CAPÍTULO

4 Principios que guían la práctica

En un libro que explora las vidas y pensamientos de los ingenieros de software, Ellen Ullman [Ull97] ilustra una parte de su vida con el relato de lo que piensa un profesional del software cuando está bajo presión:

> No tengo idea de la hora que es. En esta oficina no hay ventanas ni reloj, sólo la pantalla de un horno de microondas que parpadea su LED de color rojo: 12:00, 12:00, 12:00. Joel y yo hemos estado programando durante varios días. Tenemos una falla, endemoniada y testaruda. Así que nos sentimos bien con el pulso rojo sin tiempo, como si fuera un pasmo de nuestros cerebros, de algún modo sincronizados al mismo ritmo del parpadeo...
>
> ¿En qué estamos trabajando? Los detalles se me escapan. Tal vez ayudamos a personas pobres y enfermas o mejoramos un conjunto de rutinas de bajo nivel de un protocolo de base de datos distribuida, no me importa. Debería importarme; en otra parte de mi ser —más tarde, quizá cuando salga de este cuarto lleno de computadoras— me preocuparé mucho de por qué y para quién y con qué propósito estoy escribiendo software. Pero ahora, no. He cruzado una membrana tras la que el mundo real y sus asuntos ya no importan. Soy ingeniera de software.

La anterior es una imagen tenebrosa de la práctica de la ingeniería de software, pero si se detienen un poco a pensarlo, muchos de los lectores de este libro se verán reflejados en ella.

Las personas que elaboran software de cómputo practican el arte, artesanía o disciplina[1] conocida como ingeniería de software. Pero, ¿qué es la "práctica" de la ingeniería de software? En un sentido general, es un conjunto de conceptos, principios, métodos y herramientas a los que un ingeniero de software recurre en forma cotidiana. La práctica permite que los gerentes

Una mirada rápida

¿Qué es? La práctica de la ingeniería de software es un conjunto amplio de principios, conceptos, métodos y herramientas que deben considerarse al planear y desarrollar software.

¿Quién lo hace? Los profesionales (ingenieros de software) y sus gerentes realizan varias tareas de ingeniería de software.

¿Por qué es importante? El proceso de software proporciona a todos los involucrados en la creación de un sistema o producto basado en computadora un mapa para llegar con éxito al destino. La práctica proporciona los detalles que se necesitarán para circular por la carretera. Indica dónde se localizan los puentes, los caminos cerrados y las bifurcaciones. Ayuda a entender los conceptos y principios que deben entenderse y seguirse a fin de llegar con seguridad y rapidez. Enseña a manejar, dónde disminuir la velocidad y en qué lugares acelerar. En el contexto de la ingeniería de software, la práctica es lo que se hace día tras día conforme el software evoluciona de idea a realidad.

¿Cuáles son los pasos? Son tres los elementos de la práctica que se aplican sin importar el modelo de proceso que se elija. Se trata de: principios, conceptos y métodos. Un cuarto elemento de la práctica —las herramientas— da apoyo a la aplicación de los métodos.

¿Cuál es el producto final? La práctica incluye las actividades técnicas que generan todos los productos del trabajo definidos por el modelo del proceso de software que se haya escogido.

¿Cómo me aseguro de que lo hice bien? En primer lugar, hay que tener una comprensión sólida de los principios que se aplican al trabajo (por ejemplo, el diseño) en cuestión. Después, asegúrese de que se escogió el método apropiado para el trabajo, use herramientas automatizadas cuando sean adecuadas para la tarea y sea firme respecto de la necesidad de técnicas de aseguramiento de la calidad de los productos finales que se generen.

1 Algunos escritores afirman que cualquiera de estos términos excluye a los otros. En realidad, la ingeniería de software es las tres cosas.

administren proyectos de software y que los ingenieros de software elaboren programas de cómputo. La práctica da al modelo del proceso de software el saber técnico y administrativo para realizar el trabajo. La práctica transforma un enfoque caprichoso y disperso en algo más organizado, más eficaz y con mayor probabilidad de alcanzar el éxito.

A lo largo de lo que resta del libro se estudiarán distintos aspectos de la práctica de la ingeniería de software. En este capítulo, la atención se pone en los principios y conceptos que la guían en lo general.

4.1 Conocimiento de la ingeniería de software

En un editorial publicado hace diez años en *IEEE Software*, Steve McConnell [McC99] hizo el siguiente comentario:

> Muchos trabajadores del software piensan que el conocimiento de la ingeniería de software casi exclusivamente consiste en tecnologías específicas: Java, Perl, html, C++, Linux, Windows NT, etc. Para programar computadoras es necesario conocer los detalles tecnológicos específicos. Si alguien pide al lector que escriba un programa en C++, tiene que saber algo sobre este lenguaje a fin de que el programa funcione.
>
> Es frecuente escuchar que el conocimiento del desarrollo de software tiene una vida media de tres años, lo que significa que la mitad de lo que es necesario saber hoy será obsoleto dentro de tres años. En el dominio del conocimiento relacionado con la tecnología es probable que eso se cumpla. Pero hay otra clase de conocimiento de desarrollo de software —algo que el autor considera como los "principios de la ingeniería de software"— que no tiene una vida media de tres años. Es factible que dichos principios sirvan al programador profesional durante toda su carrera.

McConnell continúa y plantea que el cuerpo de conocimientos de la ingeniería de software (alrededor del año 2000) ha evolucionado para convertirse en un "núcleo estable" que representa cerca de "75% del conocimiento necesario para desarrollar un sistema complejo". Pero, ¿qué es lo que hay dentro de ese núcleo estable?

Como dice McConnell, los principios fundamentales —ideas elementales que guían a los ingenieros de software en el trabajo que realizan— dan ahora un fundamento a partir del cual pueden aplicarse y evaluarse los modelos, métodos y herramientas de ingeniería.

4.2 Principios fundamentales

Cita:

"En teoría no hay diferencia entre la teoría y la práctica. Pero en la práctica sí la hay."

Jan van de Snepscheut

La práctica de la ingeniería de software está guiada por un conjunto de principios fundamentales que ayudan en la aplicación del proceso de software significativo y en la ejecución de métodos eficaces de ingeniería de software. En el nivel del proceso, los principios fundamentales establecen un fundamento filosófico que guía al equipo de software cuando realiza actividades estructurales y actividades sombrilla, cuando navega por el flujo del proceso y elabora un conjunto de productos del trabajo de la ingeniería de software. En el nivel de la práctica, los principios fundamentales definen un conjunto de valores y reglas que sirven como guía cuando se analiza un problema, se diseña una solución, se implementa y prueba ésta y cuando, al final, se entrega el software a la comunidad de usuarios.

En el capítulo 1 se identificó un conjunto de principios generales que amplían el proceso y práctica de la ingeniería de software: 1) agregar valor para los usuarios finales, 2) mantenerlo sencillo, 3) fijar la visión (del producto y el proyecto), 4) reconocer que otros consumen (y deben entender) lo que usted produce, 5) abrirse al futuro, 6) planear la reutilización y 7) ¡pensar! Aunque estos principios generales son importantes, se caracterizan en un nivel tan alto de abstracción que en ocasiones son difíciles de traducir en la práctica cotidiana de la ingeniería de

software. En las subsecciones que siguen se analizan con más detalle los principios fundamentales que guían el proceso y la práctica.

4.2.1 Principios que guían el proceso

En la parte 1 de este libro se estudia la importancia del proceso de software y se describen los abundantes modelos de proceso que se han propuesto para hacer el trabajo de ingeniería de software. Sin que importe que un modelo sea lineal o iterativo, prescriptivo o ágil, puede caracterizarse con el empleo de la estructura general del proceso aplicable a todos los modelos de proceso. Los siguientes principios fundamentales se aplican a la estructura y, por extensión, a todo proceso de software:

Todo proyecto y equipo son únicos. Esto significa que debe adaptar el proceso para que se ajuste mejor a sus necesidades.

Principio 1. ***Ser ágil.*** Ya sea que el modelo de proceso que se elija sea prescriptivo o ágil, son los principios básicos del desarrollo ágil los que deben gobernar el enfoque. Todo aspecto del trabajo que se haga debe poner el énfasis en la economía de acción: en mantener el enfoque técnico tan sencillo como sea posible, hacer los productos del trabajo que se generan tan concisos como se pueda y tomar las decisiones localmente, siempre que sea posible.

Principio 2. ***En cada etapa, centrarse en la calidad.*** La condición de salida para toda actividad, acción y tarea del proceso debe centrarse en la calidad del producto del trabajo que se ha generado.

Principio 3. ***Estar listo para adaptar.*** El proceso no es una experiencia religiosa, en él no hay lugar para el dogma. Cuando sea necesario, adapte su enfoque a las restricciones impuestas por el problema, la gente y el proyecto en sí.

Principio 4. ***Formar un equipo eficaz.*** El proceso y práctica de la ingeniería de software son importantes, pero el objetivo son las personas. Forme un equipo con organización propia en el que haya confianza y respeto mutuos.

Principio 5. ***Establecer mecanismos para la comunicación y coordinación.*** Los proyectos fallan porque la información importante cae en las grietas o porque los participantes no coordinan sus esfuerzos para crear un producto final exitoso. Éstos son aspectos de la administración que deben enfrentarse.

Cita:

"La verdad es que siempre se sabe lo que es correcto hacer. La parte difícil es hacerlo."

General H. Norman Schwarzkopf

Principio 6. ***Administrar el cambio.*** El enfoque puede ser formal o informal, pero deben establecerse mecanismos para administrar la forma en la que los cambios se solicitan, evalúan, aprueban e implementan.

Principio 7. ***Evaluar el riesgo.*** Son muchas las cosas que pueden salir mal cuando se desarrolla software. Es esencial establecer planes de contingencia.

Principio 8. ***Crear productos del trabajo que agreguen valor para otros.*** Sólo genere aquellos productos del trabajo que agreguen valor para otras actividades, acciones o tareas del proceso. Todo producto del trabajo que se genere como parte de la práctica de ingeniería de software pasará a alguien más. La lista de las funciones y características requeridas se dará a la persona (o personas) que desarrollará(n) un diseño, el diseño pasará a quienes generan código y así sucesivamente. Asegúrese de que el producto del trabajo imparte la información necesaria sin ambigüedades u omisiones.

La parte 4 de este libro se centra en aspectos de la administración del proyecto y del proceso, y analiza en detalle varios aspectos de cada uno de dichos principios.

4.2.2 Principios que guían la práctica

La práctica de la ingeniería de software tiene un solo objetivo general: entregar a tiempo software operativo de alta calidad que contenga funciones y características que satisfagan las ne-

cesidades de todos los participantes. Para lograrlo, debe adoptarse un conjunto de principios fundamentales que guíen el trabajo técnico. Estos principios tienen mérito sin que importen los métodos de análisis y diseño que se apliquen, ni las técnicas de construcción (por ejemplo, el lenguaje de programación o las herramientas automatizadas) que se usen o el enfoque de verificación y validación que se elija. Los siguientes principios fundamentales son vitales para la práctica de la ingeniería de software:

Los problemas son más fáciles de resolver cuando se subdividen en entidades separadas, distintas entre sí, solucionables individualmente y verificables.

Principio 1. ***Divide y vencerás.*** Dicho en forma más técnica, el análisis y el diseño siempre deben enfatizar la *separación de entidades* (SdE). Un problema grande es más fácil de resolver si se divide en un conjunto de elementos (o *entidades*). Lo ideal es que cada entidad entregue funcionalidad distinta que pueda desarrollarse, y en ciertos casos validarse, independientemente de otras entidades.

Principio 2. ***Entender el uso de la abstracción.*** En su parte medular, una abstracción es una simplificación de algún elemento complejo de un sistema usado para comunicar significado en una sola frase. Cuando se usa la abstracción *hoja de cálculo*, se supone que se comprende lo que es una hoja de cálculo, la estructura general de contenido que presenta y las funciones comunes que se aplican a ella. En la práctica de la ingeniería de software, se usan muchos niveles diferentes de abstracción, cada uno de los cuales imparte o implica significado que debe comunicarse. En el trabajo de análisis y diseño, un equipo de software normalmente comienza con modelos que representan niveles elevados de abstracción (por ejemplo, una hoja de cálculo) y poco a poco los refina en niveles más bajos de abstracción (como una *columna* o la función *SUM*).

Joel Spolsky [Spo02] sugiere que "todas las abstracciones no triviales hasta cierto punto son esquivas". El objetivo de una abstracción es eliminar la necesidad de comunicar detalles. Pero, en ocasiones, los efectos problemáticos precipitados por estos detalles se "filtran" por todas partes. Sin la comprensión de los detalles, no puede diagnosticarse con facilidad la causa de un problema.

Principio 3. ***Buscar la coherencia.*** Ya sea que se esté creando un modelo de los requerimientos, se desarrolle un diseño de software, se genere código fuente o se elaboren casos de prueba, el principio de coherencia sugiere que un contexto familiar hace que el software sea más fácil de usar. Como ejemplo, considere el diseño de una interfaz de usuario para una *webapp*. La colocación consistente de opciones de menú, el uso de un esquema coherencia de color y el uso coherencia de íconos reconocibles ayudan a hacer que la interfaz sea muy buena en el aspecto ergonómico.

Principio 4. ***Centrarse en la transferencia de información.*** El software tiene que ver con la transferencia de información: de una base de datos a un usuario final, de un sistema heredado a una *webapp*, de un usuario final a una interfaz gráfica de usuario (GUI, por sus siglas en inglés), de un sistema operativo a una aplicación, de un componente de software a otro... la lista es casi interminable. En todos los casos, la información fluye a través de una interfaz, y como consecuencia hay posibilidades de cometer errores, omisiones o ambigüedades. Este principio implica que debe ponerse atención especial al análisis, diseño, construcción y prueba de las interfaces.

Principio 5. ***Construir software que tenga modularidad eficaz.*** La separación de entidades (principio 1) establece una filosofía para el software. La *modularidad* proporciona un mecanismo para llevar a cabo dicha filosofía. Cualquier sistema complejo puede dividirse en módulos (componentes), pero la buena práctica de la ingeniería de software demanda más. La modularidad debe ser *eficaz*. Es decir, cada módulo debe centrarse exclusivamente en un aspecto bien delimitado del sistema: debe ser cohesivo en su función o restringido en el contenido que representa. Además, los módulos deben estar interconectados en forma

relativamente sencilla: cada módulo debe tener poco acoplamiento con otros módulos, fuentes de datos y otros aspectos ambientales.

Principio 6. *Buscar patrones.* Brad Appleton [App00] sugiere que:

> El objetivo de los patrones dentro de la comunidad de software es crear un cúmulo de bibliografía que ayude a los desarrolladores de software a resolver problemas recurrentes que surgen a lo largo del desarrollo. Los patrones ayudan a crear un lenguaje compartido para comunicar perspectiva y experiencia acerca de dichos patrones y sus soluciones. La codificación formal de estas soluciones y sus relaciones permite acumular con éxito el cuerpo de conocimientos que define nuestra comprensión de las buenas arquitecturas que satisfacen las necesidades de sus usuarios.

CONSEJO

Use patrones (véase el capítulo 12) a fin de acumular conocimiento y experiencia para las futuras generaciones de ingenieros de software.

Principio 7. *Cuando sea posible, representar el problema y su solución desde varias perspectivas diferentes.* Cuando un problema y su solución se estudian desde varias perspectivas distintas, es más probable que se tenga mayor visión y que se detecten los errores y omisiones. Por ejemplo, un modelo de requerimientos puede representarse con el empleo de un punto de vista orientado a los datos, a la función o al comportamiento (véanse los capítulos 6 y 7). Cada uno brinda un punto de vista diferente del problema y de sus requerimientos.

Principio 8. *Tener en mente que alguien dará mantenimiento al software.* El software será corregido en el largo plazo, cuando se descubran sus defectos, se adapte a los cambios de su ambiente y se mejore en el momento en el que los participantes pidan más capacidades. Estas actividades de mantenimiento resultan más fáciles si se aplica una práctica sólida de ingeniería de software a lo largo del proceso de software.

Estos principios no son todo lo que se necesita para elaborar software de alta calidad, pero establecen el fundamento para todos los métodos de ingeniería de software que se estudian en este libro.

4.3 Principios que guían toda actividad estructural

Cita:

"El ingeniero ideal es una mezcla… no es un científico, no es un matemático, no es un sociólogo ni un escritor; pero para resolver problemas de ingeniería utiliza conocimiento y técnicas de algunas o de todas esas disciplinas."

N. W. Dougherty

En las secciones que siguen se consideran los principios que tienen mucha relevancia para el éxito de cada actividad estructural genérica, definida como parte del proceso de software. En muchos casos, los principios que se estudian para cada una de las actividades estructurales son un refinamiento de los principios presentados en la sección 4.2. Tan sólo son principios fundamentales planteados en un nivel más bajo de abstracción.

4.3.1 Principios de comunicación

Antes de que los requerimientos del cliente se analicen, modelen o especifiquen, deben recabarse a través de la actividad de comunicación. Un cliente tiene un problema que parece abordable mediante una solución basada en computadora. Usted responde a la solicitud de ayuda del cliente. Ha comenzado la comunicación. Pero es frecuente que el camino que lleva de la comunicación a la comprensión esté lleno de agujeros.

La comunicación efectiva (entre colegas técnicos, con el cliente y otros participantes, y con los gerentes de proyecto) se encuentra entre las actividades más difíciles que deben enfrentarse. En este contexto, aquí se estudian principios de comunicación aplicados a la comunicación con el cliente. Sin embargo, muchos de ellos se aplican por igual en todas las formas de comunicación que ocurren dentro de un proyecto de software.

Principio 1. *Escuchar.* Trate de centrarse en las palabras del hablante en lugar de formular su respuesta a dichas palabras. Si algo no está claro, pregunte para aclararlo, pero evite las interrupciones constantes. Si una persona habla, *nunca* parezca usted beligerante en sus palabras o actos (por ejemplo, con giros de los ojos o movimientos de la cabeza).

Antes de comunicarse, asegúrese de que entiende el punto de vista de la otra parte, conozca un poco sus necesidades y después escuche.

Principio 2. ***Antes de comunicarse, prepararse.*** Dedique algún tiempo a entender el problema antes de reunirse con otras personas. Si es necesario, haga algunas investigaciones para entender el vocabulario propio del negocio. Si tiene la responsabilidad de conducir la reunión, prepare una agenda antes de que ésta tenga lugar.

Principio 3. ***Alguien debe facilitar la actividad.*** Toda reunión de comunicación debe tener un líder (facilitador) que: 1) mantenga la conversación en movimiento hacia una dirección positiva, 2) sea un mediador en cualquier conflicto que ocurra y 3) garantice que se sigan otros principios.

Cita:

"Las preguntas directas y las respuestas directas son el camino más corto hacia las mayores perplejidades."

Mark Twain

Principio 4. ***Es mejor la comunicación cara a cara.*** Pero por lo general funciona mejor cuando está presente alguna otra representación de la información relevante. Por ejemplo, un participante quizá genere un dibujo o documento en "borrador" que sirva como centro de la discusión.

Principio 5. ***Tomar notas y documentar las decisiones.*** Las cosas encuentran el modo de caer en las grietas. Alguien que participe en la comunicación debe servir como "secretario" y escribir todos los temas y decisiones importantes.

Principio 6. ***Perseguir la colaboración.*** La colaboración y el consenso ocurren cuando el conocimiento colectivo de los miembros del equipo se utiliza para describir funciones o características del producto o sistema. Cada pequeña colaboración sirve para generar confianza entre los miembros del equipo y crea un objetivo común para el grupo.

Principio 7. ***Permanecer centrado; hacer módulos con la discusión.*** Entre más personas participen en cualquier comunicación, más probable es que la conversación salte de un tema a otro. El facilitador debe formar módulos de conversación para abandonar un tema sólo después de que se haya resuelto (sin embargo, considere el principio 9).

Principio 8. ***Si algo no está claro, hacer un dibujo.*** La comunicación verbal tiene sus límites. Con frecuencia, un esquema o dibujo arroja claridad cuando las palabras no bastan para hacer el trabajo.

¿Qué pasa si no puede llegarse a un acuerdo con el cliente en algún aspecto relacionado con el proyecto?

Principio 9. ***a) Una vez que se acuerde algo, avanzar. b) Si no es posible ponerse de acuerdo en algo, avanzar. c) Si una característica o función no está clara o no puede aclararse en el momento, avanzar.*** La comunicación, como cualquier actividad de ingeniería de software, requiere tiempo. En vez de hacer iteraciones sin fin, las personas que participan deben reconocer que hay muchos temas que requieren análisis (véase el principio 2) y que "avanzar" es a veces la mejor forma de tener agilidad en la comunicación.

Principio 10. ***La negociación no es un concurso o un juego. Funciona mejor cuando las dos partes ganan.*** Hay muchas circunstancias en las que usted y otros participantes deben negociar funciones y características, prioridades y fechas de entrega. Si el equipo ha

INFORMACIÓN

La diferencia entre los clientes y los usuarios finales

Los ingenieros de software se comunican con muchos participantes diferentes, pero los clientes y los usuarios finales son quienes tienen el efecto más significativo en el trabajo técnico que se desarrollará. En ciertos casos, el cliente y el usuario final son la misma persona, pero para muchos proyectos son individuos distintos que trabajan para diferentes gerentes en distintas organizaciones de negocios.

Un *cliente* es la persona o grupo que 1) solicitó originalmente que se construyera el software, 2) define los objetivos generales del negocio para el software, 3) proporciona los requerimientos básicos del producto y 4) coordina la obtención de fondos para el proyecto. En un negocio de productos o sistema, es frecuente que el cliente sea el departamento de mercadotecnia. En un ambiente de tecnologías de la información (TI), el cliente tal vez sea un componente o departamento del negocio.

Un *usuario final* es la persona o grupo que 1) usará en realidad el software que se elabore para lograr algún propósito del negocio y 2) definirá los detalles de operación del software de modo que se alcance el propósito del negocio.

CasaSegura

Errores de comunicación

La escena: Lugar de trabajo del equipo de ingeniería de software.

Participantes: Jamie Lazar, Vinod Roman y Ed Robins, miembros del equipo de software.

La conversación:

Ed: ¿Qué has oído sobre el proyecto *CasaSegura*?

Vinod: La reunión de arranque está programada para la semana siguiente.

Jamie: Traté de investigar algo, pero no salió bien.

Ed: ¿Qué quieres decir?

Jamie: Bueno, llamé a Lisa Pérez. Ella es la encargada de mercadotecnia en esto.

Vinod: ¿Y…?

Jamie: Yo quería que me dijera las características y funciones de *CasaSegura*… esa clase de cosas. En lugar de ello, comenzó a hacerme preguntas sobre sistemas de seguridad, de vigilancia… No soy experto en eso.

Vinod: ¿Qué te dice eso?

(Jamie se encoge de hombros.)

Vinod: Será que mercadotecnia quiere que actuemos como consultores y mejor que hagamos alguna tarea sobre esta área de productos antes de nuestra junta de arranque. Doug dijo que quería que "colaboráramos" con nuestro cliente, así que será mejor que aprendamos cómo hacerlo.

Ed: Tal vez hubiera sido mejor ir a su oficina. Las llamadas por teléfono simplemente no sirven para esta clase de trabajos.

Jamie: Están en lo correcto. Tenemos que actuar juntos o nuestras primeras comunicaciones serán una batalla.

Vinod: Yo vi a Doug leyendo un libro acerca de "requerimientos de ingeniería". Apuesto a que enlista algunos principios de buena comunicación. Voy a pedírselo prestado.

Jamie: Buena idea… luego nos enseñas.

Vinod (sonríe): Sí, de acuerdo.

colaborado bien, todas las partes tendrán un objetivo común. Aun así, la negociación demandará el compromiso de todas las partes.

4.3.2 Principios de planeación

La actividad de comunicación ayuda a definir las metas y objetivos generales (por supuesto, sujetos al cambio conforme pasa el tiempo). Sin embargo, la comprensión de estas metas y objetivos no es lo mismo que definir un plan para lograrlo. La actividad de planeación incluye un conjunto de prácticas administrativas y técnicas que permiten que el equipo de software defina un mapa mientras avanza hacia su meta estratégica y sus objetivos tácticos.

Cita:

"Al prepararme para una batalla siempre descubro que los planes son inútiles, pero que la planeación es indispensable."

General Dwight D. Eisenhower

Créalo, es imposible predecir con exactitud cómo se desarrollará un proyecto de software. No existe una forma fácil de determinar qué problemas técnicos se encontrarán, qué información importante permanecerá oculta hasta que el proyecto esté muy avanzado, qué malos entendidos habrá o qué aspectos del negocio cambiarán. No obstante, un buen equipo de software debe planear con este enfoque.

Hay muchas filosofías de planeación.[2] Algunas personas son "minimalistas" y afirman que es frecuente que el cambio elimine la necesidad de hacer un plan detallado. Otras son "tradicionalistas" y dicen que el plan da un mapa eficaz y que entre más detalles tenga menos probable será que el equipo se pierda. Otros más son "agilistas" y plantean que tal vez sea necesario un "juego de planeación" rápido, pero que el mapa surgirá a medida que comience el "trabajo real" con el software.

WebRef

En la dirección **www.4pm.com/repository.htm**, hay excelentes materiales informativos sobre la planeación y administración de proyectos.

¿Qué hacer? En muchos proyectos, planear en exceso consume tiempo y es estéril (porque son demasiadas las cosas que cambian), pero planear poco es una receta para el caos. Igual que la mayoría de cosas de la vida, la planeación debe ser tomada con moderación, suficiente para que dé una guía útil al equipo, ni más ni menos. Sin importar el rigor con el que se haga la planeación, siempre se aplican los principios siguientes:

2 En la parte 4 de este libro hay un análisis detallado de la planeación y administración de proyectos de software.

Principio 1. ***Entender el alcance del proyecto.*** Es imposible usar el mapa si no se sabe a dónde se va. El alcance da un destino al equipo de software.

Principio 2. ***Involucrar en la actividad de planeación a los participantes del software.*** Los participantes definen las prioridades y establecen las restricciones del proyecto. Para incluir estas realidades, es frecuente que los ingenieros de software deban negociar la orden de entrega, los plazos y otros asuntos relacionados con el proyecto.

Principio 3. ***Reconocer que la planeación es iterativa.*** Un plan para el proyecto nunca está grabado en piedra. Para cuando el trabajo comience, es muy probable que las cosas hayan cambiado. En consecuencia, el plan deberá ajustarse para incluir dichos cambios. Además, los modelos de proceso iterativo incrementales dictan que debe repetirse la planeación después de la entrega de cada incremento de software, con base en la retroalimentación recibida de los usuarios.

Cita:

"El éxito es más una función del sentido común coherente que del genio."

An Wang

Principio 4. ***Estimar con base en lo que se sabe.*** El objetivo de la estimación es obtener un índice del esfuerzo, costo y duración de las tareas, con base en la comprensión que tenga el equipo sobre el trabajo que va a realizar. Si la información es vaga o poco confiable, entonces las estimaciones tampoco serán confiables.

Principio 5. ***Al definir el plan, tomar en cuenta los riesgos.*** Si ha identificado riesgos que tendrían un efecto grande y es muy probable que ocurran, entonces es necesario elaborar planes de contingencia. Además, el plan del proyecto (incluso la programación de actividades) deberá ajustarse para que incluya la posibilidad de que ocurran uno o más de dichos riesgos.

Principio 6. ***Ser realista.*** Las personas no trabajan 100% todos los días. En cualquier comunicación humana hay ruido. Las omisiones y ambigüedad son fenómenos de la vida. Los cambios ocurren. Aun los mejores ingenieros de software cometen errores. Éstas y otras realidades deben considerarse al establecer un proyecto.

PUNTO CLAVE

El término *granularidad* se refiere al detalle con el que se representan o efectúan algunos elementos de la planeación.

Principio 7. ***Ajustar la granularidad cuando se defina el plan.*** La *granularidad* se refiere al nivel de detalle que se adopta cuando se desarrolla un plan. Un plan con "mucha granularidad" proporciona detalles significativos en las tareas para el trabajo que se planea, en incrementos durante un periodo relativamente corto (por lo que el seguimiento y control suceden con frecuencia). Un plan con "poca granularidad" da tareas más amplias para el trabajo que se planea, para plazos más largos. En general, la granularidad va de poca a mucha conforme el tiempo avanza. En las siguientes semanas o meses, el proyecto se planea con detalles significativos. Las actividades que no ocurrirán en muchos meses no requieren mucha granularidad (hay demasiadas cosas que pueden cambiar).

Principio 8. ***Definir cómo se trata de asegurar la calidad.*** El plan debe identificar la forma en la que el equipo de software busca asegurar la calidad. Si se realizan revisiones técnicas,[3] deben programarse. Si durante la construcción va a utilizarse programación por parejas (véase el capítulo 3), debe definirse en forma explícita en el plan.

Principio 9. ***Describir cómo se busca manejar el cambio.*** Aun la mejor planeación puede ser anulada por el cambio sin control. Debe identificarse la forma en la que van a recibirse los cambios a medida que avanza el trabajo de la ingeniería de software. Por ejemplo, ¿el cliente tiene la posibilidad de solicitar un cambio en cualquier momento? Si se solicita uno, ¿está obligado el equipo a implementarlo de inmediato? ¿Cómo se evalúan el efecto y el costo del cambio?

3 Las revisiones técnicas se estudian en el capítulo 15.

Principio 10. ***Dar seguimiento al plan con frecuencia y hacer los ajustes que se requieran.*** Los proyectos de software se atrasan respecto de su programación. Por tanto, tiene sentido evaluar diariamente el avance, en busca de áreas y situaciones problemáticas en las que las actividades programadas no se apeguen al avance real. Cuando se detecten desviaciones, hay que ajustar el plan en consecuencia.

Para ser más eficaz, cada integrante del equipo de software debe participar en la actividad de planeación. Sólo entonces sus miembros "firmarán" el plan.

4.3.3 Principios de modelado

Se crean modelos para entender mejor la entidad real que se va a construir. Cuando ésta es física (por ejemplo, un edificio, un avión, una máquina, etc.), se construye un modelo de forma idéntica pero a escala. Sin embargo, cuando la entidad que se va a construir es software, el modelo debe adoptar una forma distinta. Debe ser capaz de representar la información que el software transforma, la arquitectura y las funciones que permiten que esto ocurra, las características que desean los usuarios y el comportamiento del sistema mientras la transformación tiene lugar. Los modelos deben cumplir estos objetivos en diferentes niveles de abstracción, en primer lugar con la ilustración del software desde el punto de vista del cliente y después con su representación en un nivel más técnico.

Los modelos de requerimientos representan los requerimientos del cliente. Los modelos del diseño dan una especificación concreta para la construcción del software.

En el trabajo de ingeniería de software se crean dos clases de modelos: de requerimientos y de diseño. Los *modelos de requerimientos* (también conocidos como *modelos de análisis*) representan los requerimientos del cliente mediante la ilustración del software en tres dominios diferentes: el de la información, el funcional y el de comportamiento. Los *modelos de diseño* representan características del software que ayudan a los profesionales a elaborarlo con eficacia: arquitectura, interfaz de usuario y detalle en el nivel de componente.

En su libro sobre modelado ágil, Scott Ambler y Ron Jeffries [Amb02b] definen un conjunto de principios de modelado[4] dirigidos a todos aquellos que usan el modelo de proceso ágil (véase el capítulo 3), pero que son apropiados para todos los ingenieros de software que efectúan acciones y tareas de modelado:

Principio 1. ***El equipo de software tiene como objetivo principal elaborar software, no crear modelos.*** Agilidad significa entregar software al cliente de la manera más rápida posible. Los modelos que contribuyan a esto son benéficos, pero deben evitarse aquellos que hagan lento el proceso o que den poca perspectiva.

Principio 2. ***Viajar ligero, no crear más modelos de los necesarios.*** Todo modelo que se cree debe actualizarse si ocurren cambios. Más importante aún es que todo modelo nuevo exige tiempo, que de otra manera se destinaría a la construcción (codificación y pruebas). Entonces, cree sólo aquellos modelos que hagan más fácil y rápido construir el software.

El objetivo de cualquier modelo es comunicar información. Para lograr esto, use un formato consistente. Suponga que usted no estará para explicar el modelo. Por eso, el modelo debe describirse por sí solo.

Principio 3. ***Tratar de producir el modelo más sencillo que describa al problema o al software.*** No construya software en demasía [Amb02b]. Al mantener sencillos los modelos, el software resultante también lo será. El resultado es que se tendrá un software fácil de integrar, de probar y de mantener (para que cambie). Además, los modelos sencillos son más fáciles de entender y criticar por parte de los miembros del equipo, lo que da como resultado un formato funcional de retroalimentación que optimiza el resultado final.

Principio 4. ***Construir modelos susceptibles al cambio.*** Suponga que sus modelos cambiarán, pero vigile que esta suposición no lo haga descuidado. Por ejemplo, como los

4 Para fines de este libro, se han abreviado y reescrito los principios mencionados en esta sección.

requerimientos se modificarán, hay una tendencia a ignorar los modelos. ¿Por qué? Porque se sabe que de todos modos cambiarán. El problema con esta actitud es que sin un modelo razonablemente completo de los requerimientos, se creará un diseño (modelo de diseño) que de manera invariable carecerá de funciones y características importantes.

Principio 5. ***Ser capaz de enunciar un propósito explícito para cada modelo que se cree.*** Cada vez que cree un modelo, pregúntese por qué lo hace. Si no encuentra una razón sólida para la existencia del modelo, no pierda tiempo en él.

Principio 6. ***Adaptar los modelos que se desarrollan al sistema en cuestión.*** Tal vez sea necesario adaptar a la aplicación la notación del modelo o las reglas; por ejemplo, una aplicación de juego de video quizá requiera una técnica de modelado distinta que el software incrustado que controla el motor de un automóvil en tiempo real.

Principio 7. ***Tratar de construir modelos útiles, pero olvidarse de elaborar modelos perfectos.*** Cuando un ingeniero de software construye modelos de requerimientos y diseño, alcanza un punto de rendimientos decrecientes. Es decir, el esfuerzo requerido para terminar por completo el modelo y hacerlo internamente consistente deja de beneficiarse por tener dichas propiedades. ¿Se sugiere que el modelado debe ser pobre o de baja calidad? La respuesta es "no". Pero el modelado debe hacerse con la mirada puesta en las siguientes etapas de la ingeniería de software. Las iteraciones sin fin para obtener un modelo "perfecto" no cumplen la necesidad de agilidad.

Principio 8. ***No ser dogmático respecto de la sintaxis del modelo. Si se tiene éxito para comunicar contenido, la representación es secundaria.*** Aunque cada miembro del equipo de software debe tratar de usar una notación consistente durante el modelado, la característica más importante del modelo es comunicar información que permita la realización de la siguiente tarea de ingeniería. Si un modelo tiene éxito en hacer esto, es perdonable la sintaxis incorrecta.

Principio 9. ***Si su instinto dice que un modelo no es el correcto a pesar de que se vea bien en el papel, hay razones para estar preocupado.*** Si usted es un ingeniero de software experimentado, confíe en su instinto. El trabajo de software enseña muchas lecciones, algunas en el nivel del inconsciente. Si algo le dice que un modelo de diseño está destinado a fracasar (aun cuando esto no pueda demostrarse en forma explícita), hay razones para dedicar más tiempo a su estudio o a desarrollar otro distinto.

Principio 10. ***Obtener retroalimentación tan pronto como sea posible.*** Todo modelo debe ser revisado por los miembros del equipo. El objetivo de estas revisiones es obtener retroalimentación para utilizarla a fin de corregir los errores de modelado, cambiar las interpretaciones equivocadas y agregar las características o funciones omitidas inadvertidamente.

Requerimientos de los principios de modelado. En las últimas tres décadas se han desarrollado numerosos métodos de modelado de requerimientos. Los investigadores han identificado los problemas del análisis de requerimientos y sus causas, y han desarrollado varias notaciones de modelado y los conjuntos heurísticos correspondientes para resolver aquéllos. Cada método de análisis tiene un punto de vista único. Sin embargo, todos están relacionados por ciertos principios operacionales:

Principio 1. ***Debe representarse y entenderse el dominio de información de un problema.*** El *dominio de información* incluye los datos que fluyen hacia el sistema (usuarios finales, otros sistemas o dispositivos externos), los datos que fluyen fuera del sistema (por la interfaz de usuario, interfaces de red, reportes, gráficas y otros medios) y los almacenamientos de datos que recaban y organizan objetos persistentes de datos (por ejemplo, aquellos que se conservan en forma permanente).

PUNTO CLAVE

El modelado del análisis se centra en tres atributos del software: la información que se va a procesar, la función que se va a entregar y el comportamiento que va a suceder.

Principio 2. ***Deben definirse las funciones que realizará el software.*** Las funciones del software dan un beneficio directo a los usuarios finales y también brindan apoyo interno para las características que son visibles para aquéllos. Algunas funciones transforman los datos que fluyen hacia el sistema. En otros casos, las funciones activan algún nivel de control sobre el procesamiento interno del software o sobre los elementos externos del sistema. Las funciones se describen en muchos y distintos niveles de abstracción, que van de un enunciado de propósito general a la descripción detallada de los elementos del procesamiento que deben invocarse.

Cita:

"En cualquier trabajo de diseño, el primer problema del ingeniero es descubrir cuál es realmente el problema."

Autor desconocido

Principio 3. ***Debe representarse el comportamiento del software (como consecuencia de eventos externos).*** El comportamiento del software de computadora está determinado por su interacción con el ambiente externo. Las entradas que dan los usuarios finales, el control de los datos efectuado por un sistema externo o la vigilancia de datos reunidos en una red son el motivo por el que el software se comporta en una forma específica.

Principio 4. ***Los modelos que representen información, función y comportamiento deben dividirse de manera que revelen los detalles en forma estratificada (o jerárquica).*** El modelado de los requerimientos es el primer paso para resolver un problema de ingeniería de software. Eso permite entender mejor el problema y proporciona una base para la solución (diseño). Los problemas complejos son difíciles de resolver por completo. Por esta razón, debe usarse la estrategia de *divide y vencerás*. Un problema grande y complejo se divide en subproblemas hasta que cada uno de éstos sea relativamente fácil de entender. Este concepto se llama *partición* o *separación de entidades*, y es una estrategia clave en el modelado de requerimientos.

Principio 5. ***El trabajo de análisis debe avanzar de la información esencial hacia la implementación en detalle.*** El modelado de requerimientos comienza con la descripción del problema desde la perspectiva del usuario final. Se describe la "esencia" del problema sin considerar la forma en la que se implementará la solución. Por ejemplo, un juego de video requiere que la jugadora "enseñe" a su protagonista en qué dirección avanzar cuando se mueve hacia un laberinto peligroso. Ésa es la esencia del problema. La implementación detallada (normalmente descrita como parte del modelo del diseño) indica cómo se desarrollará la esencia. Para el juego de video, quizá se use una entrada de voz, o se escriba un comando en un teclado, o tal vez un *joystick* (o *mouse*) apunte en una dirección específica, o quizá se mueva en el aire un dispositivo sensible al movimiento.

Con la aplicación de estos principios, un ingeniero de software aborda el problema en forma sistemática. Pero, ¿cómo se aplican estos principios en la práctica? Esta pregunta se responderá en los capítulos 5 a 7.

Principios del modelado del diseño. El modelo del diseño del software es análogo a los planos arquitectónicos de una casa. Se comienza por representar la totalidad de lo que se va a construir (por ejemplo, un croquis tridimensional de la casa) que se refina poco a poco para que guíe la construcción de cada detalle (por ejemplo, la distribución de la plomería). De manera similar, el modelo del diseño que se crea para el software da varios puntos de vista distintos del sistema.

Cita:

"Vea primero que el diseño es sabio y justo: eso comprobado, siga resueltamente; no para uno renunciar a rechazar el propósito de que ha resuelto llevar a cabo."

William Shakespeare

No escasean los métodos para obtener los distintos elementos de un diseño de software. Algunos son activados por datos, lo que hace que sea la estructura de éstos la que determine la arquitectura del programa y los componentes de procesamiento resultantes. Otros están motivados por el patrón, y usan información sobre el dominio del problema (el modelo de requerimientos) para desarrollar estilos de arquitectura y patrones de procesamiento. Otros más están orientados a objetos, y utilizan objetos del dominio del problema como impulsores de la creación de estructuras de datos y métodos que los manipulan. No obstante la variedad, todos ellos se apegan a principios de diseño que se aplican sin importar el método empleado.

Principio 1. ***El diseño debe poderse rastrear hasta el modelo de requerimientos.*** El modelo de requerimientos describe el dominio de información del problema, las funciones visibles para el usuario, el comportamiento del sistema y un conjunto de clases de requerimientos que agrupa los objetos del negocio con los métodos que les dan servicio. El modelo de diseño traduce esta información en una arquitectura, un conjunto de subsistemas que implementan las funciones principales y un conjunto de componentes que son la realización de las clases de requerimientos. Los elementos del modelo de diseño deben poder rastrearse en el modelo de requerimientos.

WebRef

En la dirección **cs.wwc.edu/~aabyan/Design/**, se encuentran comentarios profundos sobre el proceso de diseño, así como un análisis de la estética del diseño.

Principio 2. ***Siempre tomar en cuenta la arquitectura del sistema que se va a construir.*** La arquitectura del software (véase el capítulo 9) es el esqueleto del sistema que se va a construir. Afecta interfaces, estructuras de datos, flujo de control y comportamiento del programa, así como la manera en la que se realizarán las pruebas, la susceptibilidad del sistema resultante a recibir mantenimiento y mucho más. Por todas estas razones, el diseño debe comenzar con consideraciones de la arquitectura. Sólo después de establecida ésta deben considerarse los aspectos en el nivel de los componentes.

Principio 3. ***El diseño de los datos es tan importante como el de las funciones de procesamiento.*** El diseño de los datos es un elemento esencial del diseño de la arquitectura. La forma en la que los objetos de datos se elaboran dentro del diseño no puede dejarse al azar. Un diseño de datos bien estructurado ayuda a simplificar el flujo del programa, hace más fácil el diseño e implementación de componentes de software y más eficiente el procesamiento conjunto.

Cita:

"Las diferencias no son menores; por el contrario, son como las que había entre Salieri y Mozart. Un estudio tras otro muestran que los mejores diseñadores elaboran estructuras más rápidas, pequeñas, sencillas, claras y producidas con menos esfuerzo."

Frederick P. Brooks

Principio 4. ***Las interfaces (tanto internas como externas) deben diseñarse con cuidado.*** La manera en la que los datos fluyen entre los componentes de un sistema tiene mucho que ver con la eficiencia del procesamiento, la propagación del error y la simplicidad del diseño. Una interfaz bien diseñada hace que la integración sea más fácil y ayuda a quien la somete a prueba a validar las funciones componentes.

Principio 5. ***El diseño de la interfaz de usuario debe ajustarse a las necesidades del usuario final. Sin embargo, en todo caso debe resaltar la facilidad de uso.*** La interfaz de usuario es la manifestación visible del software. No importa cuán sofisticadas sean sus funciones internas, ni lo incluyentes que sean sus estructuras de datos, ni lo bien diseñada que esté su arquitectura, un mal diseño de la interfaz con frecuencia conduce a la percepción de que el software es "malo".

Principio 6. ***El diseño en el nivel de componentes debe tener independencia funcional.*** La independencia funcional es una medida de la "mentalidad única" de un componente de software. La funcionalidad que entrega un componente debe ser cohesiva, es decir, debe centrarse en una y sólo una función o subfunción.[5]

Principio 7. ***Los componentes deben estar acoplados con holgura entre sí y con el ambiente externo.*** El acoplamiento se logra de muchos modos: con una interfaz de componente, con mensajería, por medio de datos globales, etc. A medida que se incrementa el nivel de acoplamiento, también aumenta la probabilidad de propagación del error y disminuye la facilidad general de dar mantenimiento al software. Entonces, el acoplamiento de componentes debe mantenerse tan bajo como sea razonable.

Principio 8. ***Las representaciones del diseño (modelos) deben entenderse con facilidad.*** El propósito del diseño es comunicar información a los profesionales que generarán el código, a los que probarán el software y a otros que le darán mantenimiento en el futuro. Si el diseño es difícil de entender, no servirá como medio de comunicación eficaz.

5 En el capítulo 8 hay más análisis de la cohesión.

Principio 9. ***El diseño debe desarrollarse en forma iterativa. El diseñador debe buscar más sencillez en cada iteración.*** Igual que ocurre con casi todas las actividades creativas, el diseño ocurre de manera iterativa. Las primeras iteraciones sirven para mejorar el diseño y corregir errores, pero las posteriores deben buscar un diseño tan sencillo como sea posible.

Cuando se aplican en forma apropiada estos principios de diseño, se crea uno que exhibe factores de calidad tanto externos como internos [Mye78]. Los *factores de calidad externos* son aquellas propiedades del software fácilmente observables por los usuarios (por ejemplo, velocidad, confiabilidad, corrección y usabilidad). Los *factores de calidad internos* son de importancia para los ingenieros de software. Conducen a un diseño de alta calidad desde el punto de vista técnico. Para obtener factores de calidad internos, el diseñador debe entender los conceptos básicos del diseño (véase el capítulo 8).

4.3.4 Principios de construcción

Cita:

"Durante gran parte de mi vida he sido un mirón del software, y observo furtivamente el código sucio de otras personas. A veces encuentro una verdadera joya, un programa bien estructurado escrito en un estilo consistente, libre de errores, desarrollado de modo que cada componente es sencillo y organizado, y que está diseñado de modo que el producto es fácil de cambiar."

David Parnas

La actividad de construcción incluye un conjunto de tareas de codificación y pruebas que lleva a un software operativo listo para entregarse al cliente o usuario final. En el trabajo de ingeniería de software moderna, la codificación puede ser 1) la creación directa de lenguaje de programación en código fuente (por ejemplo, Java), 2) la generación automática de código fuente que usa una representación intermedia parecida al diseño del componente que se va a construir o 3) la generación automática de código ejecutable que utiliza un "lenguaje de programación de cuarta generación" (por ejemplo, Visual C++).

Las pruebas dirigen su atención inicial al componente, y con frecuencia se denomina *prueba unitaria*. Otros niveles de pruebas incluyen 1) *de integración* (realizadas mientras el sistema está en construcción), 2) *de validación*, que evalúan si los requerimientos se han satisfecho para todo el sistema (o incremento de software) y 3) *de aceptación*, que efectúa el cliente en un esfuerzo por utilizar todas las características y funciones requeridas. Los siguientes principios y conceptos son aplicables a la codificación y prueba:

Principios de codificación. Los principios que guían el trabajo de codificación se relacionan de cerca con el estilo, lenguajes y métodos de programación. Sin embargo, puede enunciarse cierto número de principios fundamentales:

Principios de preparación: ***Antes de escribir una sola línea de código, asegúrese de:***

- Entender el problema que se trata de resolver.
- Comprender los principios y conceptos básicos del diseño.
- Elegir un lenguaje de programación que satisfaga las necesidades del software que se va a elaborar y el ambiente en el que operará.
- Seleccionar un ambiente de programación que disponga de herramientas que hagan más fácil su trabajo.
- Crear un conjunto de pruebas unitarias que se aplicarán una vez que se haya terminado el componente a codificar.

CONSEJO

Evite desarrollar un programa elegante que resuelva el problema equivocado. Ponga especial atención al primer principio de preparación.

Principios de programación: ***Cuando comience a escribir código, asegúrese de:***

- Restringir sus algoritmos por medio del uso de programación estructurada [Boh00].
- Tomar en consideración el uso de programación por parejas.
- Seleccionar estructuras de datos que satisfagan las necesidades del diseño.
- Entender la arquitectura del software y crear interfaces que son congruentes con ella.
- Mantener la lógica condicional tan sencilla como sea posible.

- Crear lazos anidados en forma tal que se puedan probar con facilidad.
- Seleccionar nombres significativos para las variables y seguir otros estándares locales de codificación.
- Escribir código que se documente a sí mismo.
- Crear una imagen visual (por ejemplo, líneas con sangría y en blanco) que ayude a entender.

Principios de validación: ***Una vez que haya terminado su primer intento de codificación, asegúrese de:***

- Realizar el recorrido del código cuando sea apropiado.
- Llevar a cabo pruebas unitarias y corregir los errores que se detecten.
- Rediseñar el código.

WebRef

En la dirección **www.literateprogramming.com/fpstyle.html**, hay una amplia variedad de vínculos a estándares de codificación.

Se han escrito más libros sobre programación (codificación) y sobre los principios y conceptos que la guían que sobre cualquier otro tema del proceso de software. Los libros sobre el tema incluyen obras tempranas sobre estilo de programación [Ker78], construcción de software práctico [McC04], perlas de programación [Ben99], el arte de programar [Knu98], temas pragmáticos de programación [Hun99] y muchísimos temas más. El análisis exhaustivo de estos principios y conceptos está más allá del alcance de este libro. Si tiene interés en profundizar, estudie una o varias de las referencias que se mencionan.

Principios de la prueba. En un libro clásico sobre las pruebas de software, Glen Myers [Mye79] enuncia algunas reglas que sirven bien como objetivos de prueba:

? ¿Cuáles son los objetivos de probar el software?

- La prueba es el proceso que ejecuta un programa con objeto de encontrar un error.
- Un buen caso de prueba es el que tiene alta probabilidad de encontrar un error que no se ha detectado hasta el momento.
- Una prueba exitosa es la que descubre un error no detectado hasta el momento.

CONSEJO

En un contexto amplio del diseño de software, recuerde que se comienza "por lo grande" y se centra en la arquitectura del software, y que se termina "en lo pequeño" y se atiende a los componentes. Para la prueba sólo se invierte el proceso.

Estos objetivos implican un cambio muy grande en el punto de vista de ciertos desarrolladores de software. Ellos avanzan contra la opinión común de que una prueba exitosa es aquella que no encuentra errores en el software. El objetivo es diseñar pruebas que detecten de manera sistemática diferentes clases de errores, y hacerlo con el mínimo tiempo y esfuerzo.

Si las pruebas se efectúan con éxito (de acuerdo con los objetivos ya mencionados), descubrirán errores en el software. Como beneficio secundario, la prueba demuestra que las funciones de software parecen funcionar de acuerdo con las especificaciones, y que los requerimientos de comportamiento y desempeño aparentemente se cumplen. Además, los datos obtenidos conforme se realiza la prueba dan una buena indicación de la confiabilidad del software y ciertas indicaciones de la calidad de éste como un todo. Pero las pruebas no pueden demostrar la inexistencia de errores y defectos; sólo demuestran que hay errores y defectos. Es importante recordar esto (que de otro modo parecería muy pesimista) cuando se efectúe una prueba.

Davis [Dav95b] sugiere algunos principios para las pruebas,[6] que se han adaptado para usarlos en este libro:

Principio 1. ***Todas las pruebas deben poder rastrearse hasta los requerimientos del cliente.***[7] El objetivo de las pruebas de software es descubrir errores. Entonces, los defec-

6 Aquí sólo se mencionan pocos de los principios de prueba de Davis. Para más información, consulte [Dav95b].

7 Este principio se refiere a las *pruebas funcionales*, por ejemplo, aquellas que se centran en los requerimientos. Las *pruebas estructurales* (las que se centran en los detalles de arquitectura o lógica) tal vez no aborden directamente los requerimientos específicos.

tos más severos (desde el punto de vista del cliente) son aquellos que hacen que el programa no cumpla sus requerimientos.

Principio 2. ***Las pruebas deben planearse mucho antes de que den comienzo.*** La planeación de las pruebas (véase el capítulo 17) comienza tan pronto como se termina el modelo de requerimientos. La definición detallada de casos de prueba principia apenas se ha concluido el modelo de diseño. Por tanto, todas las pruebas pueden planearse y diseñarse antes de generar cualquier código.

Principio 3. ***El principio de Pareto se aplica a las pruebas de software.*** En este contexto, el principio de Pareto implica que 80% de todos los errores no detectados durante las pruebas se relacionan con 20% de todos los componentes de programas. Por supuesto, el problema es aislar los componentes sospechosos y probarlos a fondo.

Principio 4. ***Las pruebas deben comenzar "en lo pequeño" y avanzar hacia "lo grande".*** Las primeras pruebas planeadas y ejecutadas por lo general se centran en componentes individuales. Conforme avanzan las pruebas, la atención cambia en un intento por encontrar errores en grupos integrados de componentes y, en última instancia, en todo el sistema.

Principio 5. ***No son posibles las pruebas exhaustivas.*** Hasta para un programa de tamaño moderado, el número de permutaciones de las rutas es demasiado grande. Por esta razón, durante una prueba es imposible ejecutar todas las combinaciones de rutas. Sin embargo, es posible cubrir en forma adecuada la lógica del programa y asegurar que se han probado todas las condiciones en el nivel de componentes.

4.3.5 Principios de despliegue

Como se dijo en la parte 1 del libro, la actividad del despliegue incluye tres acciones: entrega, apoyo y retroalimentación. Como la naturaleza de los modelos del proceso del software moderno es evolutiva o incremental, el despliegue ocurre no una vez sino varias, a medida que el software avanza hacia su conclusión. Cada ciclo de entrega pone a disposición de los clientes y usuarios finales un incremento de software operativo que brinda funciones y características utilizables. Cada ciclo de apoyo provee documentación y ayuda humana para todas las funciones y características introducidas durante los ciclos de despliegue realizados hasta ese momento. Cada ciclo de retroalimentación da al equipo de software una guía importante que da como resultado modificaciones de las funciones, de las características y del enfoque adoptado para el siguiente incremento.

Asegúrese de que su cliente sabe lo que puede esperar antes de que se entregue un incremento de software. De otra manera, puede apostar a que el cliente espera más de lo que usted le dará.

La entrega de un incremento de software representa un punto de referencia importante para cualquier proyecto de software. Cuando el equipo se prepara para entregar un incremento, deben seguirse ciertos principios clave:

Principio 1. ***Deben manejarse las expectativas de los clientes.*** Con demasiada frecuencia, el cliente espera más de lo que el equipo ha prometido entregar, y la desilusión llega de inmediato. Esto da como resultado que la retroalimentación no sea productiva y arruine la moral del equipo. En su libro sobre la administración de las expectativas, Naomi Karten [Kar94] afirma que "el punto de inicio de la administración de las expectativas es ser más consciente de lo que se comunica y de la forma en la que esto se hace". Ella sugiere que el ingeniero de software debe tener cuidado con el envío de mensajes conflictivos al cliente (por ejemplo, prometer más de lo que puede entregarse de manera razonable en el plazo previsto, o entregar más de lo que se prometió en un incremento de software y para el siguiente entregar menos).

Principio 2. ***Debe ensamblarse y probarse el paquete completo que se entregará.*** Debe ensamblarse en un CD-ROM u otro medio (incluso descargas desde web) todo el software ejecutable, archivos de datos de apoyo, documentos de ayuda y otra información rele-

vante, para después hacer una prueba beta exhaustiva con usuarios reales. Todos los *scripts* de instalación y otras características de operación deben ejecutarse por completo en tantas configuraciones diferentes de cómputo como sea posible (por ejemplo, hardware, sistemas operativos, equipos periféricos, configuraciones de red, etcétera).

Principio 3. ***Antes de entregar el software, debe establecerse un régimen de apoyo.*** Un usuario final espera respuesta e información exacta cuando surja una pregunta o problema. Si el apoyo es *ad hoc*, o, peor aún, no existe, el cliente quedará insatisfecho de inmediato. El apoyo debe planearse, los materiales respectivos deben prepararse y los mecanismos apropiados de registro deben establecerse a fin de que el equipo de software realice una evaluación categórica de las clases de apoyo solicitado.

Principio 4. Se ***deben proporcionar a los usuarios finales materiales de aprendizaje apropiados.*** El equipo de software entrega algo más que el software en sí. Deben desarrollarse materiales de capacitación apropiados (si se requirieran); es necesario proveer lineamientos para solución de problemas y, cuando sea necesario, debe publicarse "lo que es diferente en este incremento de software".[8]

Principio 5. ***El software defectuoso debe corregirse primero y después entregarse.*** Cuando el tiempo apremia, algunas organizaciones de software entregan incrementos de baja calidad con la advertencia de que los errores "se corregirán en la siguiente entrega". Esto es un error. Hay un adagio en el negocio del software que dice así: "Los clientes olvidarán pronto que entregaste un producto de alta calidad, pero nunca olvidarán los problemas que les causó un producto de mala calidad. El software se los recuerda cada día."

El software entregado brinda beneficios al usuario final, pero también da retroalimentación útil para el equipo que lo desarrolló. Cuando el incremento se libere, debe invitarse a los usuarios finales a que comenten acerca de características y funciones, facilidad de uso, confiabilidad y cualesquiera otras características.

4.4 Resumen

La práctica de la ingeniería de software incluye principios, conceptos, métodos y herramientas que los ingenieros de software aplican en todo el proceso de desarrollo. Todo proyecto de ingeniería de software es diferente. No obstante, existe un conjunto de principios generales que se aplican al proceso como un todo y a cada actividad estructural, sin importar cuál sea el proyecto o el producto.

Existe un conjunto de principios fundamentales que ayudan en la aplicación de un proceso de software significativo y en la ejecución de métodos de ingeniería de software eficaz. En el nivel del proceso, los principios fundamentales establecen un fundamento filosófico que guía al equipo de software cuando avanza por el proceso del software. En el nivel de la práctica, los principios fundamentales establecen un conjunto de valores y reglas que sirven como guía al analizar el diseño de un problema y su solución, al implementar ésta y al someterla a prueba para, finalmente, desplegar el software en la comunidad del usuario.

Los principios de comunicación se centran en la necesidad de reducir el ruido y mejorar el ancho de banda durante la conversación entre el desarrollador y el cliente. Ambas partes deben colaborar a fin de lograr la mejor comunicación.

Los principios de planeación establecen lineamientos para elaborar el mejor mapa del proceso hacia un sistema o producto terminado. El plan puede diseñarse sólo para un incremento

8 Durante la actividad de comunicación, el equipo de software debe determinar los tipos de materiales de ayuda que quiere el usuario.

del software, o para todo el proyecto. Sin que esto importe, debe definir lo que se hará, quién lo hará y cuándo se terminará el trabajo.

El modelado incluye tanto el análisis como el diseño, y describe representaciones cada vez más detalladas del software. El objetivo de los modelos es afirmar el entendimiento del trabajo que se va a hacer y dar una guía técnica a quienes implementarán el software. Los principios de modelado dan fundamento a los métodos y notación que se utilizan para crear representaciones del software.

La construcción incorpora un ciclo de codificación y pruebas en el que se genera código fuente para cierto componente y es sometido a pruebas. Los principios de codificación definen las acciones generales que deben tener lugar antes de que se escriba el código, mientras se escribe y una vez terminado. Aunque hay muchos principios para las pruebas, sólo uno predomina: la prueba es el proceso que lleva a ejecutar un programa con objeto de encontrar un error.

El despliegue ocurre cuando se presenta al cliente un incremento de software, e incluye la entrega, apoyo y retroalimentación. Los principios clave para la entrega consideran la administración de las expectativas del cliente y darle información de apoyo adecuada sobre el software. El apoyo demanda preparación anticipada. La retroalimentación permite al cliente sugerir cambios que tengan valor para el negocio y que brinden al desarrollador información para el ciclo iterativo siguiente de ingeniería de software.

Problemas y puntos por evaluar

4.1. Toda vez que la búsqueda de la calidad reclama recursos y tiempo, ¿es posible ser ágil y centrarse en ella?

4.2. De los ocho principios fundamentales que guían el proceso (lo que se estudió en la sección 4.2.1), ¿cuál cree que sea el más importante?

4.3. Describa con sus propias palabras el concepto de *separación de entidades.*

4.4. Un principio de comunicación importante establece que hay que "prepararse antes de comunicarse". ¿Cómo debe manifestarse esta preparación en los primeros trabajos que se hacen? ¿Qué productos del trabajo son resultado de la preparación temprana?

4.5. Haga algunas investigaciones acerca de cómo "facilitar" la actividad de comunicación (use las referencias que se dan u otras distintas) y prepare algunos lineamientos que se centren en la facilitación.

4.6. ¿En qué difiere la comunicación ágil de la comunicación tradicional de la ingeniería de software? ¿En qué se parecen?

4.7. ¿Por qué es necesario "avanzar"?

4.8. Investigue sobre la "negociación" para la actividad de comunicación y prepare algunos lineamientos que se centren sólo en ella.

4.9. Describa lo que significa *granularidad* en el contexto de la programación de actividades de un proyecto.

4.10. ¿Por qué son importantes los modelos en el trabajo de ingeniería de software? ¿Siempre son necesarios? ¿Hay calificadores para la respuesta que se dio sobre esta necesidad?

4.11. ¿Cuáles son los tres "dominios" considerados durante el modelado de requerimientos?

4.12. Trate de agregar un principio adicional a los que se mencionan en la sección 4.3.4 para la codificación.

4.13. ¿Qué es una prueba exitosa?

4.14. Diga si está de acuerdo o en desacuerdo con el enunciado siguiente: "Como entregamos incrementos múltiples al cliente, no debiéramos preocuparnos por la calidad en los primeros incrementos; en las iteraciones posteriores podemos corregir los problemas. Explique su respuesta.

4.15. ¿Por qué es importante la retroalimentación para el equipo de software?

Lecturas y fuentes de información adicionales

La comunicación con el cliente es una actividad de importancia crítica en la ingeniería de software, pero pocos de sus practicantes dedican tiempo a leer sobre ella. Withall (*Software Requirements Patterns*, Microsoft Press, 2007) presenta varios patrones útiles que analizan problemas en la comunicación. Sutliff (*User-Centred Requirements Engineering*, Springer, 2002) se centra mucho en los retos relacionados con la comunicación. Los libros de Weigers (*Software Requierements*, 2a. ed., Microsoft Press, 2003), Pardee (*To Satisfy and Delight Your Customer*, Dorset House, 1996) y Karten [Kar94] analizan a profundidad los métodos para tener una interacción eficaz con el cliente. Aunque su libro no se centra en el software, Hooks y Farry (*Customer Centered Products*, American Management Association, 2000) presentan lineamientos generales útiles para la comunicación con los clientes. Young (*Effective Requirements Practices*, Addison-Wesley, 2001) pone el énfasis en un "equipo conjunto" de clientes y desarrolladores que recaben los requerimientos en colaboración. Somerville y Kotonya (*Requirements Engineering: Processes and Techniques*, Wiley, 1998) analizan el concepto de "provocación" y las técnicas y otros requerimientos de los principios de ingeniería.

Los conceptos y principios de la comunicación y planeación son estudiados en muchos libros de administración de proyectos. Entre los más útiles se encuentran los de Bechtold (*Essentials of Software Project Management*, 2a. ed., Management Concepts, 2007), Wysocki (*Effective Project Management: Traditional, Adaptive, Extreme*, 4a. ed., Wiley, 2006), Leach (*Lean Project Management: Eight Principles for Success*, BookSurge Publishing, 2006) Hughes (*Software Project Management*, McGraw-Hill, 2005) y Stellman y Greene (*Applied Software Project Management*, O'Reilly Media, Inc., 2005).

Davis [Dav95] hizo una compilación excelente de referencias sobre principios de la ingeniería de software. Además, virtualmente todo libro al respecto contiene un análisis útil de los conceptos y principios para análisis, diseño y prueba. Entre los más utilizados (además de éste, claro) se encuentran los siguientes:

Abran, A., y J. Moore, *SWEBOK: Guide to the Software Engineering Body of Knowledge*, IEEE, 2002.

Christensen, M., y R. Thayer, *A Project Manager's Guide to Software Engineering Best Practices*, IEEE-CS Press (Wiley), 2002.

Jalote, P., *An Integrated Approach to Software Engineering*, Springer, 2006.

Pfleeger, S., *Software Engineering: Theory and Practice*, 3a. ed., Prentice-Hall, 2005.

Schach, S., *Object- Oriented and Classical Software Engineering*, McGraw-Hill, 7a. ed., 2006.

Sommerville, I., *Software Engineering*, 8a. ed., Addison-Wesley, 2006

Estos libros también presentan análisis detallados sobre los principios de modelado y construcción.

Los principios de modelado se estudian en muchos libros dedicados al análisis de requerimientos o diseño de software. Los libros de Lieberman (*The Art of Software Modeling*, Auerbach, 2007), Rosenberg y Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice*, Apress, 2007), Roques (*UML in Practice,* Wiley, 2004) y Penker y Eriksson (*Business Modeling with UML: Business Patterns at Work*, Wiley, 2001) analizan los principios y métodos de modelado.

Todo ingeniero de software que trate de hacer diseño está obligado a leer el texto de Norman (*The Design of Everyday Things*, Currency/Doubleday, 1990). Winograd y sus colegas (*Bringing Design to Software*, Addison-Wesley, 1996) editaron una excelente colección de ensayos sobre aspectos prácticos del diseño de software. Constantine y Lockwood (*Software for Use,* Addison-Wesley, 1999) presenta los conceptos asociados con el "diseño centrado en el usuario". Tognazzini (*Tog on Software Design*, Addison-Wesley, 1995) presenta una reflexión filosófica útil sobre la naturaleza del diseño y el efecto que tienen las decisiones sobre la calidad y la capacidad del equipo para producir software que agregue mucho valor para su cliente. Stahl y sus colegas (*Model-Driven Software Development: Technology, Engineering*, Wiley, 2006) estudian los principios del desarrollo determinado por el modelo.

Son cientos los libros que abordan uno o más elementos de la actividad de construcción. Kernighan y Plauger [Ker78] escribieron un texto clásico sobre el estilo de programación, McConell [McC93] presenta lineamientos prácticos para la construcción de software, Bentley [Ben99] sugiere una amplia variedad de perlas de la programación, Knuth [Knu99] escribió una serie clásica de tres volúmenes acerca del arte de programar y Hunt [Hun99] sugiere lineamientos pragmáticos para la programación.

Myers y sus colegas (*The Art of Software Testing*, 2a. ed., Wiley, 2004) desarrollaron una revisión importante de su texto clásico y muchos principios importantes para la realización de pruebas. Los libros de Perry (*Effective Methods for Software Testing*, 3a. ed., Wiley 2006), Whittaker (*How to Break Software*, Addison-Wesley, 2002), Kaner y sus colegas (*Lessons Learned in Software Testing*, Wiley, 2001) y Marick (*The Craft of Software Testing*, Prentice-Hall, 1997) presentan por separado conceptos y principios importantes para hacer pruebas, así como muchas guías prácticas.

En internet existe una amplia variedad de fuentes de información sobre la práctica de ingeniería de software. En el sitio web del libro se encuentra una lista actualizada de referencias en la Red Mundial que son relevantes para la ingeniería de software: **www.mhe.com/engcs/compsci/pressman/professional/olc/ser.htm**