incremento.

#### CONSEJO

Su cliente solicita la entrega para una fecha que es imposible de cumplir. Sugiera entregar uno o más incrementos en la fecha que pide, y el resto del software (incrementos adicionales) en un momento posterior.

**FIGURA 2.5****El modelo incremental**

se desarrolla un plan para el incremento que sigue. El plan incluye la modificación del producto fundamental para cumplir mejor las necesidades del cliente, así como la entrega de características adicionales y más funcionalidad. Este proceso se repite después de entregar cada incremento, hasta terminar el producto final.

El modelo de proceso incremental se centra en que en cada incremento se entrega un producto que ya opera. Los primeros incrementos son versiones desnudas del producto final, pero proporcionan capacidad que sirve al usuario y también le dan una plataforma de evaluación.<sup>8</sup>

El desarrollo incremental es útil en particular cuando no se dispone de personal para la implementación completa del proyecto en el plazo establecido por el negocio. Los primeros incrementos se desarrollan con pocos trabajadores. Si el producto básico es bien recibido, entonces se agrega más personal (si se requiere) para que labore en el siguiente incremento. Además, los incrementos se planean para administrar riesgos técnicos. Por ejemplo, un sistema grande tal vez requiera que se disponga de hardware nuevo que se encuentre en desarrollo y cuya fecha de entrega sea incierta. En este caso, tal vez sea posible planear los primeros incrementos de forma que eviten el uso de dicho hardware, y así proporcionar una funcionalidad parcial a los usuarios finales sin un retraso importante.

### 2.3.3 Modelos de proceso evolutivo

El software, como todos los sistemas complejos, evoluciona en el tiempo. Es frecuente que los requerimientos del negocio y del producto cambien conforme avanza el desarrollo, lo que hace que no sea realista trazar una trayectoria rectilínea hacia el producto final; los plazos apretados del mercado hacen que sea imposible la terminación de un software perfecto, pero debe lanzarse una versión limitada a fin de aliviar la presión de la competencia o del negocio; se comprende bien el conjunto de requerimientos o el producto básico, pero los detalles del producto o extensiones del sistema aún están por definirse. En estas situaciones y otras parecidas se necesita un modelo de proceso diseñado explícitamente para adaptarse a un producto que evoluciona con el tiempo.

Los modelos evolutivos son iterativos. Se caracterizan por la manera en la que permiten desarrollar versiones cada vez más completas del software. En los párrafos que siguen se presentan dos modelos comunes de proceso evolutivo.

#### **PUNTO CLAVE**

El modelo del proceso evolutivo genera en cada iteración una versión final cada vez más completa del software.

<sup>8</sup> Es importante observar que para todos los modelos de proceso “ágiles” que se estudian en el capítulo 3 también se usa la filosofía incremental.

**Cita:**

"Planea para lanzar uno. De todos modos hará eso. Su única elección es si tratará de vender a sus clientes lo que lanzó."

Frederick P. Brooks



**CONSEJO**  
Cuando su cliente tiene una necesidad legítima, pero ignora los detalles, como primer paso desarrolle un prototipo.

**Hacer prototipos.** Es frecuente que un cliente defina un conjunto de objetivos generales para el software, pero que no identifique los requerimientos detallados para las funciones y características. En otros casos, el desarrollador tal vez no esté seguro de la eficiencia de un algoritmo, de la adaptabilidad de un sistema operativo o de la forma que debe adoptar la interacción entre el humano y la máquina. En estas situaciones, y muchas otras, el *paradigma de hacer prototipos* tal vez ofrezca el mejor enfoque.

Aunque es posible hacer prototipos como un modelo de proceso aislado, es más común usarlo como una técnica que puede implementarse en el contexto de cualquiera de los modelos de proceso descritos en este capítulo. Sin importar la manera en la que se aplique, el paradigma de hacer prototipos le ayudará a usted y a otros participantes a mejorar la comprensión de lo que hay que elaborar cuando los requerimientos no están claros.

El paradigma de hacer prototipos (véase la figura 2.6) comienza con comunicación. Usted se reúne con otros participantes para definir los objetivos generales del software, identifica cualesquiera requerimientos que conozca y detecta las áreas en las que es imprescindible una mayor definición. Se planea rápidamente una iteración para hacer el prototipo, y se lleva a cabo el modelado (en forma de un "diseño rápido"). Éste se centra en la representación de aquellos aspectos del software que serán visibles para los usuarios finales (por ejemplo, disposición de la interfaz humana o formatos de la pantalla de salida). El diseño rápido lleva a la construcción de un prototipo. Éste se entrega y es evaluado por los participantes, que dan retroalimentación para mejorar los requerimientos. La iteración ocurre a medida de que el prototipo es afinado para satisfacer las necesidades de distintos participantes, y al mismo tiempo le permite a usted entender mejor lo que se necesita hacer.

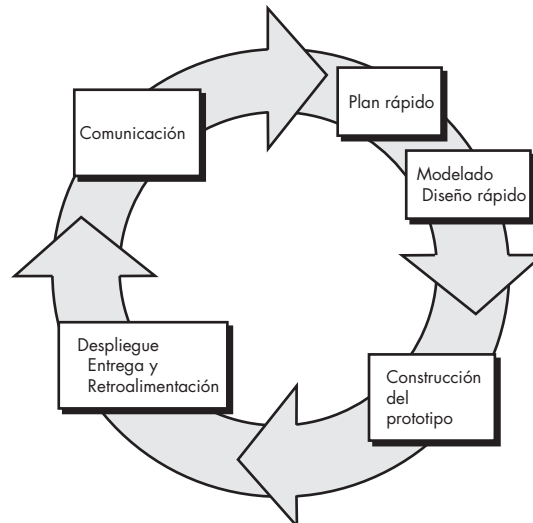
El ideal es que el prototipo sirva como mecanismo para identificar los requerimientos del software. Si va a construirse un prototipo, pueden utilizarse fragmentos de programas existentes o aplicar herramientas (por ejemplo, generadores de reportes y administradores de ventanas) que permitan generar rápidamente programas que funcionen.

Pero, ¿qué hacer con el prototipo cuando ya sirvió para el propósito descrito? Brooks [Bro95] da una respuesta:

En la mayoría de proyectos es raro que el primer sistema elaborado sea utilizable. Tal vez sea muy lento, muy grande, difícil de usar o todo a la vez. No hay más alternativa que comenzar de nuevo, con más inteligencia, y construir una versión rediseñada en la que se resuelvan los problemas.

**FIGURA 2.6**

**El paradigma de hacer prototipos**



El prototipo sirve como “el primer sistema”. Lo que Brooks recomienda es desecharlo. Pero esto quizá sea un punto de vista idealizado. Aunque algunos prototipos se construyen para ser “desechables”, otros son evolutivos; es decir, poco a poco se transforman en el sistema real.

Tanto a los participantes como a los ingenieros de software les gusta el paradigma de hacer prototipos. Los usuarios adquieren la sensación del sistema real, y los desarrolladores logran construir algo de inmediato. No obstante, hacer prototipos llega a ser problemático por las siguientes razones:



*Resista la presión para convertir un prototipo burdo en un producto terminado. Como resultado de ello, casi siempre disminuye la calidad.*

1. Los participantes ven lo que parece ser una versión funcional del software, sin darse cuenta de que el prototipo se obtuvo de manera caprichosa; no perciben que en la prisa por hacer que funcionara, usted no consideró la calidad general del software o la facilidad de darle mantenimiento a largo plazo. Cuando se les informa que el producto debe rehacerse a fin de obtener altos niveles de calidad, los participantes gritan que es usted un tonto y piden “unos cuantos arreglos” para hacer del prototipo un producto funcional. Con demasiada frecuencia, el gerente de desarrollo del software cede.
2. Como ingeniero de software, es frecuente que llegue a compromisos respecto de la implementación a fin de hacer que el prototipo funcione rápido. Quizá utilice un sistema operativo inapropiado, o un lenguaje de programación tan sólo porque cuenta con él y lo conoce; tal vez implementó un algoritmo ineficiente sólo para demostrar capacidad. Después de un tiempo, quizá se sienta cómodo con esas elecciones y olvide todas las razones por las que eran inadecuadas. La elección de algo menos que lo ideal ahora ha pasado a formar parte del sistema.

Aunque puede haber problemas, hacer prototipos es un paradigma eficaz para la ingeniería de software. La clave es definir desde el principio las reglas del juego; es decir, todos los participantes deben estar de acuerdo en que el prototipo sirva como el mecanismo para definir los requerimientos. Después se descartará (al menos en parte) y se hará la ingeniería del software real con la mirada puesta en la calidad.

## CASA SEGURA



### Selección de un modelo de proceso, parte 1

**La escena:** Sala de juntas del grupo de ingeniería de software de CPI Corporation, compañía (ficticia) que manufactura artículos de consumo para el hogar y para uso comercial.

**Participantes:** Lee Warren, gerente de ingeniería; Doug Miller, gerente de ingeniería de software; Jamie Lazar, miembro del equipo de software; Vinod Raman, miembro del equipo de software; y Ed Robbins, miembro del equipo de software.

#### La conversación:

**Lee:** Recapitulemos. He dedicado algún tiempo al análisis de la línea de productos CasaSegura, según la vemos hasta el momento. No hay duda de que hemos efectuado mucho trabajo tan sólo para definir el concepto, pero me gustaría que ustedes comenzaran a pensar en cómo van a enfocar la parte del software de este proyecto.

**Doug:** Pareciera que en el pasado hemos estado muy desorganizados en nuestro enfoque del software.

**Ed:** No sé, Doug, siempre sacamos el producto.

**Doug:** Es cierto, pero no sin muchos sobresaltos, y este proyecto parece más grande y complejo que cualquier cosa que hayamos hecho antes.

**Jamie:** No parece tan mal, pero estoy de acuerdo... nuestro enfoque *ad hoc* de los proyectos anteriores no funcionará en éste, en particular si tenemos una fecha de entrega muy apretada.

**Doug (sonríe):** Quiero ser un poco más profesional en nuestro enfoque. La semana pasada asistí a un curso breve y aprendí mucho sobre ingeniería de software... algo bueno. Aquí necesitamos un proceso.

**Jamie (con el ceño fruncido):** Mi trabajo es producir programas de computadora, no papel.

**Doug:** Den una oportunidad antes de ser tan negativos conmigo. Lo que quiero decir es esto: [Doug pasa a describir la estructura del proceso vista en este capítulo y los modelos de proceso prescriptivo presentados hasta el momento.]

**Doug:** De cualquier forma, parece que un modelo lineal no es para nosotros... pues supone que conocemos todos los requerimientos y, conociendo esta empresa, eso no parece probable.

**Vinod:** Sí, y parece demasiado orientado a las tecnologías de información... tal vez sea bueno para hacer un sistema de control de inventarios o algo así, pero no parece bueno para *CasaSegura*.

**Doug:** Estoy de acuerdo.

**Ed:** Ese enfoque de hacer prototipos parece bueno. En todo caso, se asemeja mucho a lo que hacemos aquí.

**Vinod:** Eso es un problema. Me preocupa que no nos dé suficiente estructura.

**Doug:** No te preocupes. Tenemos muchas opciones más, y quisiera que ustedes, muchachos, elijan la que sea mejor para el equipo y para el proyecto.

**El modelo espiral.** Propuesto en primer lugar por Barry Boehm [Boe88], el *modelo espiral* es un modelo evolutivo del proceso del software y se acopla con la naturaleza iterativa de hacer prototipos con los aspectos controlados y sistémicos del modelo de cascada. Tiene el potencial para hacer un desarrollo rápido de versiones cada vez más completas. Boehm [Boe01a] describe el modelo del modo siguiente:

El modelo de desarrollo espiral es un generador de *modelo de proceso* impulsado por el *riesgo*, que se usa para guiar la ingeniería concurrente con participantes múltiples de sistemas intensivos en software. Tiene dos características distintivas principales. La primera es el enfoque *cíclico* para el crecimiento incremental del grado de definición de un sistema y su implementación, mientras que disminuye su grado de riesgo. La otra es un conjunto de *puntos de referencia de anclaje puntual* para asegurar el compromiso del participante con soluciones factibles y mutuamente satisfactorias.

Con el empleo del modelo espiral, el software se desarrolla en una serie de entregas evolutivas. Durante las primeras iteraciones, lo que se entrega puede ser un modelo o prototipo. En las iteraciones posteriores se producen versiones cada vez más completas del sistema cuya ingeniería se está haciendo.

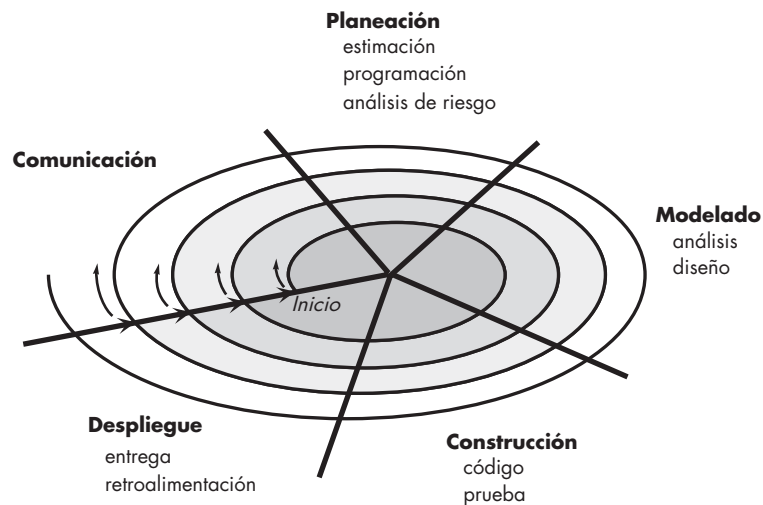
Un modelo en espiral es dividido por el equipo de software en un conjunto de actividades estructurales. Para fines ilustrativos, se utilizan las actividades estructurales generales ya analizadas.<sup>9</sup> Cada una de ellas representa un segmento de la trayectoria espiral ilustrada en la figura 2.7. Al comenzar el proceso evolutivo, el equipo de software realiza actividades implícitas en un

### PUNTO CLAVE

El modelo en espiral se adapta para emplearse a lo largo de todo el ciclo de vida de una aplicación, desde el desarrollo del concepto hasta el mantenimiento.

**FIGURA 2.7**

**Modelo de espiral común**



<sup>9</sup> El modelo espiral estudiado en esta sección es una variante del propuesto por Boehm. Para más información acerca del modelo espiral original, consulte [Boe88]. En [Boe98] se encuentra un análisis más reciente del modelo espiral del mismo autor.

circuito alrededor de la espiral en el sentido horario, partiendo del centro. El riesgo se considera conforme se desarrolla cada revolución (capítulo 28). En cada paso evolutivo se marcan *puntos de referencia puntuales*: combinación de productos del trabajo y condiciones que se encuentran a lo largo de la trayectoria de la espiral.

El primer circuito alrededor de la espiral da como resultado el desarrollo de una especificación del producto; las vueltas sucesivas se usan para desarrollar un prototipo y, luego, versiones cada vez más sofisticadas del software. Cada paso por la región de planeación da como resultado ajustes en el plan del proyecto. El costo y la programación de actividades se ajustan con base en la retroalimentación obtenida del cliente después de la entrega. Además, el gerente del proyecto ajusta el número planeado de iteraciones que se requieren para terminar el software.

A diferencia de otros modelos del proceso que finalizan cuando se entrega el software, el modelo espiral puede adaptarse para aplicarse a lo largo de toda la vida del software de cómputo. Entonces, el primer circuito alrededor de la espiral quizá represente un “proyecto de desarrollo del concepto” que comienza en el centro de la espiral y continúa por iteraciones múltiples<sup>10</sup> hasta que queda terminado el desarrollo del concepto. Si el concepto va a desarrollarse en un producto real, el proceso sigue hacia fuera de la espiral y comienza un “proyecto de desarrollo de producto nuevo”. El nuevo producto evolucionará a través de cierto número de iteraciones alrededor de la espiral. Más adelante puede usarse un circuito alrededor de la espiral para que represente un “proyecto de mejora del producto”. En esencia, la espiral, cuando se caracteriza de este modo, sigue operativa hasta que el software se retira. Hay ocasiones en las que el proceso está inmóvil, pero siempre que se inicia un cambio comienza en el punto de entrada apropiado (por ejemplo, mejora del producto).

El modelo espiral es un enfoque realista para el desarrollo de sistemas y de software a gran escala. Como el software evoluciona a medida que el proceso avanza, el desarrollador y cliente comprenden y reaccionan mejor ante los riesgos en cada nivel de evolución. El modelo espiral usa los prototipos como mecanismo de reducción de riesgos, pero, más importante, permite aplicar el enfoque de hacer prototipos en cualquier etapa de la evolución del producto. Mantiene el enfoque de escalón sistemático sugerido por el ciclo de vida clásico, pero lo incorpora en una estructura iterativa que refleja al mundo real en una forma más realista. El modelo espiral demanda una consideración directa de los riesgos técnicos en todas las etapas del proyecto y, si se aplica de manera apropiada, debe reducir los riesgos antes de que se vuelvan un problema.

Pero, como otros paradigmas, el modelo espiral no es una panacea. Es difícil convencer a los clientes (en particular en situaciones bajo contrato) de que el enfoque evolutivo es controlable. Demanda mucha experiencia en la evaluación del riesgo y se basa en ella para llegar al éxito. No hay duda de que habrá problemas si algún riesgo importante no se descubre y administra.

### 2.3.4 Modelos concurrentes

El *modelo de desarrollo concurrente*, en ocasiones llamado *ingeniería concurrente*, permite que un equipo de software represente elementos iterativos y concurrentes de cualquiera de los modelos de proceso descritos en este capítulo. Por ejemplo, la actividad de modelado definida para el modelo espiral se logra por medio de invocar una o más de las siguientes acciones de software: hacer prototipos, análisis y diseño.<sup>11</sup>

La figura 2.8 muestra la representación esquemática de una actividad de ingeniería de software dentro de la actividad de modelado con el uso del enfoque de modelado concurrente. La

#### WebRef

En la dirección [www.sei.cmu.edu/publications/documents/00.reports/00sr008.html](http://www.sei.cmu.edu/publications/documents/00.reports/00sr008.html) se encuentra información útil sobre el modelo espiral.



*Si su administración pide un desarrollo apegado al presupuesto (mala idea, por lo general), la espiral se convierte en un problema. El costo se revisa y modifica cada vez que se termina un circuito.*

#### Cita:

*“Sólo voy aquí y sólo el mañana me guía.”*

**Dave Matthews Band**



*Con frecuencia, el modelo concurrente es más apropiado para proyectos de ingeniería de productos en los que se involucran varios equipos de trabajo.*

10 Las flechas que apuntan hacia dentro a lo largo del eje que separa la región del **despliegue** de la de **comunicación** indican un potencial para la iteración local a lo largo de la misma trayectoria espiral.

11 Debe observarse que el análisis y diseño son tareas complejas que requieren mucho análisis. La parte 2 de este libro considera en detalle dichos temas.

## CASA SEGURA



## Selección de un modelo de proceso, parte 2

**La escena:** Sala de juntas del grupo de ingeniería de software de CPI Corporation, compañía que manufactura productos de consumo para uso doméstico y comercial.

**Participantes:** Lee Warren, gerente de ingeniería; Doug Miller, gerente de ingeniería de software; Vinod y Jamie, miembros del equipo de ingeniería de software.

**La conversación:** [Doug describe las opciones de proceso evolutivo.]

**Jamie:** Ahora me doy cuenta de algo. El enfoque incremental tiene sentido, y en verdad me gusta el flujo del modelo en espiral. Es bastante realista.

**Vinod:** De acuerdo. Entregamos un incremento, aprendemos de la retroalimentación del cliente, volvemos a planear y luego entregamos

otro incremento. También se ajusta a la naturaleza del producto. Podemos lanzar con rapidez algo al mercado y luego agregar funcionalidad con cada versión, digo... con cada incremento.

**Lee:** Un momento. Doug, ¿dijiste que volveríamos a hacer el plan a cada vuelta de la espiral? Eso no es nada bueno; necesitamos un plan, un programa de actividades y apegarnos a ellos.

**Doug:** Ésa es la vieja escuela, Lee. Como dijeron los chicos, tenemos que hacerlo apegado a la realidad. Afirmo que es mejor afinar el plan a medida de que aprendamos más y conforme se soliciten cambios. Eso es más realista. ¿Qué sentido tiene un plan si no refleja la realidad?

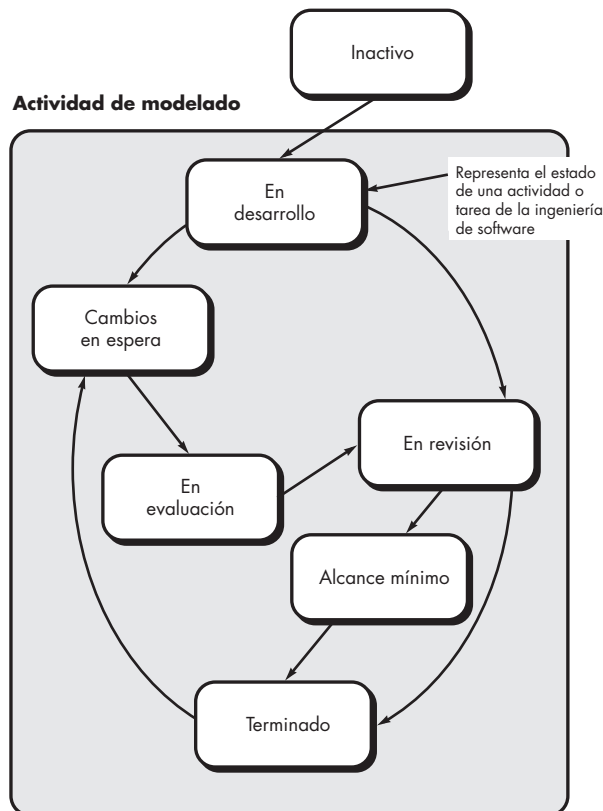
**Lee (con el ceño fruncido):** Supongo, pero... a la alta dirección no le va a gustar... quieren un plan fijo.

**Doug (sonriendo):** Entonces tendrás que reeducarlos, amigo.

actividad —**modelado**— puede estar en cualquiera de los estados<sup>12</sup> mencionados en un momento dado. En forma similar, es posible representar de manera análoga otras actividades, acciones o tareas (por ejemplo, **comunicación** o **construcción**). Todas las actividades de ingeniería de software existen de manera concurrente, pero se hallan en diferentes estados.

FIGURA 2.8

Un elemento del modelo de proceso concurrente



<sup>12</sup> Un *estado* es algún modo de comportamiento observable externamente.

Por ejemplo, la actividad de comunicación (no se muestra en la figura) termina su primera iteración al principio de un proyecto y existe en el estado de **cambios en espera**. La actividad de modelado (que existía en estado **inactivo** mientras concluía la comunicación inicial, ahora hace una transición al estado **en desarrollo**. Sin embargo, si el cliente indica que deben hacerse cambios en los requerimientos, la actividad de modelado pasa del estado **en desarrollo** al de **cambios en espera**.

El modelado concurrente define una serie de eventos que desencadenan transiciones de un estado a otro para cada una de las actividades, acciones o tareas de la ingeniería de software. Por ejemplo, durante las primeras etapas del diseño (acción importante de la ingeniería de software que ocurre durante la actividad de modelado), no se detecta una inconsistencia en el modelo de requerimientos. Esto genera el evento *corrección del modelo de análisis*, que disparará la acción de análisis de requerimientos del estado **terminado** al de **cambios en espera**.

El modelado concurrente es aplicable a todos los tipos de desarrollo de software y proporciona un panorama apropiado del estado actual del proyecto. En lugar de confinar las actividades, acciones y tareas de la ingeniería de software a una secuencia de eventos, define una red del proceso. Cada actividad, acción o tarea de la red existe simultáneamente con otras actividades, acciones o tareas. Los eventos generados en cierto punto de la red del proceso desencadenan transiciones entre los estados.

#### Cita:

"Todo proceso en su organización tiene un cliente, y un proceso sin cliente no tiene propósito."

V. Daniel Hunt

### 2.3.5 Una última palabra acerca de los procesos evolutivos

Ya se dijo que el software de cómputo moderno se caracteriza por el cambio continuo, por tiempos de entrega muy apretados y por una necesidad apremiante de la satisfacción del cliente o usuario. En muchos casos, el tiempo para llegar al mercado es el requerimiento administrativo más importante. Si se pierde un nicho de mercado, todo el proyecto de software podría carecer de sentido.<sup>13</sup>

Los modelos de proceso evolutivo fueron concebidos para cumplir esos requisitos, pero, aun así, como clase general de modelos de proceso tienen demasiadas debilidades, que fueron resumidas por Nogueira y sus colegas [Nog00]:

A pesar de los beneficios incuestionables de los procesos evolutivos de software, existen algunas preocupaciones. La primera es que hacer prototipos (y otros procesos evolutivos más sofisticados) plantea un problema para la planeación del proyecto debido a la incertidumbre en el número de ciclos que se requieren para elaborar el producto. La mayor parte de técnicas de administración y estimación de proyectos se basa en un planteamiento lineal de las actividades, por lo que no se ajustan por completo.

En segundo lugar, los procesos evolutivos de software no establecen la velocidad máxima de la evolución. Si las evoluciones ocurren demasiado rápido, sin un periodo de relajamiento, es seguro que el proceso se volverá un caos. Por otro lado, si la velocidad es muy lenta, se verá perjudicada la productividad...

En tercer lugar, los procesos de software deben centrarse en la flexibilidad y capacidad de extensión en lugar de en la alta calidad. Esto suena preocupante. Sin embargo, debe darse prioridad a la velocidad del desarrollo con el enfoque de cero defectos. Extender el desarrollo a fin de lograr alta calidad podría dar como resultado la entrega tardía del producto, cuando haya desaparecido el nicho de oportunidad. Este cambio de paradigma es impuesto por la competencia al borde del caos.

En realidad, sí parece preocupante un proceso del software que se centre en la flexibilidad, expansión y velocidad del desarrollo por encima de la calidad. No obstante, esta idea ha sido propuesta por varios expertos en ingeniería de software muy respetados ([You95], [Bac97]).

<sup>13</sup> Sin embargo, es importante notar que ser el primero en llegar al mercado no es garantía de éxito. En realidad, muchos productos de software muy exitosos han llegado en segundo o hasta en tercer lugar al mercado (aprendiendo de los errores de sus antecesores).

El objetivo de los modelos evolutivos es desarrollar software de alta calidad<sup>14</sup> en forma iterativa o incremental. Sin embargo, es posible usar un proceso evolutivo para hacer énfasis en la flexibilidad, expansibilidad y velocidad del desarrollo. El reto para los equipos de software y sus administradores es establecer un balance apropiado entre estos parámetros críticos del proyecto y el producto, y la satisfacción del cliente (árbitro definitivo de la calidad del software).

## 2.4 MODELOS DE PROCESO ESPECIALIZADO

Los modelos de proceso especializado tienen muchas de las características de uno o más de los modelos tradicionales que se presentaron en las secciones anteriores. Sin embargo, dichos modelos tienden a aplicarse cuando se elige un enfoque de ingeniería de software especializado o definido muy específicamente.<sup>15</sup>

### 2.4.1 Desarrollo basado en componentes

#### WebRef

En la dirección [www.cbd-hq.com](http://www.cbd-hq.com) hay información útil sobre el desarrollo basado en componentes.

Los componentes comerciales de software general (COTS, por sus siglas en inglés), desarrollados por vendedores que los ofrecen como productos, brindan una funcionalidad que se persigue con interfaces bien definidas que permiten que el componente se integre en el software que se va a construir. El *modelo de desarrollo basado en componentes* incorpora muchas de las características del modelo espiral. Es de naturaleza evolutiva [Nie92] y demanda un enfoque iterativo para la creación de software. Sin embargo, el modelo de desarrollo basado en componentes construye aplicaciones a partir de fragmentos de software prefabricados.

Las actividades de modelado y construcción comienzan con la identificación de candidatos de componentes. Éstos pueden diseñarse como módulos de software convencional o clases orientadas a objetos o paquetes<sup>16</sup> de clases. Sin importar la tecnología usada para crear los componentes, el modelo de desarrollo basado en componentes incorpora las etapas siguientes (se implementan con el uso de un enfoque evolutivo):

1. Se investigan y evalúan, para el tipo de aplicación de que se trate, productos disponibles basados en componentes.
2. Se consideran los aspectos de integración de los componentes.
3. Se diseña una arquitectura del software para que reciba los componentes.
4. Se integran los componentes en la arquitectura.
5. Se efectúan pruebas exhaustivas para asegurar la funcionalidad apropiada.

El modelo del desarrollo basado en componentes lleva a la reutilización del software, y eso da a los ingenieros de software varios beneficios en cuanto a la mensurabilidad. Si la reutilización de componentes se vuelve parte de la cultura, el equipo de ingeniería de software tiene la posibilidad tanto de reducir el ciclo de tiempo del desarrollo como el costo del proyecto. En el capítulo 10 se analiza con más detalle el desarrollo basado en componentes.

<sup>14</sup> En este contexto, la calidad del software se define con mucha amplitud para que agrupe no sólo la satisfacción del cliente sino también varios criterios técnicos que se estudian en los capítulos 14 y 16.

<sup>15</sup> En ciertos casos, los modelos de proceso especializado pueden caracterizarse mejor como un conjunto de técnicas o “metodología” para alcanzar una meta específica de desarrollo de software. No obstante, implican un proceso.

<sup>16</sup> En el apéndice 2 se estudian los conceptos orientados a objetos, y se utilizan en toda la parte 2 del libro. En este contexto, una clase agrupa un conjunto de datos y los procedimientos para procesarlos. Un paquete de clases es un conjunto de clases relacionadas que funcionan juntas para alcanzar cierto resultado final.

### 2.4.2 El modelo de métodos formales

El *modelo de métodos formales* agrupa actividades que llevan a la especificación matemática formal del software de cómputo. Los métodos formales permiten especificar, desarrollar y verificar un sistema basado en computadora por medio del empleo de una notación matemática rigurosa. Ciertas organizaciones de desarrollo de software aplican una variante de este enfoque, que se denomina *ingeniería de software de quirófano* [Mil87, Dye92].

Cuando durante el desarrollo se usan métodos formales (capítulo 21), se obtiene un mecanismo para eliminar muchos de los problemas difíciles de vencer con otros paradigmas de la ingeniería de software. Lo ambiguo, incompleto e inconsistente se descubre y corrige con más facilidad, no a través de una revisión *ad hoc* sino con la aplicación de análisis matemático. Si durante el diseño se emplean métodos formales, éstos sirven como base para la verificación del programa, y así permiten descubrir y corregir errores que de otro modo no serían detectados.

Aunque el modelo de los métodos formales no es el más seguido, promete un software libre de defectos. Sin embargo, se han expresado preocupaciones acerca de su aplicabilidad en un ambiente de negocios:

- El desarrollo de modelos formales consume mucho tiempo y es caro.
- Debido a que pocos desarrolladores de software tienen la formación necesaria para aplicar métodos formales, se requiere mucha capacitación.
- Es difícil utilizar los modelos como mecanismo de comunicación para clientes sin complejidad técnica.

A pesar de estas preocupaciones, el enfoque de los métodos formales ha ganado partidarios entre los desarrolladores que deben construir software de primera calidad en seguridad (por ejemplo, control electrónico de aeronaves y equipos médicos), y entre los desarrolladores que sufrirían graves pérdidas económicas si ocurrieran errores en su software.

### 2.4.3 Desarrollo de software orientado a aspectos

Sin importar el proceso del software que se elija, los constructores de software complejo implementan de manera invariable un conjunto de características, funciones y contenido de información localizados. Estas características localizadas del software se modelan como componentes (clases orientadas a objetos) y luego se construyen dentro del contexto de una arquitectura de sistemas. A medida que los sistemas modernos basados en computadora se hacen más sofisticados (y complejos), ciertas *preocupaciones* —propiedades que requiere el cliente o áreas de interés técnico— se extienden a toda la arquitectura. Algunas de ellas son las propiedades de alto nivel de un sistema (por ejemplo, seguridad y tolerancia a fallas). Otras afectan a funciones (aplicación de las reglas de negocios), mientras que otras más son sistémicas (sincronización de la tarea o administración de la memoria).

Cuando las preocupaciones afectan múltiples funciones, características e información del sistema, es frecuente que se les llame *preocupaciones globales*. Los *requerimientos del aspecto* definen aquellas preocupaciones globales que tienen algún efecto a través de la arquitectura del software. El *desarrollo de software orientado a aspectos* (DSOA), conocido también como *programación orientada a aspectos* (POA), es un paradigma de ingeniería de software relativamente nuevo que proporciona un proceso y enfoque metodológico para definir, especificar, diseñar y construir *aspectos*: “mecanismos más allá de subrutinas y herencia para localizar la expresión de una preocupación global” [Elr01].

Grundy [Gru02] analiza con más profundidad los aspectos en el contexto de lo que denomina *ingeniería de componentes orientada a aspectos* (ICOA):

La ICOA usa el concepto de rebanadas horizontales a través de componentes de software descompuestos verticalmente, llamados “aspectos”, para caracterizar las propiedades globales funcionales y



**Si con los métodos formales puede demostrarse lo correcto de un software, ¿por qué no son ampliamente utilizados?**

#### WebRef

Existen muchos recursos e información sobre SOA en la dirección: [aosd.net](http://aosd.net)



El DSOA define “aspectos” que expresan preocupaciones del cliente que afectan múltiples funciones, características e información del sistema.

no funcionales de los componentes. Los aspectos comunes y sistémicos incluyen interfaces de usuario, trabajo en colaboración, distribución, persistencia, administración de la memoria, procesamiento de las transacciones, seguridad, integridad, etc. Los componentes pueden proveer o requerir uno o más “detalles de aspectos” en relación con un aspecto particular, como un mecanismo de visión, alcance extensible y clase de interfaz (aspectos de la interfaz de usuario); generación de eventos, transporte y recepción (aspectos de distribución); almacenamiento, recuperación e indización de datos (aspectos de persistencia); autenticación, encriptación y derechos de acceso (aspectos de seguridad); descomposición de las transacciones, control de concurrencia y estrategia de registro (aspectos de las transacciones), entre otros. Cada detalle del aspecto tiene cierto número de propiedades relacionadas con las características funcionales o no del detalle del aspecto.

Aún no madura un proceso distinto orientado a aspectos. Sin embargo, es probable que un proceso así adopte características tanto de los modelos de proceso evolutivo como concurrente. El modelo evolutivo es apropiado en tanto los aspectos se identifican y después se construyen. La naturaleza paralela del desarrollo concurrente es esencial porque la ingeniería de aspectos se hace en forma independiente de los componentes de software localizados; aun así, los aspectos tienen un efecto directo sobre éstos. De esta forma, es esencial disponer de comunicación asincrónica entre las actividades de proceso del software aplicadas a la ingeniería, y la construcción de los aspectos y componentes.

El análisis detallado del desarrollo de software orientado al aspecto se deja a libros especializados en el tema. Si el lector tiene interés en profundizar, se le invita a consultar [Saf08], [Cla05], [Jac04] y [Gra03].

#### HERRAMIENTAS DE SOFTWARE



##### Administración del proceso

**Objetivo:** Ayudar a la definición, ejecución y administración de modelos de proceso prescriptivo.

**Mecánica:** Las herramientas de administración del proceso permiten que una organización o equipo de software defina un modelo completo del proceso (actividades estructurales, acciones, tareas, aseguramiento de la calidad, puntos de revisión, referencias y productos del trabajo). Además, las herramientas proporcionan un mapa conforme los ingenieros de software realizan el trabajo técnico, y una plantilla para los gerentes que deben dar seguimiento y controlar el proceso del software.

##### Herramientas representativas:<sup>17</sup>

*GDPA*, grupo de herramientas de investigación de definición del proceso, desarrollada por la Universidad de Bremen, en Alemania

([www.informatik.uni-bremen.de/uniform/gdpa/home.htm](http://www.informatik.uni-bremen.de/uniform/gdpa/home.htm)), proporciona una amplia variedad de funciones para modelar y administrar procesos.

*SpeedDev*, desarrollada por SpeedDev Corporation ([www.speedev.com](http://www.speedev.com)), incluye un conjunto de herramientas para la definición del proceso, administración de los requerimientos, resolución de problemas, y planeación y seguimiento del proyecto.

*ProVision BPMx*, desarrollado por Proforma ([www.proforma-corp.com](http://www.proforma-corp.com)), es representativo de muchas herramientas que ayudan a definir el proceso y que automatizan el flujo del trabajo.

En la dirección [www.processwave.net/Links/tool\\_links.htm](http://www.processwave.net/Links/tool_links.htm), se encuentra una lista extensa de muchas herramientas diferentes asociadas con el proceso del software.

## 2.5 EL PROCESO UNIFICADO

En su libro fundamental, *Unified Process*, Ivar Jacobson, Grady Booch y James Rumbaugh [Jac99] analizan la necesidad de un proceso del software “impulsado por el caso de uso, centrado en la arquitectura, iterativo e incremental”, con la afirmación siguiente:

<sup>17</sup> Las herramientas mencionadas aquí no representan una obligación; sólo son una muestra de las de esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus desarrolladores respectivos.

En la actualidad, la tendencia en el software es hacia sistemas más grandes y complejos. Eso se debe en parte al hecho de que año tras año las computadoras son más poderosas, lo que hace que los usuarios esperen más de ellas. Esta tendencia también se ha visto influida por el uso creciente de internet para intercambiar toda clase de información [...] Nuestro apetito por software cada vez más sofisticado aumenta conforme aprendemos, entre un lanzamiento y otro de un producto, cómo mejorar éste. Queremos software que se adapte mejor a nuestras necesidades, pero eso a su vez lo hace más complejo. En pocas palabras, queremos más.

En cierto modo, el proceso unificado es un intento por obtener los mejores rasgos y características de los modelos tradicionales del proceso del software, pero en forma que implemente muchos de los mejores principios del desarrollo ágil de software (véase el capítulo 3). El proceso unificado reconoce la importancia de la comunicación con el cliente y los métodos directos para describir su punto de vista respecto de un sistema (el caso de uso).<sup>18</sup> Hace énfasis en la importancia de la arquitectura del software y “ayuda a que el arquitecto se centre en las metas correctas, tales como que sea comprensible, permita cambios futuros y la reutilización” [Jac99]: Sugiere un flujo del proceso iterativo e incremental, lo que da la sensación evolutiva que resulta esencial en el desarrollo moderno del software.

### 2.5.1 Breve historia

Al principio de la década de 1990, James Rumbaugh [Rum91], Grady Booch [Boo94] e Ivar Jacobson [Jac92] comenzaron a trabajar en un “método unificado” que combinaría lo mejor de cada uno de sus métodos individuales de análisis y diseño orientado a objetos. El resultado fue un UML, *lenguaje de modelado unificado*, que contiene una notación robusta para el modelado y desarrollo de los sistemas orientados a objetos.

El UML se utiliza en toda la parte 2 del libro para representar tanto los modelos de requerimientos como el diseño. En el apéndice 1 se presenta un método introductorio a la enseñanza para quienes no están familiarizados con las reglas básicas de notación y modelado con el UML. El estudio exhaustivo del UML se deja a libros dedicados al tema. En el apéndice 1 se enlistan los textos recomendables.

El UML brinda la tecnología necesaria para apoyar la práctica de la ingeniería de software orientada a objetos, pero no da la estructura del proceso que guíe a los equipos del proyecto cuando aplican la tecnología. En los siguientes años, Jacobson, Rumbaugh y Booch desarrollaron el *proceso unificado*, estructura para la ingeniería de software orientado a objetos que utiliza UML. Actualmente, el proceso unificado (PU) y el UML se usan mucho en proyectos de toda clase orientados a objetos. El modelo iterativo e incremental propuesto por el PU puede y debe adaptarse para que satisfaga necesidades específicas del proyecto.

### 2.5.2 Fases del proceso unificado<sup>19</sup>

Al principio de este capítulo se estudiaron cinco actividades estructurales generales y se dijo que podían usarse para describir cualquier modelo de proceso del software. El proceso unificado no es la excepción. La figura 2.9 ilustra las “fases” del PU y las relaciona con las actividades generales estudiadas en el capítulo 1 y al inicio de éste.

La *fase de concepción* del PU agrupa actividades tanto de comunicación con el cliente como de planeación. Al colaborar con los participantes, se identifican los requerimientos del negocio,

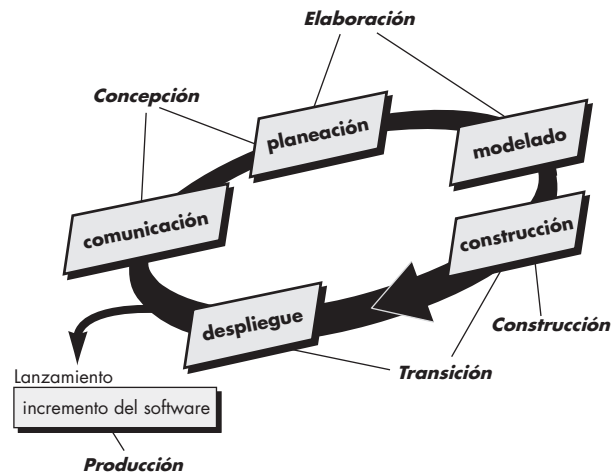


Las fases del PU tienen un objetivo similar al de las actividades estructurales generales definidas en este libro.

<sup>18</sup> El *caso de uso* (véase el capítulo 5) es la narración o plantilla que describe una función o rasgo de un sistema desde el punto de vista del usuario. Éste escribe un caso en uso que sirve como base para la creación de un modelo de requerimientos más completos.

<sup>19</sup> El proceso unificado en ocasiones recibe el nombre de *Proceso Racional Unificado* (PRU), acuñado por Rational Corporation (adquirida posteriormente por IBM), que contribuyó desde el principio al desarrollo y mejora del PU y a la elaboración de ambientes completos (herramientas y tecnología) que apoyan el proceso.

FIGURA 2.9

El proceso  
unificado

se propone una arquitectura aproximada para el sistema y se desarrolla un plan para la naturaleza iterativa e incremental del proyecto en cuestión. Los requerimientos fundamentales del negocio se describen por medio de un conjunto de casos de uso preliminares (véase el capítulo 5) que detallan las características y funciones que desea cada clase principal de usuarios. En este punto, la arquitectura no es más que un lineamiento tentativo de subsistemas principales y la función y rasgos que tienen. La arquitectura se mejorará después y se expandirá en un conjunto de modelos que representarán distintos puntos de vista del sistema. La planeación identifica los recursos, evalúa los riesgos principales, define un programa de actividades y establece una base para las fases que se van a aplicar a medida que avanza el incremento del software.

La *fase de elaboración* incluye las actividades de comunicación y modelado del modelo general del proceso (véase la figura 2.9). La elaboración mejora y amplía los casos de uso preliminares desarrollados como parte de la fase de concepción y aumenta la representación de la arquitectura para incluir cinco puntos de vista distintos del software: los modelos del caso de uso, de requerimientos, del diseño, de la implementación y del despliegue. En ciertos casos, la elaboración crea una “línea de base de la arquitectura ejecutable” [Arl02] que representa un sistema ejecutable de “primer corte”.<sup>20</sup> La línea de base de la arquitectura demuestra la viabilidad de ésta, pero no proporciona todas las características y funciones que se requieren para usar el sistema. Además, al terminar la fase de elaboración se revisa con cuidado el plan a fin de asegurar que el alcance, riesgos y fechas de entrega siguen siendo razonables. Es frecuente que en este momento se hagan modificaciones al plan.

La *fase de construcción* del PU es idéntica a la actividad de construcción definida para el proceso general del software. Con el uso del modelo de arquitectura como entrada, la fase de construcción desarrolla o adquiere los componentes del software que harán que cada caso de uso sea operativo para los usuarios finales. Para lograrlo, se completan los modelos de requerimientos y diseño que se comenzaron durante la fase de elaboración, a fin de que reflejen la versión final del incremento de software. Después se implementan en código fuente todas las características y funciones necesarias para el incremento de software (por ejemplo, el lanzamiento). A medida de que se implementan los componentes, se diseñan y efectúan pruebas unitarias<sup>21</sup> para cada uno. Además, se realizan actividades de integración (ensamble de compo-

**WebRef**

En la dirección [www.ambysoft.com/unifiedprocess/agileUP.html](http://www.ambysoft.com/unifiedprocess/agileUP.html), se encuentra un análisis interesante del PU en el contexto del desarrollo ágil.

<sup>20</sup> Es importante darse cuenta de que la línea de base de la arquitectura no es un prototipo y que no se desecha. Por el contrario, es revestida durante la fase siguiente del PU.

<sup>21</sup> En los capítulos 17 a 20 se presenta el análisis exhaustivo de las pruebas del software (incluso las *pruebas unitarias*).

nentes y pruebas de integración). Se emplean casos de uso para obtener un grupo de pruebas de aceptación que se ejecutan antes de comenzar la siguiente fase del PU.

La *fase de transición* del PU incluye las últimas etapas de la actividad general de construcción y la primera parte de la actividad de despliegue general (entrega y retroalimentación). Se da el software a los usuarios finales para las pruebas beta, quienes reportan tanto los defectos como los cambios necesarios. Además, el equipo de software genera la información de apoyo necesaria (por ejemplo, manuales de usuario, guías de solución de problemas, procedimientos de instalación, etc.) que se requiere para el lanzamiento. Al finalizar la fase de transición, el software incrementado se convierte en un producto utilizable que se lanza.

La *fase de producción* del PU coincide con la actividad de despliegue del proceso general. Durante esta fase, se vigila el uso que se da al software, se brinda apoyo para el ambiente de operación (infraestructura) y se reportan defectos y solicitudes de cambio para su evaluación.

Es probable que al mismo tiempo que se llevan a cabo las fases de construcción, transición y producción, comience el trabajo sobre el siguiente incremento del software. Esto significa que las cinco fases del PU no ocurren en secuencia sino que concurren en forma escalonada.

El flujo de trabajo de la ingeniería de software está distribuido a través de todas las fases del PU. En el contexto de éste, un *flujo de trabajo* es análogo al conjunto de tareas (que ya se describió en este capítulo). Es decir, un flujo de trabajo identifica las tareas necesarias para completar una acción importante de la ingeniería de software y los productos de trabajo que se generan como consecuencia de la terminación exitosa de aquéllas. Debe notarse que no toda tarea identificada para el flujo de trabajo del PU es realizada en todos los proyectos de software. El equipo adapta el proceso (acciones, tareas, subtareas y productos del trabajo) a fin de que cumpla sus necesidades.

## 2.6 MODELOS DEL PROCESO PERSONAL Y DEL EQUIPO

El mejor proceso del software es el que está cerca de las personas que harán el trabajo. Si un modelo del proceso del software se ha desarrollado en un nivel corporativo u organizacional, será eficaz sólo si acepta una adaptación significativa para que cubra las necesidades del equipo de proyecto que en realidad hace el trabajo de ingeniería de software. En la situación ideal se crearía un proceso que se ajustara del mejor modo a los requerimientos, y al mismo tiempo cubriera las más amplias necesidades del equipo y de la organización. En forma alternativa, el equipo crearía un proceso propio que satisficiera las necesidades más estrechas de los individuos y las más generales de la organización. Watts Humphrey ([Hum97] y [Hum00]) afirma que es posible crear un “proceso personal de software” y/o un “proceso del equipo de software”. Ambos requieren trabajo duro, capacitación y coordinación, pero los dos son asequibles.<sup>22</sup>

### Cita:

“La persona que es exitosa tan sólo se ha hecho el hábito de hacer las cosas que no hacen las personas que no tienen éxito.”

Dexter Yager

### WebRef

En la dirección [www.ipd.uka.de/](http://www.ipd.uka.de/) PSP, se hallan muchos recursos para el PPS.

### 2.6.1 Proceso personal del software (PPS)

Todo desarrollador utiliza algún proceso para elaborar software de cómputo. El proceso puede ser caprichoso o *ad hoc*; quizá cambie a diario; tal vez no sea eficiente, eficaz o incluso no sirva; pero sí existe un “proceso”. Watts Humphrey [Hum97] sugiere que a fin de cambiar un proceso personal ineficaz, un individuo debe pasar por las cuatro fases, cada una de las cuales requiere capacitación e instrumentación cuidadosa. El *proceso personal del software* (PPS) pone el énfasis en la medición personal tanto del producto del trabajo que se genera como de su calidad. Además, el PPS responsabiliza al profesional acerca de la planeación del proyecto (por ejemplo,

<sup>22</sup> Es útil notar que quienes proponen un desarrollo ágil del software (véase el capítulo 3) también plantean que el proceso debe ser cercano al equipo. Para lograr esto sugieren un método alternativo.

estimación y programación de actividades) y delega en el practicante el poder de controlar la calidad de todos los productos del trabajo de software que se desarrollen. El modelo del PPS define cinco actividades estructurales:

**? ¿Qué actividades estructurales se usan durante el PPS?**

**Planeación.** Esta actividad aísla los requerimientos y desarrolla las estimaciones tanto del tamaño como de los recursos. Además, realiza la estimación de los defectos (el número de defectos proyectados para el trabajo). Todas las mediciones se registran en hojas de trabajo o plantillas. Por último, se identifican las tareas de desarrollo y se crea un programa para el proyecto.

**Diseño de alto nivel.** Se desarrollan las especificaciones externas para cada componente que se va a construir y se crea el diseño de componentes. Si hay incertidumbre, se elaboran prototipos. Se registran todos los aspectos relevantes y se les da seguimiento.

**Revisión del diseño de alto nivel.** Se aplican métodos de verificación formal (véase el capítulo 21) para descubrir errores en el diseño. Se mantienen las mediciones para todas las tareas y resultados del trabajo importantes.

**Desarrollo.** Se mejora y revisa el diseño del componente. El código se genera, revisa, compila y prueba. Las mediciones se mantienen para todas las tareas y resultados de trabajo de importancia.

**Post mórtem.** Se determina la eficacia del proceso por medio de medidas y mediciones obtenidas (ésta es una cantidad sustancial de datos que deben analizarse con métodos estadísticos). Las medidas y mediciones deben dar la guía para modificar el proceso a fin de mejorar su eficacia.

### PUNTO CLAVE

El PPS pone el énfasis en la necesidad de registrar y analizar los tipos de errores que se cometen, de modo que se desarrollen estrategias para eliminarlos.

El PPS enfatiza la necesidad de detectar pronto los errores; de igual importancia es entender los tipos de ellos que es probable cometer. Esto se logra a través de una actividad de evaluación rigurosa ejecutada para todos los productos del trabajo que se generen.

El PPS representa un enfoque disciplinado basado en la medición para la ingeniería de software que quizá sea un choque cultural para muchos de sus practicantes. Sin embargo, cuando se introduce el PPS en forma apropiada en los ingenieros de software [Hum96], es significativa la mejora resultante en la productividad de la ingeniería respectiva y en la calidad del software [Fer97]. No obstante, el PPS no ha sido adoptado con amplitud por la industria. Es triste reconocer que las razones de esto tienen que ver más con la naturaleza humana y la inercia organizacional que con las fortalezas y debilidades del enfoque del PPS. Dicho enfoque plantea desafíos intelectuales y demanda un nivel de compromiso (por parte de los practicantes y sus administradores) que no siempre es posible obtener. La capacitación es relativamente larga y sus costos elevados. El nivel requerido de las mediciones es culturalmente difícil para muchas personas de la comunidad del software.

¿Es posible usar el PPS como un proceso eficaz de software a nivel personal? La respuesta es un rotundo “sí”. Pero aun si no se adoptara por completo el PPS, muchos de los conceptos del proceso de mejora personal que introduce constituyen un aprendizaje provechoso.

## 2.6.2 Proceso del equipo de software (PES)

Debido a que muchos proyectos de software industrial son elaborados por un equipo de profesionales, Watts Humphrey extendió las lecciones aprendidas de la introducción del PPS y propuso un *proceso del equipo de software* (PES). El objetivo de éste es construir un equipo “autodirigido” para el proyecto, que se organice para producir software de alta calidad. Humphrey [Hum98] define los objetivos siguientes para el PES:

- Formar equipos autodirigidos que planeen y den seguimiento a su trabajo, que establezcan metas y que sean dueños de sus procesos y planes. Éstos pueden ser equipos de software puros o de productos integrados (EPI) constituidos por 3 a 20 ingenieros.

### WebRef

En la dirección [www.sei.cmu.edu/tsp/](http://www.sei.cmu.edu/tsp/), hay información sobre la formación de equipos de alto rendimiento que usan PES y PPS.

- Mostrar a los gerentes cómo dirigir y motivar a sus equipos y cómo ayudarlos a mantener un rendimiento máximo.
- Acelerar la mejora del proceso del software, haciendo del modelo de madurez de la capacidad, CMM,<sup>23</sup> nivel 5, el comportamiento normal y esperado.
- Brindar a las organizaciones muy maduras una guía para la mejora.
- Facilitar la enseñanza universitaria de aptitudes de equipo con grado industrial.



Para formar un equipo autodirigido, usted debe colaborar bien en lo interno y comunicarse bien en lo externo.

Un equipo autodirigido tiene la comprensión consistente de sus metas y objetivos generales; define el papel y responsabilidad de cada miembro del equipo; da seguimiento cuantitativo a los datos del proyecto (sobre la productividad y calidad); identifica un proceso de equipo que sea apropiado para el proyecto y una estrategia para implementarlo; define estándares locales aplicables al trabajo de ingeniería de software del equipo; evalúa en forma continua el riesgo y reacciona en consecuencia; y da seguimiento, administra y reporta el estado del proyecto.

El PES define las siguientes actividades estructurales: **inicio del proyecto, diseño de alto nivel, implementación, integración y pruebas, y post mórtem**. Como sus contrapartes del PPS (observe que la terminología es algo diferente), estas actividades permiten que el equipo planee, diseñe y construya software en forma disciplinada, al mismo tiempo que mide cuantitativamente el proceso y el producto. La etapa post mórtem es el escenario de las mejoras del proceso.

El PES utiliza una variedad amplia de *scripts*, formatos y estándares que guían a los miembros del equipo en su trabajo. Los *scripts* definen actividades específicas del proceso (por ejemplo, inicio del proyecto, diseño, implementación, integración y pruebas del sistema, y post mórtem), así como otras funciones más detalladas del trabajo (planeación del desarrollo, desarrollo de requerimientos, administración de la configuración del software y prueba unitaria) que forman parte del proceso de equipo.

El PES reconoce que los mejores equipos de software son los autodirigidos.<sup>24</sup> Los miembros del equipo establecen los objetivos del proyecto, adaptan el proceso para que cubra las necesidades, controlan la programación de actividades del proyecto y, con la medida y análisis de las mediciones efectuadas, trabajan de manera continua en la mejora del enfoque de ingeniería de software que tiene el equipo.

Igual que el PPS, el PES es un enfoque riguroso para la ingeniería de software y proporciona beneficios distintivos y cuantificables en productividad y calidad. El equipo debe tener un compromiso total con el proceso y recibir capacitación completa para asegurar que el enfoque se aplique en forma apropiada.



Los scripts del PES definen elementos del proceso del equipo y de las actividades que ocurren dentro del proceso.

## 2.7 TECNOLOGÍA DEL PROCESO

El equipo del software debe adaptar uno o más de los modelos del proceso estudiados en las secciones precedentes. Para ello, se han desarrollado *herramientas de tecnología del proceso* que ayudan a las organizaciones de software a analizar su proceso actual, organizar las tareas de trabajo, controlar y vigilar el avance, y administrar la calidad técnica.

Las herramientas de tecnología del proceso permiten que una organización de software construya un modelo automatizado de la estructura del proceso, conjuntos de tareas y actividades sombrilla, estudiados en la sección 2.1. El modelo, que normalmente se representa como

<sup>23</sup> El modelo de madurez de la capacidad (CMM), que es una medida de la eficacia de un proceso del software, se estudia en el capítulo 30.

<sup>24</sup> En el capítulo 31 se analiza la importancia de los equipos “autoorganizados” como elemento clave del desarrollo ágil del software.

una red, se analiza para determinar el flujo de trabajo normal y se examinan estructuras alternativas del proceso que podrían llevar a disminuir el tiempo o costo del desarrollo.

Una vez creado un proceso aceptable, se emplean otras herramientas de tecnología para asignar, vigilar e incluso controlar todas las actividades, acciones y tareas de la ingeniería de software definidas como parte del modelo del proceso. Cada miembro de un equipo de software utiliza dichas herramientas para desarrollar una lista de verificación de las tareas de trabajo que deben realizarse. La herramienta de tecnología del proceso también se usa para coordinar el empleo de otras herramientas de la ingeniería de software que sean apropiadas para una tarea particular del trabajo.



### Herramientas de modelado del proceso

**Objetivo:** Si una organización trabaja para mejorar un proceso (o software) de negocios, primero debe entenderlo. Las herramientas de modelado del proceso (también llamadas herramientas de *tecnología del proceso* o de *administración del proceso*) se usan para representar los elementos clave de un proceso, de modo que se entienda mejor. Dichas herramientas también se relacionan con descripciones del proceso que ayudan a los involucrados a entender las acciones y tareas del trabajo que se requieren para llevarlo a cabo. Las herramientas de modelado del proceso tienen vínculos con otras que dan apoyo a las actividades del proceso definido.

**Mecánica:** Las herramientas en esta categoría permiten que un equipo defina los elementos de un modelo de proceso único (acciones, tareas, productos del trabajo, puntos de aseguramiento de la

calidad, etc.), dan una guía detallada acerca del contenido o descripción de cada elemento del proceso, y después administran el proceso conforme se realiza. En ciertos casos, las herramientas de tecnología del proceso incorporan tareas estándar de administración de proyectos, tales como estimación, programación, seguimiento y control.

#### Herramientas representativas:<sup>25</sup>

*Igrafx Process Tools:* herramientas que permiten que un equipo mapee, mida y modele el proceso del software ([www.micrografx.com](http://www.micrografx.com))

*Adeptia BPM Server:* diseñado para administrar, automatizar y optimizar procesos de negocios ([www.adptia.com](http://www.adptia.com))

*SpeedDev Suite:* conjunto de seis herramientas con mucho énfasis en las actividades de administración de la comunicación y modelado ([www.speeddev.com](http://www.speeddev.com))

## HERRAMIENTAS DE SOFTWARE

## 2.8 PRODUCTO Y PROCESO

Si el proceso es deficiente, no cabe duda de que el producto final sufrirá. Pero también es peligrosa la dependencia excesiva del proceso. En un ensayo corto escrito hace muchos años, Margaret Davis [Dav95a] hace comentarios atemporales sobre la dualidad del producto y del proceso:

Cada diez años, más o menos, la comunidad del software redefine “el problema” por medio de cambiar su atención de aspectos del producto a aspectos del proceso. Así, hemos adoptado lenguajes de programación estructurada (producto) seguidos de métodos de análisis estructurados (proceso) que van seguidos por el encapsulamiento de datos (producto) a los que siguieron el énfasis actual en el modelo de madurez de la capacidad, del Instituto de Ingeniería de Software para el Desarrollo de Software (proceso) (seguido por métodos orientados a objetos, a los que sigue el desarrollo ágil de software).

En tanto que la tendencia natural de un péndulo es alcanzar el estado de reposo en el punto medio entre dos extremos, la atención de la comunidad del software cambia constantemente porque se aplica una nueva fuerza al fallar la última oscilación. Estos vaivenes son dañinos en sí mismos porque confunden al profesional promedio del software al cambiar en forma radical lo que significa hacer el trabajo bien. Los cambios periódicos no resuelven “el problema” porque están predestinados a fallar toda vez que el producto y el proceso son tratados como si fueran una dicotomía en lugar de una dualidad.

<sup>25</sup> Las herramientas mencionadas aquí no son obligatorias, sino una muestra de las que hay en esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

En la comunidad científica existe el precedente de adoptar nociones de dualidad cuando las contradicciones en las observaciones no pueden ser explicadas por alguna teoría alternativa. La naturaleza dual de la luz, que parece ser al mismo tiempo onda y partícula, ha sido aceptada desde la década de 1920, cuando la propuso Louis de Broglie. Pienso que las observaciones que podemos hacer sobre el conjunto del software y su desarrollo demuestran una dualidad fundamental entre el producto y el proceso. Nunca es posible derivar u obtener todo el conjunto, su contexto, uso, significado y beneficios si se le ve sólo como proceso o sólo como producto...

Toda la actividad humana es un proceso, pero cada uno de nosotros obtiene un sentido de beneficio propio gracias a aquellas actividades que dan como resultado una representación o instancia que puede usar o apreciar más de una persona, utilizarla una y otra vez, o emplearla en algún otro contexto no considerado. Es decir, obtenemos sentimientos de satisfacción por la reutilización de nuestros productos, ya sea que lo hagamos nosotros u otras personas.

Entonces, si bien la rápida asimilación de las metas de reutilización en el desarrollo del software incrementa potencialmente la satisfacción que obtienen los profesionales del software en su trabajo, también aumenta la urgencia de la aceptación de la dualidad de producto y proceso. Pensar en un artefacto reutilizable como si fuera sólo un producto o sólo un proceso oscurece el contexto y las formas de emplearlo, o bien oculta el hecho de que cada uso da como resultado un producto que a su vez será utilizado como entrada para alguna otra actividad de desarrollo de software. Privilegiar un punto de vista sobre el otro reduce mucho las oportunidades para la reutilización y, por tanto, se pierde la oportunidad de aumentar la satisfacción por el trabajo.

La gente obtiene tanta (o más) satisfacción del proceso creativo como del producto final. Un artista disfruta las pinceladas tanto como el resultado que enmarca. Un escritor goza de la búsqueda de la metáfora apropiada tanto como del libro terminado. Como profesional creativo del software, usted también debe obtener tanta satisfacción del proceso como del producto final. La dualidad de producto y proceso es un elemento importante para hacer que personas creativas se involucren conforme la ingeniería de software evoluciona.

## 2.9 RESUMEN

Un modelo general del proceso para la ingeniería de software incluye un conjunto de actividades estructurales y sombrija, acciones y tareas de trabajo. Cada uno de los modelos de proceso puede describirse por un flujo distinto del proceso: descripción de cómo se organizan secuencial y cronológicamente las actividades estructurales, acciones y tareas. Los patrones del proceso pueden utilizarse para resolver los problemas comunes que surgen como parte del proceso del software.

Los modelos de proceso prescriptivo se han aplicado durante muchos años en un esfuerzo por introducir orden y estructura al desarrollo de software. Cada uno de dichos modelos sugiere un flujo de proceso algo distinto, pero todos llevan a cabo el mismo conjunto de actividades estructurales generales: comunicación, planeación, modelado, construcción y desarrollo.

Los modelos de proceso secuencial, como el de la cascada y en V, son los paradigmas más antiguos del software. Sugieren un flujo lineal del proceso que con frecuencia no es congruente con las realidades modernas (cambio continuo, sistemas en evolución, plazos ajustados, etc.) del mundo del software. Sin embargo, tienen aplicación en situaciones en las que los requerimientos están bien definidos y son estables.

Los modelos de proceso incremental son de naturaleza iterativa y producen con mucha rapidez versiones funcionales del software. Los modelos de proceso evolutivo reconocen la naturaleza iterativa e incremental de la mayoría de proyectos de ingeniería de software y están diseñados para aceptar los cambios. Los modelos evolutivos, tales como el de hacer prototipos y el espiral, generan rápido productos de trabajo incremental (o versiones funcionales del software). Estos modelos se adoptan para aplicarse a lo largo de todas las actividades de la inge-

niería de software, desde el desarrollo del concepto hasta el mantenimiento del sistema a largo plazo.

El modelo de proceso concurrente permite que un equipo de software represente los elementos iterativos y concurrentes de cualquier modelo de proceso. Los modelos especializados incluyen el basado en componentes, que pone el énfasis en la reutilización y ensamble de los componentes; el modelo de métodos formales consiste en un enfoque basado en matemáticas para desarrollar y verificar el software; y el modelo orientado a aspectos implica preocupaciones globales que afectan toda la arquitectura del sistema. **El proceso unificado es un proceso del software diseñado como estructura para los métodos y herramientas del UML, y está “impulsado por el caso de uso, centrado en la arquitectura, y es iterativo e incremental”.**

**Se han propuesto modelos personal y del equipo para el proceso del software. Ambos enfatizan la medición, planeación y autodirección como los ingredientes clave para un proceso exitoso del software.**

## PROBLEMAS Y PUNTOS POR EVALUAR

- 2.1.** En la introducción de este capítulo, Baetjer afirma que: “El proceso genera interacción entre usuarios y diseñadores, entre usuarios y herramientas cambiantes [tecnología].” Enliste cinco preguntas que a) los diseñadores deben responder a los usuarios, b) los usuarios deben plantear a los diseñadores, c) los usuarios deben hacerse a sí mismos sobre el producto de software que ha de elaborarse, d) los diseñadores deben plantearse acerca del producto de software que va a construirse y del proceso que se usará para ello.
- 2.2.** Trate de desarrollar un conjunto de acciones para la actividad de comunicación. Seleccione una acción y defina un conjunto de tareas para ella.
- 2.3.** Un problema común durante la **comunicación** ocurre cuando se encuentra a dos participantes que tienen ideas en conflicto sobre lo que debe ser el software, es decir, que tienen requerimientos mutuamente conflictivos. Desarrolle un patrón del proceso (esto sería un patrón de la etapa) con el empleo de la plantilla presentada en la sección 2.1.3 que aborda este problema y sugiera un enfoque eficaz para él.
- 2.4.** Investigue un poco sobre el PPS y haga una breve presentación que describa los tipos de mediciones que se pide hacer a un ingeniero individual de software y la forma en la que pueden usarse para mejorar la eficacia personal.
- 2.5.** El uso de scripts (mecanismo requerido en el PES) no es apreciado de manera universal en la comunidad del software. Haga una lista de pros y contras en relación con los scripts y sugiera al menos dos situaciones en las que serían útiles, y otras dos en las que generarían menos beneficios.
- 2.6.** Lea a [Nog00] y escriba un ensayo de dos o tres páginas donde analice el efecto que tiene el “caos” en la ingeniería de software.
- 2.7.** Dé tres ejemplos de proyectos de software que podrían efectuarse con el modelo de cascada. Sea específico.
- 2.8.** Proporcione tres ejemplos de proyectos de software que podrían abordarse con el modelo de hacer prototipos. Sea específico.
- 2.9.** ¿Qué adaptaciones del proceso se requerirían si el proyecto evolucionara en un sistema o producto que se entregase?
- 2.10.** Diga tres ejemplos de proyectos de software que podrían realizarse con el modelo incremental. Sea específico.
- 2.11.** Conforme avanza hacia fuera por el flujo de proceso en espiral, ¿qué puede decirse sobre el software que se está desarrollando o que está en mantenimiento?
- 2.12.** ¿Es posible combinar modelos de proceso? Si es así, diga un ejemplo.
- 2.13.** El modelo de proceso concurrente define un conjunto de “estados”. Describa con sus propias palabras qué es lo que representan, y después indique cómo entran en juego dentro del modelo de proceso concurrente.

- 2.14.** ¿Cuáles son las ventajas y desventajas de desarrollar software en el que la calidad no es “suficientemente buena”? Es decir, ¿qué pasa cuando se pone el énfasis en la velocidad de desarrollo sobre la calidad del producto?
- 2.15.** Dé tres ejemplos de proyectos de software que serían abordables con el modelo basado en componentes. Sea específico.
- 2.16.** ¿Es posible demostrar que un componente de software, o incluso un programa completo, es correcto? Entonces, ¿por qué no todos lo hacen?
- 2.17.** ¿Son lo mismo el proceso unificado y el UML? Explique su respuesta.

## LECTURAS ADICIONALES Y FUENTES DE INFORMACIÓN

La mayor parte de los libros de ingeniería de software consideran en detalle los modelos de proceso tradicionales. Libros como el de Sommerville (*Software Engineering*, 8a. ed., Addison-Wesley, 2006), Pfleeger y Atlee (*Software Engineering*, 3a. ed., Prentice-Hall, 2005), y Schach (*Object-Oriented and Classical Software Engineering*, 7a. ed., McGraw-Hill, 2006) consideran los paradigmas tradicionales y estudian sus fortalezas y debilidades. Glass (*Facts and Fallacies of Software Engineering*, Prentice-Hall, 2002) da un punto de vista pragmático y crudo del proceso de ingeniería de software. Aunque no se dedica específicamente al proceso, Brooks (*The Mythical Man-Month*, 2a. ed., Addison-Wesley, 1995) presenta la sabiduría antigua sobre los proyectos y plantea que todo tiene que ver con el proceso.

Firesmith y Henderson-Sellers (*The OPEN Process Framework: An Introduction*, Addison-Wesley, 2001) presenta una plantilla general para crear “procesos de software flexibles pero con disciplina” y analiza los atributos y objetivos del proceso. Madachy (*Software Process Dynamics*, Wiley-IEEE, 2008) estudia técnicas de modelado que permiten analizar los elementos técnicos y sociales interrelacionados del proceso del software. Sharpe y McDermott (*Workflow Modeling: Tools for Process Improvement and Application Development*, Artech House, 2001) presentan herramientas para modelar procesos tanto de software como de negocios.

Lim (*Managing Software Reuse*, Prentice-Hall, 2004) estudia la reutilización desde la perspectiva del gerente. Ezran, Morisio y Tully (*Practical Software Reuse*, Springer, 2002) y Jacobson, Griss y Jonsson (*Software Reuse*, Addison-Wesley, 1997) presentan mucha información útil sobre el desarrollo basado en componentes. Heineman y Council (*Component-Based Software Engineering*, Addison-Wesley, 2001) describen el proceso requerido para implementar sistemas basados en componentes. Kenett y Baker (*Software Process Quality: Management and Control*, Marcel Dekker, 1999) analizan la manera en la que se conectan íntimamente la administración de la calidad y el diseño del proceso.

Nygard (*Release It!: Design and Deploy Production-Ready Software*, Pragmatic Bookshelf, 2007) y Richardson y Gwaltney (*Ship it! A Practical Guide to Successful Software Projects*, Pragmatic Bookshelf, 2005) presentan una amplia colección de lineamientos útiles aplicables a la actividad de despliegue.

Además del libro fundamental de Jacobson, Rumbaugh y Booch acerca del proceso unificado [Jac99], los libros de Arlow y Neustadt (*UML 2 and the Unified Process*, Addison-Wesley, 2005), Kroll y Kruchten (*The Rational Unified Process Made Easy*, Addison-Wesley, 2003) y Farve (*UML and the Unified Process*, IRM Press, 2003) proveen información complementaria excelente. Gibbs (*Project Management with the IBM Rational Unified Process*, IBM Press, 2006) analiza la administración de proyectos dentro del contexto del PU.

En internet existe una amplia variedad de fuentes de información sobre la ingeniería de software y el proceso del software. En el sitio web del libro, [www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm](http://www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm), hay una lista actualizada de referencias en la Red Mundial que son relevantes para el proceso del software.