La refactorización del software es, por tanto, la mejor opción para abordar el mantenimiento del software ecológico, porque la refactorización es la única forma de tratar el código si queremos dotar al sistema de software de capacidades ecológicas. El punto clave ahora es averiguar qué se debe refactorizar y qué tipo de mejoras reducen el consumo de energía. Aplicar la refactorización para dotar a un sistema de capacidades ecológicas puede entenderse como reingeniería ecológica del software o *ingeniería ecológica del software*.

9.3.2 Malos olores

Los malos olores podrían entenderse como estilos de programación antiestéticos y estrategias de diseño deficientes que aparecen en el código fuente como consecuencia de un desarrollo rápido y descontrolado. En [6], el autor reúne un conjunto de olores de código que aparecen con frecuencia en los sistemas de software. La tabla 9.1 resume el conjunto de code smells presentado en [6], junto con una breve definición. El autor también considera un conjunto de posibles enfoques de refactorización para cada mal olor, pero esto va más allá del alcance de este capítulo.

En el mantenimiento clásico, la refactorización del software se lleva a cabo para resolver el problema de calidad que subyace a los malos olores. Como propone el autor, para cada mal olor, es posible aplicar una solución de refactorización para transformar el estado actual del sistema en otro equivalente de mejor calidad y sin el mal olor.

La cuestión es la siguiente: en el mantenimiento clásico, estas transformaciones son deseables y necesarias, pero no está claro hasta qué punto existe algún tipo de correlación entre el mantenimiento clásico y el ecológico que garantice una mejora de la ecologización del sistema. En este sentido, es necesaria la experimentación para establecer qué malos olores deben eliminarse y qué soluciones de refactorización deben aplicarse. La encrucijada mencionada en el apartado 9.5.3 establece que hay algunas mejoras de la mantenibilidad que son inversamente proporcionales a la mejora de la greenability.

Además del mal olor y la definición, la tabla 9.2 incluye una columna llamada *impacto*. El impacto hace referencia al efecto sobre la ecologización de la refactorización de un determinado mal olor. Como hemos dicho antes, no se ha investigado el efecto de la refactorización en la ecologización, por lo que los valores de la columna de *impacto* se han hipotetizado (es decir, es necesario experimentar para validarlos). Se han designado tres valores para predecir el impacto de una refactorización: (1) "+", cuando la refactorización puede tener un efecto positivo, mejorando la ecologización; (2) "0", cuando no está claro si la refactorización tiene un efecto positivo o negativo sobre la ecologización; y "-", cuando la refactorización empeora la ecologización.

Cuadro 9.2 Código de malos olores y posible impacto sobre la ecologabilidad

Mal olor	Definición	Impacto
Código duplicado	Un fragmento de código se repite una o varias veces	-
Método largo	Métodos con demasiadas variables, parámetros y código	0
Clase grande	Una clase con demasiadas variables de instancia. Demasiadas responsabilidades	-
Lista de parámetros larga	Método con una lista larga y comprensible de parámetros	0
Cambio divergente	El código es similar pero no igual	+
Cirugía con escopeta	Para un cambio determinado, son necesarios muchos cambios en otros lugares	+
Envidia de características	Un método de una clase está más interesado en otra clase que en la clase propietaria. El método *envidia* los datos del otro objeto	+
Grupos de datos	El mismo conjunto de datos se repite en distintos lugares (clases, métodos, parámetros, código, etc.).	-
Obsesión primitiva	Uso de tipos primitivos en lugar de objetos, lo que reduce la comprensibilidad	+
Declaraciones de cambio	La misma sentencia switch está dispersa en diferentes lugares	+
Jerarquías de herencia paralelas	Cuando se crea una subclase de una clase, también se debe crear una subclase de otra clase.	+
Clase perezosa	Una clase que no hace nada, excepto costar dinero para mantener y entender	+
Generalidad especulativa	Se incluyen demasiados tipos, ganchos y casos especiales en el código para una necesidad incierta en el futuro	+
Campo temporal	Las variables de instancia de una clase no siempre se instancian. Esto dificulta la comprensibilidad	0
Cadenas de mensajes	El cliente está acoplado a una larga y compleja estructura de navegación cuando solicita un objeto	+
Intermediario	Una clase delega gran parte de su comportamiento en una segunda clase	+
Intimidad inapropiada	Dos clases están estrechamente vinculadas	+
Clases alternativas con interfaces diferentes	Dos métodos que hacen lo mismo, con firmas diferentes. Dos clases que hacen algo muy parecido	0
Clase de biblioteca incompleta	Las bibliotecas no suelen contener toda la funcionalidad que necesitamos	0
Clase de datos	Clases sin responsabilidad y sólo campos con métodos get/set	+
Legado rechazado	Una clase no necesita todos los campos y métodos de una clase padre	+

9.3.3 Antipatrones

Según [2], los antipatrones son "una forma literaria que describe una solución común a un problema que genera consecuencias decididamente negativas. El antipatrón puede ser el resultado de que un gestor o desarrollador no sepa hacer nada mejor, no tenga suficientes conocimientos o experiencia en la resolución de un determinado tipo de

problema, o haber aplicado un patrón perfectamente bueno en el contexto equivocado". Estos antipatrones repercuten negativamente en la calidad del software y pueden empeorar su mantenimiento.

Los antipatrones también podrían ser una posible vía para abordar la ecologización del software. Encontrar y resolver los antipatrones del software mejora su calidad, pero la cuestión ahora es averiguar hasta qué punto la refactorización del software (para eliminar los antipatrones) también afecta a la ecologización.

En [2], los autores abordan los patrones desde tres perspectivas diferentes: (1) antipatrones de desarrollo de software o problemas técnicos y soluciones que introducen los programadores, (2) antipatrones arquitectónicos o problemas comunes en la forma en que se estructuran los sistemas, y (3) antipatrones de gestión o problemas en los procesos de software y las organizaciones de desarrollo. Desde el punto de vista de la ingeniería de software, el tipo más interesante de antipatrones de software es el primero, los antipatrones de desarrollo, porque representan problemas que deberían abordarse en el mantenimiento del software. Los antipatrones arquitectónicos también podrían considerarse interesantes porque la mayoría de sus consecuencias podrían tratarse en el código fuente (al menos aliviando sus síntomas). Por otro lado, el antipatrón de gestión no es útil si abordamos la ecologización desde el punto de vista de la ingeniería de software. Los antipatrones de gestión identifican problemas relacionados con los recursos humanos implicados en los proyectos de software. Aunque algunos antipatrones de gestión (los relacionados con los métodos de trabajo humanos) pueden estar directamente relacionados con la ecologización del software, no son el tema central de este capítulo. La Tabla 9.3 resume los antipatrones de desarrollo más comunes.

La tabla 9.4 presenta los antipatrones de arquitectura más comunes. Según [18], los anti-patrones de arquitectura "se centran en algunos problemas y errores comunes en la creación, implementación y gestión de la arquitectura."

Las tablas 9.3 y 9.4 incluyen una columna de *impacto* para representar el posible impacto de aplicar la refactorización del software para resolver los antipatrones. En este caso, hay cuatro valores posibles: "+" si la refactorización tiene un impacto positivo en la ecologización, "-" si la refactorización tiene un impacto negativo en la ecologización, "0" si no está claro si hay un impacto en la ecologización, y "NA" cuando no hay refactorización de software para resolver el antipatrón (es decir, está más relacionado con el proceso que con el producto).

Como señalamos con los olores de código, la predicción del impacto de la refactorización de los antipatrones es sólo una hipótesis. Es en este punto donde deben realizarse experimentos con cada antipatrón (y con combinaciones de ellos) para averiguar cuál de ellos sería adecuado para mejorar la ecologabilidad del software en un mantenance stage (i.e., green maintenance).

9.3.4 Consideraciones

A pesar de que hemos analizado por separado el posible impacto de los malos olores y los antipatrones en la ecologización del software, existe una importante

relación entre ambos conceptos que debe tenerse en cuenta. Por un lado, los antipatrones

Cuadro 9.2 Código de malos olores y posible impacto sobre la ecologabilidad

Tabla 9.3 Antipatrones de desarrollo de software

Antipatrón	Definición	Impacto
La mancha (*clase de dios*)	Una clase contiene la mayoría de las responsabilidades, mientras que las otras sólo contienen datos y pequeños procesamientos	-
Obsolescencia continua	La evolución de la tecnología dificulta a los desarrolladores la correcta interoperabilidad del software con otros productos.	0
Flujo de lava	El código muerto y el diseño olvidado se congelan en un diseño siempre cambiante	0
Descomposición funcional	Sistemas OO producidos por desarrolladores no orientados a objetos. El código orientado a objetos se parece al lenguaje estructural	+
Poltergeist	Clases de ciclo de vida muy corto. Estas clases suelen encargarse de iniciar procesos	+
Ancla de barco	Artefacto de software o hardware muy costoso pero sin ninguna utilidad	+
Martillo de oro	Los equipos de desarrollo aplican una y otra vez soluciones en las que tienen mucha experiencia, en lugar de explorar otras nuevas adecuadas	0
Callejón sin salida	Se modifica un componente reutilizable soportado por un vendedor o proveedor. Esto complica la integración de dichas modificaciones en las nuevas versiones.	0
Código espagueti	La estructura del software se hace de forma ad hoc, por lo que es difícil de ampliar y mantener	-
Entrada kludge	Los algoritmos ad hoc gestionan la entrada de programas	+
Caminando por un campo de minas	Los productos se lanzan demasiado pronto, y un número importante de errores están (muy probablemente) en el código	+
Programación de cortar y pegar	Copiar y pegar bloques de código fuente conlleva problemas de mantenimiento	-
Gestión de los hongos	Los desarrolladores están aislados del usuario final del sistema. Los requisitos se reciben indirectamente, a través de otros intermediarios	0

son decisiones de diseño comunes (pero indeseables) o malos diseños tomados por malos diseñadores o limitaciones en el proyecto. Por otro lado, los malos olores son un síntoma de que algo no funciona bien. En otras palabras, los malos olores son la evidencia de un posible antipatrón. Por ejemplo, si consideramos *el blob* (también conocido como *el antipatrón de la clase dios*), hay varios malos olores que podrían demostrar la existencia de un antipatrón: el *método largo*, la *clase grande* y la *clase de datos*. Ninguno de estos tres malos olores en un programa es prueba por sí mismo de la existencia *del antipatrón blob*, pero encontrar cualquiera de ellos es una razón para sospechar que este antipatrón existe realmente.

Tabla 9.4 Antipatrones de arquitectura de software

Antipatrón	Definición	Impacto
Estufa autogenerada	Un sistema local se migra a una arquitectura distribuida. Si el diseño sigue siendo el mismo, aparecen problemas, como la forma en que los datos se transferido	+
Jumble	Se mezclan elementos verticales y horizontales. El software es complejo y difícil de evolucionar y reutilizar.	-
Sistema de tubos de estufa	No existe abstracción ni documentación de los subsistemas. Su integración debe realizarse de forma ad hoc.	+
Cubra sus activos	Los requisitos están repartidos en "toneladas" de documentos. Los desarrolladores no saben qué hacer con semejante maraña de información.	NA
Fijación del proveedor	Un producto adopta una tecnología determinada y pasa a depender de las condiciones del proveedor. Los problemas surgen cuando el producto se actualiza	+
Billete de lobo	Un producto que cumple las normas de software pero cuyas interfaces pueden variar con respecto a las normas publicadas.	0
Arquitectura por implicación	Los arquitectos demasiado confiados creen que cierta documentación arquitectónica importante no es necesaria. Tienen mucha experiencia y luego consideran que algo no es necesario, ya que permanece en su mente. El desarrollo no es posible sin esa información en forma de documento	0
Cuerpos calientes	Se asignan muchos programadores a un proyecto, pero sólo unos pocos son desarrolladores de calidad	NA
Diseño por comité	Un diseño de software complejo suele ser desarrollado por un comité de expertos. Las opiniones diferentes y democráticas dan lugar a diseños complejos y difíciles de aplicar	NA
Navaja suiza	Interfaz de clase excesivamente compleja. La clase intenta servir para demasiados usos	-
Reinventar la rueda	Falta de transferencia tecnológica entre un proyecto y otro nuevo. Se aprovecha al máximo la ventaja de tener conocimientos de diseño	0
El Gran Duque de York	El talento de las personas no se tiene en cuenta a la hora de definir la arquitectura del sistema. El talento de las personas es importante para decidir en qué fase de desarrollo del software interviene una determinada persona en el el proyecto debe funcionar	0

9.4 Deuda técnica y deuda ecológica

La deuda técnica puede definirse como "la escritura de código inmaduro o no del todo correcto con el fin de lanzar un nuevo producto al mercado más rápidamente"; podemos encontrarla de muchas formas (proceso, alcance, pruebas y diseño) [4, 16]. La deuda técnica también podría entenderse como el "resultado invisible de decisiones pasadas sobre el software que afectan al futuro" [14]. La ecuación (9.1) presenta una ecuación muy sencilla que resume cómo podría calcularse la deuda técnica . En esta fórmula, el concepto de *fallo tecnológico* representa cualquier tipo de mal olor, antipatrón o falta de documentación o de casos de prueba, por ejemplo:

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.

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.