

GUIA DE PREGUNTAS

Material "MANTENIMIENTO DE SOFTWARE.
Ingeniería del Software (Capítulo 20) de Roger S. Pressman (3^{ra} Edición)"

1. Defina mantenimiento correctivo.
2. Defina mantenimiento adaptativo.
3. Defina los dos tipos de mantenimiento perfectivo.
4. Enuncie puntos de vista para comprender las características del mantenimiento de software.
5. Esquematice el flujo de sucesos que se pueden dar como resultado de una petición de mantenimiento de software.
6. Defina mantenimiento estructurado.
7. Ejemplifique costes intangibles del mantenimiento del software
8. Plantee un modelo del esfuerzo de mantenimiento.
9. Enuncie problemas asociados con el mantenimiento de software.
10. Defina facilidad de mantenimiento del software.
11. Enuncie factores de control relacionados con el entorno de desarrollo.
12. Enuncie métricas de la facilidad de mantenimiento.
13. De un esquema y describa el funcionamiento de una organización de mantenimiento.
14. Defina formulario de petición de mantenimiento (informe de problemas del software).
15. Defina que informe de cambios de software y que se informa en él.
16. De un esquema del flujo de sucesos del mantenimiento.
17. Enuncie preguntas de repaso de la situación pertinentes una vez concluida la tarea de mantenimiento.
18. Enuncie información a registrar concluida la tarea de mantenimiento.
19. Enuncie medidas del rendimiento del mantenimiento.
20. Enuncie cambios de código que tienen mayor probabilidad de inducir a error.
21. Enuncie cambios en los datos que tienen mayor probabilidad de inducir a error.
22. Enuncie acciones a desarrollar cuando se deba realizar mantenimiento de código "ajeno".
23. Defina ingeniería inversa.
24. Defina reingeniería.
25. Enuncie puntos a considerar antes de redesarrollar un sistema software.
26. Defina nivel de abstracción de un proceso de ingeniería inversa.
27. Defina completitud de un proceso de ingeniería inversa e interactividad en su contexto.
28. Defina direccionalidad de un proceso de ingeniería inversa.

CAPITULO 20

MANTENIMIENTO DEL SOFTWARE

El *mantenimiento del software* ha sido caracterizado [CAN72] como un “iceberg”: esperamos que lo que está inmediatamente visible sea todo lo que hay. Realmente, sabemos que bajo la superficie se esconde una gran cantidad de problemas potenciales y de costes. El mantenimiento del software existente puede llevarse hasta el 70 por 100 de todo el esfuerzo gastado por una organización de desarrollo. El porcentaje sigue subiendo a medida que se produce más software. En el horizonte podemos prever una “barrera de mantenimiento” para las organizaciones de desarrollo de software, que no podrán producir nuevo software debido a que gastarán todos los recursos disponibles en el mantenimiento del antiguo software.

Los lectores que no conozcan el tema se pueden preguntar por qué se requiere tanto mantenimiento y por qué se desperdicia tanto esfuerzo. Una respuesta parcial nos la da Osborne y Chikofsky [OSB90]:

Una gran cantidad del software del que dependemos actualmente tiene una media de edad de entre 10 y 15 años. Incluso aunque esos programas se hubieran creado con las mejores técnicas de diseño y codificación existentes en su momento [y muchos no lo fueron], fueron creados teniendo como principales requisitos el tamaño del programa y el espacio de almacenamiento. Luego, migraron a plataformas nuevas, fueron ajustados a cambios en las máquinas y en los sistemas operativos y mejorados para satisfacer las necesidades de sus nuevos usuarios —todo ello sin tener en cuenta su arquitectura general.

El resultado ha sido la existencia de sistemas de software con unas estructuras de datos pobremente diseñadas, una pobre codificación, una lógica pobre y una documentación pobre, teniendo que seguir funcionando en la actualidad...

La naturaleza omnipresente del cambio mediatiza todo el trabajo en software. El cambio es inevitable en la construcción de sistemas basados en computadora;

por ello, debemos desarrollar mecanismos de evaluación, control e implementación de modificaciones.

A lo largo de este libro hemos tratado el proceso de la ingeniería del software. Un fin primordial de este proceso es el de mejorar la facilidad con la que se puedan acomodar los cambios y reducir la cantidad de esfuerzo empleado en el mantenimiento. En este capítulo y en el que le sigue trataremos las actividades específicas que nos permiten alcanzar este fin.

20.1. UNA DEFINICION DE MANTENIMIENTO DEL SOFTWARE

Tras leer la introducción de este capítulo, un lector podría protestar: "...pero yo no empleo el 60 por 100 de mi tiempo solucionando errores del programa que desarrollo". El mantenimiento del software es, por supuesto, mucho más que una "corrección de errores". Podemos describir el mantenimiento describiendo las cuatro actividades que se llevan a cabo tras distribuir un programa.

La primera actividad de mantenimiento es debida a que no es razonable asumir que la prueba del software haya descubierto todos los errores latentes de un gran sistema de software. Durante el uso de cualquier gran programa, se encontrarán errores, siendo informado el equipo de desarrollo. El proceso que incluye el diagnóstico y la corrección de uno o más errores se denomina *mantenimiento correctivo*.

La segunda actividad que contribuye a la definición de mantenimiento se produce por el rápido cambio inherente a cualquier aspecto de la informática. Se anuncian nuevas generaciones de hardware en ciclos de unos 24 meses; regularmente, aparecen nuevos sistemas operativos o nuevas versiones de los antiguos; frecuentemente, se mejoran o modifican los equipos periféricos y otros elementos de los sistemas. Por otro lado, la vida útil del software de aplicación puede fácilmente superar los diez años, sobreviviendo al entorno del sistema para el que fue originalmente desarrollado. Por tanto, el *mantenimiento adaptativo* —una actividad que modifica el software para que interaccione adecuadamente con su entorno cambiante— es tan necesario como usual.

La tercera actividad que se puede aplicar a la definición de mantenimiento se produce cuando un paquete de software tiene éxito. A medida que se usa el software, se reciben de los usuarios recomendaciones sobre nuevas posibilidades, sobre modificaciones de funciones ya existentes y sobre mejoras en general. Para satisfacer estas peticiones, se lleva a cabo el *mantenimiento perfectivo*. Esta actividad contabiliza la mayor cantidad de esfuerzo empleado en el mantenimiento de software.

La cuarta actividad de mantenimiento se da cuando se cambia el software para mejorar una futura facilidad de mantenimiento o fiabilidad, o para proporcionar una base mejor para futuras mejoras. A menudo denominada *mantenimiento perfectivo*, esta actividad está caracterizada por las técnicas de *ingeniería inversa* y de *reingeniería*, tratadas más adelante en este capítulo.

Los términos usados para describir las tres primeras actividades de mantenimiento fueron introducidos por Swanson [SWA76]. El cuarto término se usa normalmente en el mantenimiento de hardware y de otros sistemas físicos. Sin embargo, se debe insistir en que se pueden malinterpretar las analogías entre el mantenimiento de software y el de hardware. Como ya indicamos en el primer capítulo de este libro, el software, a diferencia del hardware, no se desgasta y, por tanto, la actividad principal asociada con el mantenimiento de hardware —el reemplazamiento de piezas estropeadas o rotas— simplemente no se aplica aquí.

Algunos profesionales del software están preocupados por la inclusión de la segunda y la tercera actividad dentro de la definición de mantenimiento. En realidad, las tareas que se dan como parte del mantenimiento adaptativo o del perfectivo son las mismas que se aplican durante la fase de desarrollo del proceso de ingeniería del software. Para adaptar o perfeccionar, debemos determinar nuevos requisitos, rediseñar, generar código y probar el software existente. Tradicionalmente, tales tareas, cuando se aplican a un programa existente, han sido denominadas mantenimiento.

20.2. CARACTERISTICAS DEL MANTENIMIENTO

El mantenimiento de software ha sido hasta hace muy poco la fase negra del proceso de ingeniería del software. Los libros sobre mantenimiento contienen muy pocos epígrafes, a diferencia de los de actividades de desarrollo. Se ha llevado a cabo relativamente poca investigación o producción de datos sobre el tema y se han propuesto pocos enfoques o “métodos” técnicos.

Para comprender las características del mantenimiento de software, consideraremos el asunto desde tres puntos de vista diferentes:

1. Las actividades requeridas para cubrir la fase de mantenimiento y el impacto de un enfoque de ingeniería del software (o de su ausencia) sobre la eficacia de tales actividades.
2. Los costes asociados con la fase de mantenimiento.
3. Los problemas que se encuentran frecuentemente cuando se lleva a cabo el mantenimiento.

En las secciones que siguen, se describen las características del mantenimiento desde cada una de las perspectivas que se acaban de enumerar.

20.2.1. Mantenimiento estructurado frente al no estructurado

En la Figura 20.1 se muestra el flujo de sucesos que se pueden dar como resultado de una petición de mantenimiento. Si sólo se dispone del código fuente como elemento de configuración, la actividad de mantenimiento comienza con una dolorosa evaluación del código, a menudo complicada por la pobre docu-

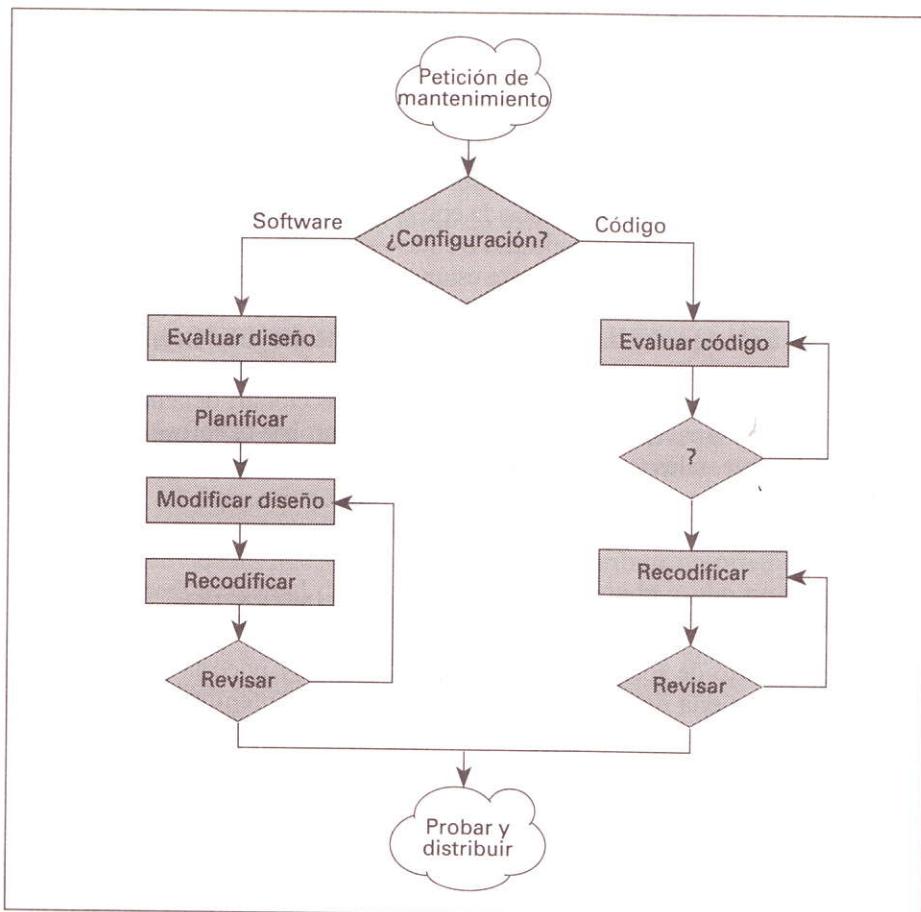


Figura 20.1. Mantenimiento estructurado frente al no estructurado.

mentación interna¹. Las delicadas características, tales como la estructura del programa, las estructuras de datos globales, las interfaces del sistema, el rendimiento y/o las limitaciones del diseño, son difíciles de descubrir y, frecuentemente, mal interpretadas. Es difícil asegurar cuáles son las ramificaciones de los cambios realizados finalmente sobre el código. Es imposible llevar a cabo pruebas de regresión (repetición de pruebas anteriores para asegurar que las modificaciones no han introducido fallos en el software previamente operativo), ya que no existe ningún registro de pruebas. Lo que hacemos es un *mantenimiento no estructurado* y pagamos el precio (en esfuerzo desperdiciado y en frustracio-

¹ Una nueva generación de herramientas de ingeniería inversa (tratadas más adelante) facilitan la evaluación del código. Sin embargo, esas herramientas no eliminan la necesidad de una buena documentación de diseño.

nes personales) asociado a todo software que no haya sido desarrollado mediante una metodología bien definida.

Si existe una completa configuración del software, la tarea de mantenimiento comienza con una evaluación de la documentación del diseño. Se determinan las importantes características estructurales, de rendimiento y de interfaz del software. Se estudia el impacto de las correcciones o modificaciones requeridas y se traza un plan de actuación. Se modifica el diseño (usando técnicas idénticas a las discutidas en capítulos anteriores) y se revisa. Se desarrolla nuevo código fuente, se realizan pruebas de regresión mediante la información contenida en la *especificación de prueba* y se vuelve a lanzar el software.

Esta secuencia de sucesos constituye el *mantenimiento estructurado* y aparece como resultado de una anterior aplicación de una metodología de ingeniería del software. Aunque la existencia de una configuración del software no garantiza un mantenimiento libre de problemas, sí que reduce la cantidad de esfuerzo requerido y mejora la calidad general del cambio o de la corrección.

20.2.2. Costes del mantenimiento

El coste del mantenimiento de software ha crecido rápidamente durante los últimos veinte años. Durante los años 70, el mantenimiento constituía entre el 35 y el 40 por 100 del presupuesto de una organización de sistemas de información. Ese número ascendió hasta el 60 por 100 durante los años 80. Si no se hace algo para mejorar las técnicas de mantenimiento, a mediados de los 90 muchas compañías gastarán cerca del 90 por 100 del presupuesto en el mantenimiento.

Los costes del mantenimiento en dinero es lo que obviamente más nos interesa. Sin embargo, otros costes, menos tangibles, pueden, en último lugar, ser la causa de muchas preocupaciones. Citando a Daniel McCracken [MCC80]:

Los retrasos en las nuevas aplicaciones y en los grandes cambios medidos en años se van alargando. Como industria, ni siquiera podemos hacer frente a —no digamos sólo ponernos al día con— lo que nuestros usuarios quieren que hagamos.

McCracken alude a la organización situada en la barrera del mantenimiento. Un coste intangible del mantenimiento del software se encuentra en una oportunidad de desarrollo que se ha de posponer o que se pierde, debido a que los recursos disponibles deben estar dedicados a las tareas de mantenimiento. Otros costes intangibles son:

- Insatisfacción del cliente cuando una petición de reparación o de modificación aparentemente legítima no se puede atender en un tiempo razonable.
- Disminución de la calidad global del software debido a los errores latentes que introducen los cambios en el software mantenido.
- Trastornos en otros esfuerzos de desarrollo al tener que “poner” a trabajar a la plantilla en tareas de mantenimiento.

El coste final del mantenimiento de software es una drástica reducción de la productividad (medida en LDC por persona/mes o en puntos de función por persona/mes) que se produce cuando se inicia el mantenimiento de viejos programas. Se ha informado de reducciones de la productividad de 40 a 1 [BOE79]. Es decir, un esfuerzo de desarrollo que haya resultado en un coste de 2 500 ptas. por línea de código desarrollada, costaría 100 000 ptas. por cada línea de código que sea mantenida.

El esfuerzo empleado en el mantenimiento se puede dividir en actividades productivas (p. ej.: análisis y evaluación, modificación del diseño, codificación) y actividades “menos productivas” (p. ej.: tratar de comprender lo que es el código; tratar de interpretar las estructuras de datos, las características de la interfaz, los límites de rendimiento). La siguiente expresión [BEL72] proporciona un modelo del esfuerzo de mantenimiento:

$$M = p + Ke^{(c-f)}$$

donde M = esfuerzo total empleado en el mantenimiento

p = esfuerzo productivo (tal como se ha descrito)

K = una constante empírica

c = una medida de la complejidad que se puede atribuir a la falta de un buen diseño y de documentación

f = una medida del grado de familiaridad con el software.

Este modelo indica que el esfuerzo (y el coste) puede aumentar exponencialmente si se usa un pobre enfoque de desarrollo de software (o sea, una falta de ingeniería del software) y si la persona o grupo que usó ese enfoque no está disponible para llevar a cabo el mantenimiento.

20.2.3. Problemas

La mayoría de los problemas asociados con el mantenimiento de software se debe a las deficiencias de la forma en que el software ha sido definido y desarrollado. Aquí aparece el clásico síndrome del “pague ahora o pague después”. La falta de control y disciplina en las actividades de desarrollo del proceso de ingeniería del software casi siempre se traduce en problemas para el mantenimiento del software.

Entre los muchos problemas clásicos [SCH87] asociados con el mantenimiento de software se encuentran los siguientes:

- A menudo es difícil o imposible seguir la evolución del software a través de varias versiones. Los cambios no están adecuadamente documentados.
- A menudo es difícil o imposible seguir el proceso por el que se construyó el software.
- A menudo es excepcionalmente difícil comprender un programa “ajeno”. A medida que existen menos elementos de configuración del software,

mayor es la dificultad. Si sólo existe el código indocumentado, es de esperar que aparezcan serios problemas.

- Ese personaje “ajeno”, a menudo, no se encuentra cerca para que pueda explicar lo que hizo. La movilidad entre los profesionales del software es actualmente muy grande. No podemos esperar una explicación personal del software por el que lo ha desarrollado una vez que se requiere el mantenimiento.
- No existe una documentación apropiada o está mal preparada. Un primer paso es el reconocimiento de que el software debe ser documentado, pero para que la documentación sea de algún valor debe ser comprensible y consistente con el código fuente.
- La mayoría del software no ha sido diseñado previendo el cambio. A menos que el método de diseño prevea el cambio mediante conceptos tales como la independencia funcional o las clases de objetos, las modificaciones del software serán difíciles y propensas a errores.
- El mantenimiento no se ve como un trabajo atractivo. Muchas de las causas de esto vienen dadas por el alto nivel de frustración asociado con el trabajo de mantenimiento.

Todos los problemas que se acaban de describir se pueden, en parte, atribuir al gran número de programas actualmente existentes que han sido desarrollados sin tener en cuenta la ingeniería del software. Tampoco se debe ver como una panacea el uso de una metodología disciplinada. Sin embargo, la ingeniería del software, por lo menos, proporciona soluciones parciales a cada problema asociado con el mantenimiento.

Como consecuencia de los problemas asociados con el mantenimiento de software, surgen varias cuestiones técnicas y de organización. ¿Es posible desarrollar software que esté bien diseñado y sea fácil de mantener? ¿Podemos mantener la integridad del software cuando tiene que ser modificado? ¿Existen enfoques técnicos y organizativos que se puedan aplicar con éxito al mantenimiento de software? En las secciones que vienen a continuación se tratan estos y otros puntos.

20.3. FACILIDAD DE MANTENIMIENTO

Las características descritas en la sección anterior están todas afectadas por la *facilidad de mantenimiento* del software. La facilidad de mantenimiento se puede definir cualitativamente como la facilidad de comprender, corregir, adaptar y/o mejorar el software. Como ya hemos insistido a lo largo del libro, la facilidad de mantenimiento es un fin clave que guía los pasos de cualquier metodología de ingeniería del software.

20.3.1. Factores de control

La facilidad de mantenimiento que se consiga para el software se ve afectada por muchos factores. Una falta de cuidado en el diseño, en la codificación o en

la prueba tiene un impacto obviamente negativo sobre nuestra capacidad de mantener fácilmente el software. Una pobre configuración del software puede tener un similar impacto negativo incluso cuando se hayan seguido cuidadosamente los ya mencionados pasos técnicos.

Además de los factores que se pueden asociar con la metodología de desarrollo, Kopetz [KOP79] define varios factores relacionados con el entorno de desarrollo:

- Disponibilidad de una plantilla de software cualificada.
- Estructura del sistema comprensible.
- Facilidad de manejo del sistema.
- Uso de lenguajes de programación estandarizados.
- Uso de sistemas operativos estandarizados.
- Estructura de la documentación estandarizada.
- Disponibilidad de los casos de prueba.
- Facilidades de depuración incorporadas.
- Disponibilidad de una computadora apropiada para llevar a cabo el mantenimiento.

Además de estos factores, podríamos añadir (de forma un poco jocosa): la disponibilidad de la persona o grupo que haya desarrollado originalmente el software.

Muchos de los factores anteriormente establecidos reflejan las características de los recursos de software y de hardware que se usan durante el desarrollo. Por ejemplo, no hay duda de que la ausencia de compiladores de lenguajes de alto nivel (con la consecuente necesidad de uso del lenguaje ensamblador) tiene un efecto perjudicial sobre la facilidad de mantenimiento. Otros factores indican la necesidad de la estandarización de los métodos, de los recursos y de los enfoques. Probablemente, el factor más importante que afecta a la facilidad de mantenimiento es la planificación condicionada al mantenimiento. Si se ve el software como un elemento del sistema que inevitablemente estará sujeto a cambios, serán sustancialmente mayores las posibilidades de producir un software fácilmente mantenable.

20.3.2. Medidas cuantitativas

La facilidad de mantenimiento del software, como la calidad o la fiabilidad, es un término difícil de cuantificar. Sin embargo, podemos evaluar la facilidad de mantenimiento indirectamente, considerando los atributos de la actividad de mantenimiento que se pueden medir. Gilb [GIL79] da un número de *métricas de la facilidad de mantenimiento* relacionadas con el esfuerzo gastado durante el mantenimiento:

1. Tiempo de reconocimiento del problema.
2. Tiempo de retraso administrativo.

3. Tiempo de recolección de herramientas de mantenimiento.
4. Tiempo de análisis del problema.
5. Tiempo de especificación de los cambios.
6. Tiempo activo de corrección (o modificación).
7. Tiempo de prueba local.
8. Tiempo de prueba global.
9. Tiempo de revisión del mantenimiento.
10. Tiempo total de recuperación.

Cada una de las anteriores métricas puede, de hecho, ser registrada sin mucha dificultad. Tales datos pueden proporcionar a un gestor una indicación de la eficacia de las nuevas técnicas y herramientas.

Además de esas medidas orientadas al tiempo, se puede medir indirectamente la facilidad de mantenimiento considerando medidas de la estructura del diseño (Capítulo 10) y métricas de la complejidad del software (Capítulo 17). Varios estudios industriales (p. ej.: [KAF87], [ROM87]) han detectado la fuerte correlación entre la complejidad y estructura del programa y la facilidad de mantenimiento resultante para el programa.

20.3.3. Revisiones

Debido a que la facilidad de mantenimiento debe ser una característica esencial de cualquier software, debemos asegurarnos de que los factores señalados en la Sección 20.3.1 sean incorporados durante la fase de desarrollo. En cada nivel del proceso de revisión de la ingeniería del software ha de ser considerada la facilidad de mantenimiento. Durante la revisión de los requisitos hay que anotar las áreas de futuras mejoras y las posibles revisiones; hay que discutir la portabilidad del software y hay que considerar las interfaces del sistema que puedan tener impacto sobre el mantenimiento del software. Durante las revisiones del diseño debe evaluarse el diseño de datos, el diseño arquitectónico y el diseño procedimental, para que sea fácil de modificar, sea modular y funcionalmente independiente. En las revisiones del código se debe cuidar el estilo y la documentación interna, dos factores que tienen influencia sobre la facilidad de mantenimiento. Finalmente, cada paso de prueba debe proporcionar anotaciones sobre las partes del programa que puedan necesitar un mantenimiento preventivo antes de que el software sea formalmente lanzado.

La revisión de mantenimiento más formal se da al terminar la prueba y se denomina revisión de la configuración. La revisión de la configuración, tratada en el siguiente capítulo, asegura que todos los elementos de la configuración del software son completos, comprensibles y dispuestos para un control de modificación.

La propia tarea de mantenimiento debe ser revisada tras terminar cada proyecto de software. En la siguiente sección se tratan distintos métodos de evaluación.

20.4. TAREAS DE MANTENIMIENTO

Las tareas asociadas con el mantenimiento del software comienzan mucho antes de que se haga una petición de mantenimiento. Inicialmente, se debe establecer una organización de mantenimiento (de hecho o formal); se deben prescribir procedimientos de evaluación y de información, y se debe definir una secuencia estandarizada de sucesos para cada petición de mantenimiento. Además, se debe establecer un sistema de registro de información de las actividades de mantenimiento y definir criterios de revisión y de evaluación.

20.4.1. Una organización de mantenimiento

Como ya hemos indicado anteriormente, existen casi tantas estructuras organizativas como organizaciones de desarrollo de software. Por esta razón, en este libro se evitan las “estructuras organizativas recomendadas” para el desarrollo de software y para el mantenimiento de software. Sin embargo, en el caso del mantenimiento, raramente existen organizaciones formales (con las notables excepciones de los desarrollos de software muy grandes) de modo que el mantenimiento se lleva a cabo “como se pueda”.

Aunque no hay necesidad de establecer una organización de mantenimiento formal, una delegación informal de responsabilidades es absolutamente esencial, incluso para grupos pequeños de desarrollo. En la Figura 20.2 se muestra

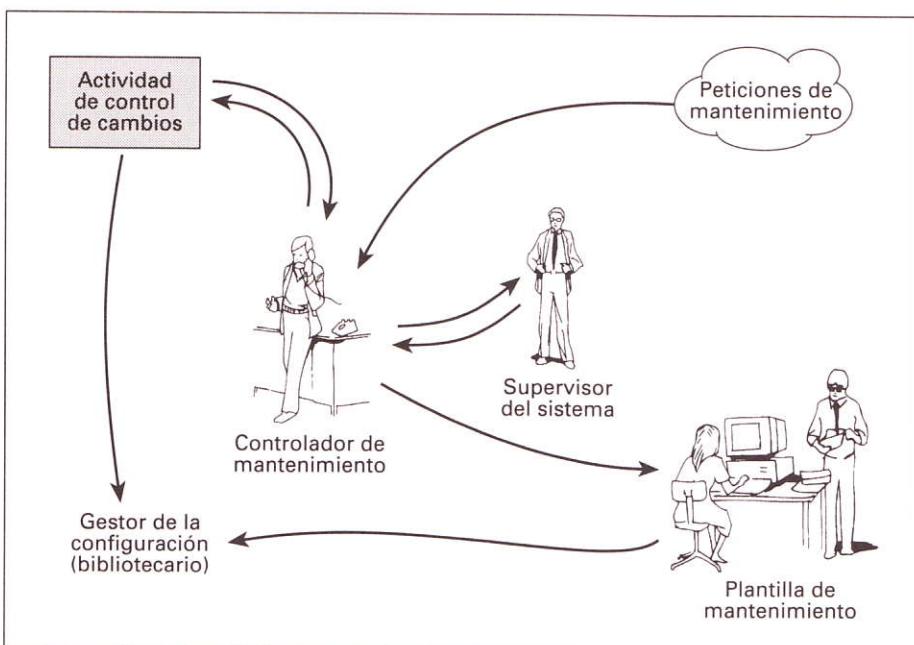


Figura 20.2. Organización.

uno de esos esquemas. Las peticiones de mantenimiento se dirigen hacia un *controlador de mantenimiento* que remite cada petición a un *supervisor del sistema* para que la evalúe. El supervisor del sistema es un miembro de la plantilla técnica al que se asigna la responsabilidad de familiarizarse con un pequeño subconjunto de programas producidos. Una vez que se hace la evaluación, una autoridad de control de cambios (a veces denominada *consejo de control de cambios*) debe determinar las acciones a llevar a cabo.

La organización sugerida reduce la confusión y mejora el flujo de las actividades de mantenimiento. Dado que todas las peticiones de mantenimiento se canalizan a través de un solo individuo (o grupo), los “arreglos descontrolados” (es decir, los cambios que no han sido sancionados y, por tanto, pueden causar confusión) son menos probables. Como por lo menos un individuo estará familiarizado con un programa producido, las peticiones de cambio (mantenimiento) se pueden evaluar más rápidamente. Al implementar un mecanismo de control de cambios² para aprobar los cambios específicos, la organización puede evitar realizar cambios que beneficien a un solicitante pero que tengan un impacto negativo sobre otros muchos usuarios.

Cada uno de los anteriores puestos de trabajo sirve para establecer un área de responsabilidad en el mantenimiento. El controlador y la autoridad de control de cambios puede ser una misma persona o (para grandes programas) un grupo de jefes y de individuos de la plantilla técnica. El supervisor del sistema puede tener otras obligaciones sin dejar de constituir un “contacto” con los paquetes específicos de software.

Cuando se asignan responsabilidades antes de comenzar la actividad de mantenimiento, se reduce enormemente la confusión. Y, más importante, la pronta definición de responsabilidades puede moderar cualquier sentimiento de incomodidad que aparece cuando una persona es “sacada” atropelladamente de una tarea de desarrollo para que se ocupe del mantenimiento.

20.4.2. Informes

Todas las peticiones de mantenimiento de software deben ser presentadas de una forma estandarizada. Normalmente, el equipo de desarrollo de software genera un *formulario de petición de mantenimiento* (FPM), a veces denominado *informe de problemas del software*, que ha de ser llenado por el usuario que desea la actividad del mantenimiento. Si se encuentra un error, se debe incluir una completa descripción de las circunstancias que llevaron al error (incluyendo datos de entrada, listados y otro material de soporte). Para peticiones de mantenimiento adaptativo o perfectivo, se ha de remitir una breve especificación de cambios (una especificación de requisitos abreviada). El formulario de petición de mantenimiento será evaluado en la forma descrita en la anterior sección.

El FPM es un documento generado exteriormente que se usa como base para la planificación de las tareas de mantenimiento. Internamente, la organización

² Este asunto se trata con más detalle en el Capítulo 21.

de software desarrolla un *informe de cambios del software* (ICS), indicando (1) la magnitud del esfuerzo requerido para satisfacer el FPM; (2) la naturaleza de las modificaciones requeridas; (3) la prioridad de la petición y (4) otros datos sobre las modificaciones. El ICS se envía a la autoridad de control de cambios antes de iniciar cualquier planificación del mantenimiento.

20.4.3. Flujo de sucesos

La Figura 20.3 muestra la secuencia de sucesos que se produce como resultado de una petición de mantenimiento. El primer requisito es determinar el tipo de mantenimiento que se va a llevar a cabo. En muchos casos un usuario puede ver una petición como una indicación de un error del software (mantenimiento correctivo), mientras que el equipo de desarrollo puede ver la misma petición como una adaptación o una mejora. Si existe diversidad de opiniones, se debe negociar un acuerdo.

De conformidad con el flujo que se muestra en la Figura 20.3, una petición de mantenimiento correctivo (camino de *error*) comienza con una evaluación de la severidad del error. Si existe un error serio (p. ej.: una función crítica del sistema no funciona), el personal es asignado a las directrices del supervisor del sistema y el análisis del problema comienza inmediatamente. Para los errores menos serios, se evalúa y se clasifica la petición de mantenimiento correctivo para luego planificarla de acuerdo con otras tareas que precisen los recursos de desarrollo.

En algunos casos, el error puede ser tan serio que haya que abandonar momentáneamente los controles normales del mantenimiento. El código habrá de ser modificado inmediatamente, sin la correspondiente evaluación de los posibles efectos secundarios y sin una adecuada actualización de la documentación. Este modo de *actuación inmediata* para el mantenimiento correctivo queda reservado únicamente para situaciones de “crisis” y debe representar sólo un pequeño porcentaje de todas las actividades de mantenimiento. Se debe insistir en que la actuación inmediata pospone, pero no elimina, la necesidad de controles y evaluaciones. Una vez que se ha resuelto la crisis, se deben conducir esas actividades, para asegurar que los recientes arreglos no propagarán problemas incluso más serios.

Las peticiones de mantenimiento adaptativo y de mantenimiento perfectivo siguen un camino diferente. Las adaptaciones se evalúan y se clasifican (se asigna prioridad) antes de situarlas en una cola de acciones de mantenimiento. Las mejoras siguen la misma evaluación. Sin embargo, no todas las peticiones de mejoras se llevan a cabo. La estrategia comercial, los recursos disponibles, la línea de los productos de software actuales y futuros y otros muchos aspectos pueden hacer que se rechace una petición de mejora. Las mejoras que sean aceptadas también se sitúan en una cola de mantenimiento. Se establece la prioridad de cada petición y se planifica el trabajo requerido como si se tratara de otro esfuerzo de desarrollo más (para todos los propósitos e intenciones así es). Si se fija una prioridad extremadamente alta, el trabajo debe empezar inmediatamente.

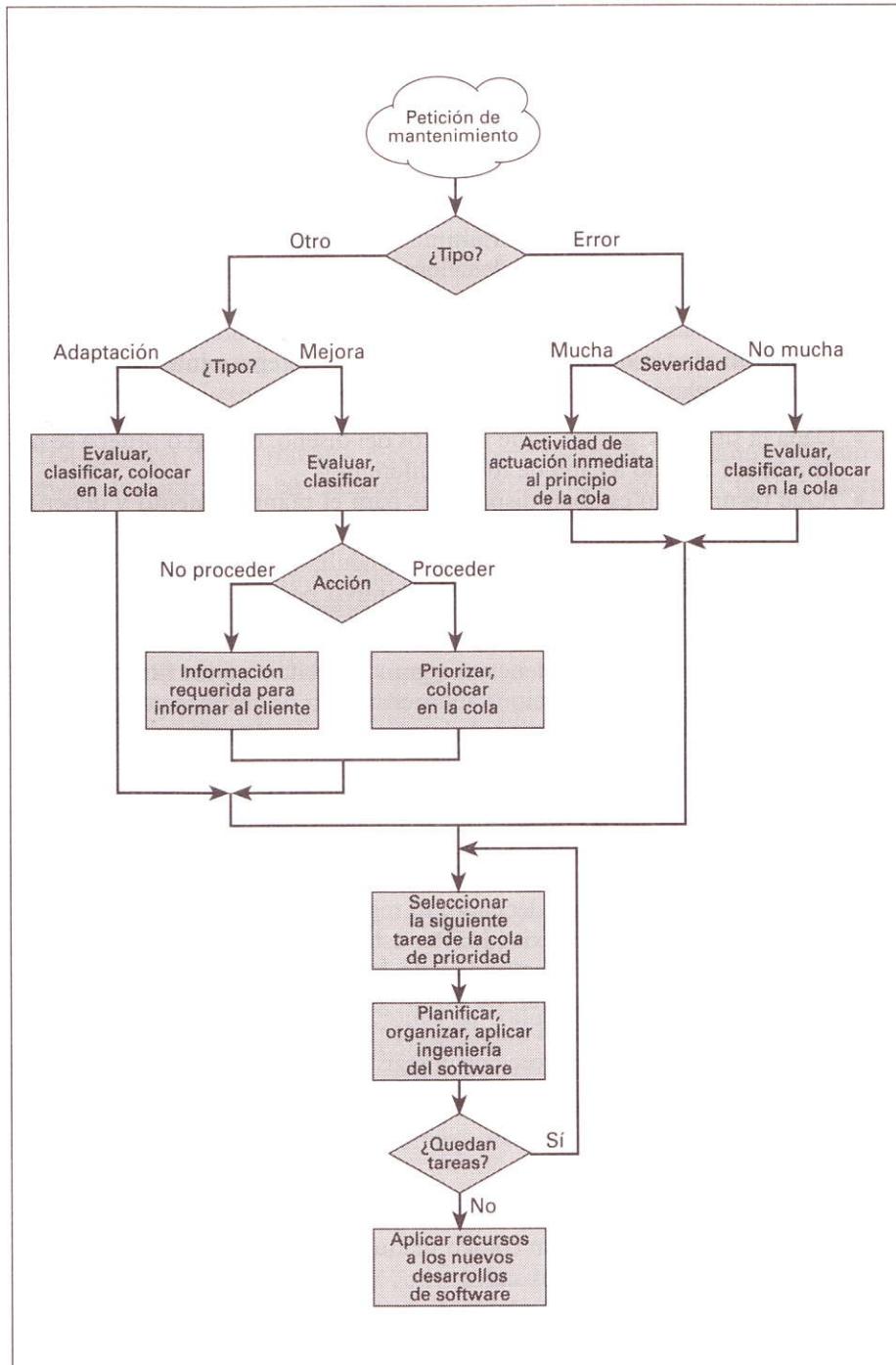


Figura 20.3. Flujo de sucesos del mantenimiento.

Independientemente del tipo de mantenimiento, se siguen las mismas tareas técnicas. Estas tareas de mantenimiento incluyen la modificación del diseño del software, la revisión, las modificaciones oportunas en el código, la prueba de unidad y de integración (incluyendo pruebas de regresión usando los casos de prueba ya existentes), la prueba de validación y la revisión. De hecho, el mantenimiento de software es realmente una aplicación recursiva de la ingeniería del software. El énfasis variará dependiendo de cada tipo de mantenimiento, pero el enfoque general será el mismo. El último suceso del flujo de mantenimiento es una revisión que revalide todos los elementos de la configuración del software y asegure que, de hecho, se ha satisfecho el FPM.

Una vez terminada la tarea de mantenimiento, a menudo es una buena idea llevar a cabo un repaso de la situación. En general, el repaso intenta responder a las siguientes preguntas:

- Dada la situación actual, ¿qué aspectos del diseño, código o prueba se podría haber llevado a cabo de forma diferente?
- ¿Qué recursos no estaban disponibles para el mantenimiento y deberían haberlo estado?
- ¿Cuáles fueron las mayores (menores) dificultades del esfuerzo?
- ¿Es adecuado un mantenimiento preventivo por los tipos de solicitudes dadas?

El repaso de la situación puede tener una gran influencia en los futuros esfuerzos de mantenimiento y proporciona un mecanismo de realimentación que es importante para una gestión efectiva de la organización de software.

20.4.4. Registro de información

Históricamente, el registro de información de todas las fases del proceso de ingeniería del software ha sido inadecuado. El registro de información de mantenimiento prácticamente no ha existido. Por esta razón, frecuentemente, no somos capaces de asegurar la efectividad de las técnicas de mantenimiento, incapaces de determinar la calidad de un programa de producción y de determinar lo que cuesta realmente el mantenimiento.

El primer problema que aparece con el registro de información es el de comprender qué datos merece la pena registrar. Swanson [SWA76] da una extensa lista:

1. Identificación del programa.
2. Número de sentencias fuente.
3. Número de instrucciones en código máquina.
4. Lenguaje de programación usado.
5. Fecha de instalación del programa.
6. Número de ejecuciones del programa desde la instalación.
7. Número de fallos de procesamiento asociados con el punto 6.
8. Nivel e identificación de cambios sobre el programa.

9. Número de sentencias fuente añadidas en los cambios del programa.
10. Número de sentencias eliminadas en los cambios del programa.
11. Número de personas-hora por cambio.
12. Fecha de cambio del programa.
13. Identificación del ingeniero de software.
14. Identificación del FPM.
15. Tipo de mantenimiento.
16. Fechas de comienzo y final del mantenimiento.
17. Número de personas/hora acumulado del mantenimiento.
18. Beneficios netos asociados con el mantenimiento realizado.

Los datos anteriores han de ser recogidos para cada esfuerzo de mantenimiento. Swanson propone esos puntos como básicos para una base de datos que pueda ser evaluada tal y como se describe en la Sección 20.4.5.

La mayoría de los trabajos recientes sobre métricas del mantenimiento de software se centran en las características del software que más probablemente afectan a la frecuencia del mantenimiento y en modelos empíricos de predicción de la cantidad de trabajo de mantenimiento basándose en otras características del programa. En un estudio sobre unos 400 programas en COBOL, Vessey y Weber [VES83] descubrieron que la complejidad del programa y el estilo de programación eran los factores más significativos en la ocurrencia del mantenimiento correctivo. Usando como base el modelo COCOMO (Capítulo 3) [SCH85], fue sugerido el siguiente modelo para predecir el número de personas/mes, E.mant, dedicadas anualmente en el mantenimiento del software:

$$E.\text{mant} = TCA \times 2,4 \times KLDC^{1,05}$$

donde TCA es el *tráfico de cambios anual*, definido como:

$$TCA = \frac{\text{KLDC para un sistema en mantenimiento}}{IC}$$

donde IC = número de instrucciones de código fuente que se modifican o se añaden durante un año de mantenimiento.

Aunque los resultados de los estudios clínicos se deben aplicar con cuidado, los modelos cuantitativos proporcionan un elemento de control de la gestión que siempre ha faltado en el mantenimiento de software.

20.4.5. Evaluación

A menudo, la evaluación de las actividades de mantenimiento de software se complica por la falta de datos registrados. Si se sigue un registro de información, se pueden realizar varias medidas del rendimiento del mantenimiento. De nuevo, de la obra Swanson [SWA76] presentamos una lista resumida de medidas potenciales:

1. Número medio de fallos de procesamiento por ejecución del programa.
2. Total de personas/hora por cada categoría de mantenimiento.
3. Número medio de cambios por programa, por lenguaje y por tipo de mantenimiento.
4. Número medio de personas/hora por sentencia fuente añadida o eliminada debido al mantenimiento.
5. Media de personas/hora por lenguaje.
6. Tiempo total medio por FPM.
7. Porcentaje de peticiones de mantenimiento por tipos.

Estas siete medidas pueden proporcionar una base cuantitativa sobre la que tomar decisiones sobre la técnica de desarrollo, la selección de un lenguaje, las previsiones de esfuerzo de mantenimiento, de disposición de recursos y muchos otros aspectos. Claramente, esos datos se pueden aplicar en la evaluación de las tareas de mantenimiento.

20.5. EFECTOS SECUNDARIOS DEL MANTENIMIENTO

La modificación del software es peligrosa. Todos hemos oido el siguiente lamento: "...pero si todo lo que hice fue cambiar esta sentencia..." Desgraciadamente, cada vez que se introduce un cambio en un complejo procedimiento lógico, la posibilidad de error aumenta. La documentación del diseño y una cuidadosa prueba de regresión ayudan a eliminar los errores, pero seguirán apareciendo *efectos secundarios* del mantenimiento.

Cuando se usa en el contexto del mantenimiento de software, el término "efectos secundarios" implica un error u otro comportamiento indeseable aparecido como resultado de una modificación. Freedman y Weinberg [FRE90] definen tres grandes categorías de efectos secundarios, que se tratan en las siguientes secciones.

20.5.1. Efectos secundarios sobre el código

Un sencillo cambio sobre una sola sentencia puede a veces tener resultados desastrosos. El cambio inadvertido (e indetectado) de una "," por un "." tiene consecuencias casi trágicas si lo que falla es el software de control de vuelo de una sofisticada nave espacial. Aunque no todos los efectos secundarios tienen consecuencias tan dramáticas, el cambio invita al error y el error siempre lleva a problemas.

Nos comunicamos con la máquina mediante código fuente en un lenguaje de programación. Las posibilidades de efectos secundarios abundan. Aunque cada modificación del código potencialmente puede inducir a error, los siguientes cambios [FRE90] tienen mayor probabilidad de inducir a error que otros:

1. Un subprograma eliminado o cambiado.
2. Eliminación o modificación de una sentencia de etiqueta.

3. Eliminación o modificación de un identificador.
4. Cambios para mejorar el rendimiento en ejecución.
5. Modificación de apertura o cierre de archivos.
6. Modificación de operadores lógicos.
7. Traducción de cambios del diseño en importantes cambios sobre el código.
8. Cambios sobre las pruebas de límites.

Los efectos secundarios sobre el código van desde los insulsos errores que se detectan y se remedian durante las pruebas de regresión, hasta los problemas que pueden hacer que falle la operación del software. De nuevo podemos parafrasear la ley de Murphy: si un cambio en una sentencia de código puede introducir un error, lo hará.

20.5.2. Efectos secundarios sobre los datos

En el Capítulo 10 se recalcó la importancia de las estructuras de datos en el diseño del software. Durante el mantenimiento, a menudo se hacen cambios sobre determinados elementos de una estructura de datos o sobre la propia estructura de datos. Cuando cambian los datos, el diseño del software puede no cuadrar con los datos y aparecer errores. Los efectos secundarios sobre los datos también aparecen como resultado de las modificaciones sobre la estructura de información del software.

Los siguientes cambios en los datos [FRE90], a menudo, producen efectos secundarios: (1) redefinición de constantes locales o globales; (2) redefinición de formatos de registros o de archivos; (3) aumento o disminución del tamaño de arrays o de otras estructuras de datos de mayor orden; (4) modificación de datos globales; (5) reinicialización de indicadores de control o de punteros; (6) reorganización de argumentos de E/S o de subprogramas. Los efectos secundarios sobre los datos se pueden limitar mediante una profunda documentación de diseño que describa las estructuras de datos y dé una referencia cruzada que asocie los elementos de datos, los registros, los archivos y otras estructuras a los módulos del software.

20.5.3. Efectos secundarios sobre la documentación

El mantenimiento se debe centrar en la configuración completa del software y no sólo en las modificaciones del código fuente. Los efectos secundarios sobre la documentación se dan cuando no se reflejan los cambios del código fuente en la documentación de diseño y en los manuales orientados al usuario.

Siempre que se haga un cambio sobre el flujo de datos, sobre la arquitectura del diseño, sobre los procedimientos (módulos) o sobre cualquier otra característica asociada, se debe actualizar la documentación técnica de soporte. La documentación de diseño que no refleja fielmente el estado actual del software puede ser peor incluso que la ausencia total de documentación. Cuando un examen inocente de la documentación técnica lleva a una determinación inco-

rrecta de las características del software, se darán efectos secundarios en los subsiguientes esfuerzos de mantenimiento.

Para el usuario, el software sólo es tan bueno como lo sea la documentación (tanto escrita como interactiva) que describe su uso. Si no se reflejan los cambios del software ejecutable en la documentación de usuario, los efectos secundarios están garantizados. Por ejemplo, los cambios de formato en la entrada interactiva, si no están adecuadamente documentados, pueden producir problemas significativos. También, los nuevos mensajes de error no documentados pueden causar gran confusión; tablas, índices o texto no actualizados pueden llevar al usuario a la frustración y la insatisfacción.

Los efectos secundarios sobre la documentación se pueden reducir sustancialmente si se revisa la configuración entera antes de lanzar la nueva versión del software. De hecho, algunas peticiones de mantenimiento pueden no requerir cambios en el diseño del software o en el código fuente, sino indicar una falta de claridad en la documentación de usuario. En tales casos, el esfuerzo de mantenimiento se centrará en la documentación.

20.6. MANTENIMIENTO DE CODIGO “AJENO”

Casi cualquier organización de desarrollo de software madura debe mantener programas que han sido desarrollados hace quince o más años. A veces, a tales programas se les denomina “código ajeno” porque (1) los miembros del equipo técnico actual no trabajaron en el desarrollo del programa; (2) no se aplicó una metodología de desarrollo y, por tanto, existe un pobre diseño arquitectónico y de datos; la documentación es incompleta y no existe registro de los cambios anteriores.

Anteriormente, en este capítulo, hemos hablado de la necesidad de un *supervisor del sistema* —una persona que se familiariza con un subconjunto de los programas producidos que puedan requerir mantenimiento. La familiarización con los programas desarrollados bajo un enfoque de ingeniería del software se facilita con una completa configuración del software y un buen diseño. ¿Qué se puede hacer con el código ajeno? Yourdon [YOU75] hace varias sugerencias bastante útiles:

1. Estudiar el programa antes de entrar en “modo de emergencia”. Tratar de obtener la mayor información posible...
2. Tratar de familiarizarse con el flujo de control general del programa; ignorar en principio los detalles de codificación. Puede resultar muy útil dibujar un diagrama de estructura y un diagrama de flujo de alto nivel propios, si es que no existen ya.
3. Evaluar las características de la documentación existente; insertar comentarios propios en el listado (fuente) si se cree que serán de ayuda.
4. Hacer un buen uso de los listados de referencias cruzadas, de las tablas de símbolos y de otras ayudas que generalmente proporciona el compilador y/o ensamblador.

5. Hacer cambios en el programa con el mayor cuidado. Respetar el estilo y el formato del programa en la medida de lo posible. Señalar en el propio listado las instrucciones modificadas.
 6. No eliminar código hasta estar totalmente seguro.
 7. No intentar usar variables o almacenamiento temporal que ya existan en el programa. Insertar variables propias para evitar problemas.
 8. Mantener un detallado registro (de las actividades de mantenimiento y sus resultados).
 9. Evitar la necesidad irracional de tirar el programa y reescribirlo.
- [Nota del autor: Esta “necesidad” es a veces tanto racional como práctica].
10. Insertar comprobación de errores.

Cada una de las directrices anteriores ayudará en el mantenimiento de programas antiguos. Sin embargo, existe una clase de programa con un flujo de control que es el equivalente gráfico de un plato de espaguetis: con “módulos” de más de 2 000 líneas, con tres líneas de comentario útiles por cada 9 000 sentencias fuente y sin ningún otro elemento de configuración del software. Incrediblemente, tales programas pueden estar funcionando durante años, pero cuando se pide un mantenimiento, la tarea es inabordable. En la siguiente sección examinamos lo que se puede hacer.

20.7. INGENIERIA INVERSA Y REINGENIERIA

El término “ingeniería inversa” tiene sus orígenes en el mundo del hardware. Una compañía desensambla un producto de hardware de la competencia en un esfuerzo por comprender los “secretos” del diseño y la fabricación del competidor. Esos secretos se podrían comprender fácilmente si se obtuvieran las especificaciones del diseño y de la fabricación del competidor. Pero esos documentos son privados y no están disponibles para la compañía que lleva a cabo la ingeniería inversa. Esencialmente, una ingeniería inversa fructífera desemboca en una o más especificaciones de diseño y fabricación de un producto, obtenidas examinando ejemplares reales del producto.

La *ingeniería inversa* del software es bastante similar. Sin embargo, en la mayoría de los casos, el programa objeto de la ingeniería inversa no es un producto de la competencia. Más bien, se trata de un trabajo propio de la compañía (a menudo realizado muchos años atrás). Los “secretos” a comprender están ocultos porque no se desarrolló nunca una especificación. Por tanto, la ingeniería inversa para el software es el proceso de analizar un programa en un esfuerzo de crear una representación del programa de mayor nivel de abstracción que el código fuente. La ingeniería inversa es un proceso de *recuperación de diseño*. Las herramientas de ingeniería inversa extraen la información del diseño de datos, arquitectónico y procedimental de un programa.

La *reingeniería*, también denominada *renovación* o *reclamación* [CHI90], no sólo recupera la información de diseño de un software existente, sino que usa esa información para alterar o reconstruir el sistema existente, en un esfuerzo

de mejorar la calidad general. En la mayoría de los casos, el software resultante de la reingeniería reimplementa la función del sistema existente. Pero, al mismo tiempo, el desarrollador añade nuevas funciones y/o mejora el rendimiento general.

Dado que cualquier gran compañía (y muchas de las pequeñas) tienen millones de líneas de código que son candidatas para la ingeniería inversa y/o la reingeniería, podría parecer que cada compañía debería llevar a cabo el enorme esfuerzo que supone aplicar la reingeniería a todos los programas de su biblioteca. Desgraciadamente, esto no es realista por una serie de razones: (1) algunos de esos programas no se usan con mucha frecuencia y no es probable que vayan a cambiar; (2) las herramientas de ingeniería inversa y de reingeniería todavía se encuentran en su infancia³; (3) por ello, esas herramientas pueden realizar ingeniería inversa y reingeniería sólo para una clase limitada de aplicaciones y (4) el coste (en esfuerzo y en dinero) puede ser prohibitivo.

Para realizar el mantenimiento preventivo, una organización de software debe seleccionar los programas que sean susceptibles de cambiar en el futuro cercano y prepararlos para ese cambio. Para llevar a cabo esa tarea de mantenimiento, se utilizan la ingeniería inversa y la reingeniería. En las secciones que siguen examinamos el mantenimiento preventivo y estudiamos con más detalle la ingeniería inversa y la reingeniería.

20.7.1. Mantenimiento preventivo

Un programa con un flujo de control que recuerde un plato de espaguetis —con “módulos” de 2000 líneas, con tres comentarios significativos por cada 9000 sentencias fuente y sin ninguna otra documentación— debe ser modificado para acomodar los cambios en los requisitos de los usuarios. Tenemos las siguientes opciones:

1. Adentrarnos por las modificaciones, “luchando” con el diseño implícito y el código fuente para implementar los cambios oportunos.
2. Intentar comprender a grandes rasgos el funcionamiento interno del programa, en un esfuerzo de llevar a cabo las modificaciones de forma más efectiva.
3. Rediseñar, recodificar y probar las partes del software que requieran modificaciones, aplicando un enfoque de ingeniería del software a todos los segmentos revisados.
4. Rediseñar, recodificar y probar completamente el programa, utilizando herramientas CASE (herramientas de ingeniería inversa y de reingeniería) que nos ayuden a comprender el diseño actual.

³ Algunas herramientas CASE para campos de aplicación limitados son relativamente sofisticadas, pero, en general, las herramientas de ingeniería inversa y de reingeniería son bastante rudimentarias.

No existe una única opción “válida”. Las circunstancias pueden dictar la primera opción incluso cuando fueran más deseables las otras.

En lugar de esperar a recibir una petición de mantenimiento, la organización de desarrollo o de mantenimiento selecciona un programa que (1) vaya a estar en uso durante una determinada serie de años; (2) esté siendo usado correctamente y (3) pueda necesitar en un futuro cercano modificaciones o mejoras importantes. En este caso, se aplica la opción 2, 3 o 4⁴.

El enfoque de mantenimiento preventivo fue liderado por Miller [MIL81] bajo el título de “retroajuste estructurado”. Definió el concepto como “la aplicación de las metodologías actuales a sistemas de ayer para facilitar los requisitos de mañana”.

A primera vista, la sugerencia de redesarrollar un gran programa cuando existe ya una versión operativa puede parecer bastante extravagante. Antes de establecer un juicio, debemos considerar los siguientes puntos:

1. El coste de mantener una línea de código fuente puede ser de entre 20 y 40 veces el coste de desarrollo inicial de esa línea.
2. Rediseñar la arquitectura del software (estructura de datos y/o de programa) con metodologías modernas de diseño puede facilitar enormemente un futuro mantenimiento.
3. Debido a la existencia de un prototipo del software, la productividad de desarrollo puede ser mucho mayor que la media.
4. El usuario ya tiene experiencia con el software. Por tanto, es más fácil descubrir nuevos requisitos y la dirección del cambio.
5. Las herramientas CASE de ingeniería inversa y de reingeniería pueden automatizar algunas partes del trabajo.
6. Tras terminar el mantenimiento preventivo existirá una completa configuración del software.

Cuando una organización de desarrollo de software vende productos de software, el mantenimiento preventivo se ve como “nuevas versiones” del programa. Una gran casa de desarrollo de software (p. ej.: un grupo de desarrollo de software de sistemas de gestión para una gran compañía de productos de consumo) puede tener entre 500 y 2 000 programas bajo su responsabilidad. Deberá asignar prioridades de importancia a esos programas y así revisarlos como candidatos para el mantenimiento preventivo.

20.7.2. Elementos de ingeniería inversa

La ingeniería inversa conjura la imagen del “túnel mágico”. A un lado del tunel colocamos un listado fuente indocumentado y no estructurado y, por el otro

⁴ Antes de continuar, conviene volver a leer la Sección 1.1.3. Si la gestión se ajusta a la que denominábamos “planta de software que envejece”, el mantenimiento preventivo seguramente sea algo obligatorio para la compañía.

lado, sacamos una documentación completa del programa de computadora. Desgraciadamente, el tunel mágico no existe. La ingeniería inversa puede extraer información de diseño del código fuente, pero el nivel de abstracción, la completitud de la documentación, el grado en que las herramientas y el analista humano trabajan conjuntamente y la direccionalidad del proceso varían enormemente [CAS88].

El *nivel de abstracción* de un proceso de ingeniería inversa y de las herramientas que se usan para efectuarla se refiere a la sofisticación de la información de diseño que se pueda extraer del código fuente. Idealmente, el nivel de abstracción debería ser el mayor posible. Es decir, el proceso de ingeniería inversa debería poder obtener representaciones del diseño procedimental (un bajo nivel de abstracción), información de la estructura del programa y de los datos (un nivel algo mayor de abstracción), modelos del flujo de datos y de control (un nivel relativamente alto de abstracción) y modelos de relaciones de entidades (un alto nivel de abstracción). A medida que aumente el nivel de abstracción, el ingeniero de software obtiene una información que le permitirá comprender más fácilmente el programa.

La *completitud* del proceso de ingeniería inversa se refiere al nivel de detalle que proporcione en el nivel de abstracción. En la mayoría de los casos, la completitud decrece según aumenta el nivel de abstracción. Por ejemplo, dado un listado de código fuente, es relativamente fácil desarrollar una representación completa del diseño procedimental. También se pueden obtener sencillas representaciones del flujo de datos, pero será más difícil desarrollar un conjunto completo de diagramas de flujo de datos.

La completitud mejora en proporción directa con la cantidad de análisis que lleve a cabo la persona que realiza la ingeniería inversa. La *interactividad* se refiere al grado en que la persona esté “integrada” con las herramientas automáticas para crear un proceso efectivo de ingeniería inversa. En la mayoría de los casos, al aumentar el nivel de abstracción, la interactividad debe aumentar para que no sufra la completitud.

Si la *direccionalidad* del proceso de ingeniería inversa es de un sólo sentido, toda la información extraída del código fuente se le proporciona al ingeniero de software para que la use durante la actividad de mantenimiento. Si es en los dos sentidos, esa información alimenta una herramienta de reingeniería que intenta reestructurar o regenerar el programa antiguo.

20.7.3. Una técnica de reestructuración para la reingeniería

La reingeniería combina las características de extracción de información de análisis y de diseño de la ingeniería inversa con una capacidad de reestructuración de los datos, la arquitectura y la lógica del programa. La reestructuración sirve para obtener un diseño que produzca la misma función con una mayor calidad (Capítulo 10) que la del programa original.

Como ejemplo sencillo de reestructuración, consideraremos las técnicas de simplificación lógica de Warnier [WAR74]. Esas técnicas se pueden usar para

reestructurar la lógica “espagueti” de un programa y obtener un diseño procedimental que siga la filosofía de la programación estructurada (Capítulo 10). Para comprender cómo se puede llevar a cabo la reestructuración, debemos examinar primero la propia técnica de simplificación. Supongamos que utilizamos una herramienta de ingeniería inversa sobre un programa que usa los elementos de datos A , B , C y D y las acciones de procesamiento (segmentos de código) V , W , X , Y , Z y \bar{R} (no R). En la Figura 20.4 se muestra una representación tabular de los cuatro elementos de datos, A , B , C , D , y las correspondientes acciones de procesamiento que se generan. Esa *tabla de verdad* indica las circunstancias bajo las que se ejecutan las acciones de procesamiento. Por ejemplo, por la tercera línea, se ejecutarán las acciones V , e Y cuando se encuentre el elemento de datos C . V se puede ejecutar bajo dos conjuntos diferentes de condiciones. Aplicando el álgebra de Boole, obtenemos

$$\begin{aligned} V &= \bar{A} \cdot \bar{B} \cdot C \cdot \bar{D} + \bar{A} \cdot \bar{B} \cdot C \cdot D + \bar{A} \cdot B \cdot C \cdot \bar{D} + \bar{A} \cdot B \cdot C \cdot D \\ &\quad + A \cdot \bar{B} \cdot C \cdot \bar{D} + A \cdot \bar{B} \cdot C \cdot D + A \cdot B \cdot C \cdot \bar{D} + A \cdot B \cdot C \cdot D \end{aligned}$$

donde, en notación booleana, el punto indica una Y lógica y el + indica una O lógica. Aplicando reglas de simplificación lógica, obtenemos las siguientes ecuaciones:

$$\begin{aligned} V &= \bar{A} \cdot \bar{B} \cdot C(\bar{D} + D) + \bar{A} \cdot B \cdot C(\bar{D} + D) + A \cdot \bar{B} \cdot C(\bar{D} + D) \\ &\quad + A \cdot B \cdot C(\bar{D} + D) \\ &= \bar{A} \cdot \bar{B} \cdot C + \bar{A} \cdot B \cdot C + A \cdot \bar{B} \cdot C + A \cdot B \cdot C \end{aligned}$$

	DATOS ABCD	V	W	X	Y	Z	\bar{R}
		0000	0001	0010	0011	0100	0101
Alternativas complejas	0110	X			X		X
	0111	X			X	X	X
	1000						X
	1001						X
	1010	X	X	X			
	1011	X	X	X			
	1100						X
	1101						X
	1110	X	X				
	1111	X	X				

Figura 20.4. Una tabla de verdad. (Reimpreso con permiso de Van Nostrand Reinhold Company. Copyright 1974, *Les Editions d'Organisation*.)

$$\begin{aligned}
 &= \overline{A} \cdot C(\overline{B} + B) + A \cdot C(\overline{B} + B) \\
 &= \overline{A} \cdot C + A \cdot C = C(\overline{A} + A) \\
 &= C
 \end{aligned}$$

y, de forma similar,

$$\begin{aligned}
 X &= A \cdot \overline{B} \cdot C \cdot \overline{D} + A \cdot \overline{B} \cdot C \cdot D \\
 &= A \cdot \overline{B} \cdot C(\overline{D} + D) \\
 &= A \cdot \overline{B} \cdot C
 \end{aligned}$$

Por tanto, V se ejecutará siempre que se encuentre C , y X se ejecutará cuando estén presentes A y C pero no B .

A partir de la Tabla de la Figura 20.4, se pueden hacer las siguientes simplificaciones para las otras acciones de procesamiento:

$$\begin{aligned}
 V &= C \\
 W &= C \cdot A \\
 X &= C \cdot A \cdot \underline{B} \\
 Y &= C \cdot \underline{A} \cdot \underline{B} \\
 Z &= C \cdot \underline{A} \cdot B \\
 R &= \underline{C}
 \end{aligned}$$

Se pueden usar las simplificaciones lógicas como técnica de reingeniería para el software “no estructurado”. Para conseguir la *reestructuración*, se pueden seguir los siguientes pasos:

1. Desarrollar un diagrama de flujo para el software.
2. Derivar una expresión booleana para cada secuencia de procesamiento.
3. Compilar la tabla de verdad.
4. Reconstruir el software usando las técnicas descritas; añadir modificaciones según se requiera.

Como ejemplo [WAR74], consideremos el diagrama de flujo no estructurado y la tabla de verdad resultante, que se muestran en las Figuras 20.5 y 20.6. El mantenimiento requiere que se apliquen las modificaciones (“añadidos”) que se muestran en la Figura 20.6. Tras modificar la tabla de verdad y simplificarla, se puede derivar el diseño procedural estructurado que se muestra en la Figura 20.7. El proceso completo de los cuatro pasos anteriores se puede automatizar e implementar en una herramienta CASE para reingeniería.

También se han propuesto otras técnicas para herramientas de reingeniería. Por ejemplo, Choi y Scacchi [CHO90] sugieren la creación de un diagrama de intercambio de recursos que hace corresponder cada módulo de programa con los recursos (tipos de datos, procedimientos y variable) que intercambia con otros

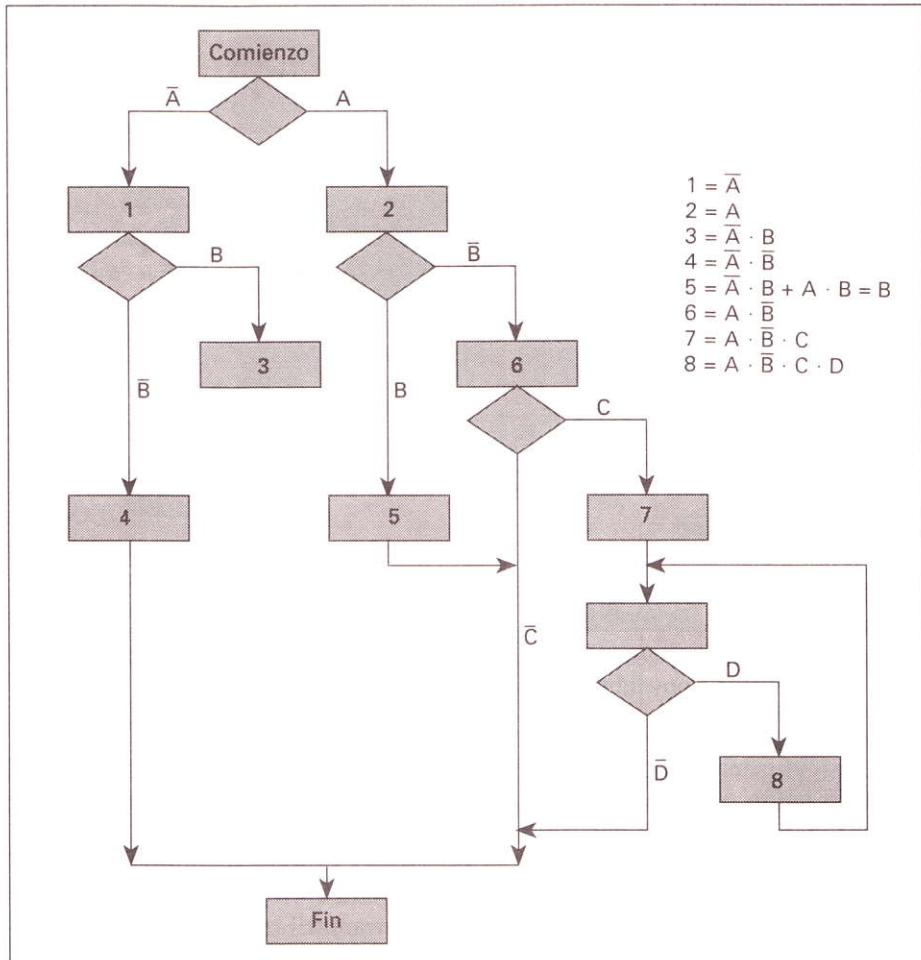


Figura 20.5. Un programa no estructurado. (Reimpreso con permiso de Van Nostrand Reinhold Company. Copyright 1974, *Les Editions d'Organisation*.)

módulos. Al crear representaciones del flujo de recursos, se puede reestructurar la arquitectura del programa para conseguir el menor nivel de acoplamiento entre los módulos.

20.8. RESUMEN

El mantenimiento, la última fase del proceso de ingeniería del software, se lleva la mayor parte del presupuesto destinado al software de computadora. A medida que se desarrollan más programas, surge una tendencia molesta —la cantidad de esfuerzo y de recursos dedicados al mantenimiento crece. Al final, al-

E ABCD	1	2	3	4	5	6	7	8
0000	X			X				
0001	X			X				
0010	X			X				
0011	X			X				
0100	X		X		X			
0101	X		X		X			
0110	X		X		X			
0111	X		X		X			
1000		X				X		
1001		X				X		
1010		X				X	X	
1011		X				X	X	X
1100		X			X			
1101		X			X			
1110		X			X			
1111		X			X			

Figura 20.6. Tabla de verdad resultante. (Reimpreso con permiso de Van Nostrand Reinhold Company. Copyright 1974, *Les Editions d'Organisation*.)

gunas organizaciones de software pueden llegar a encontrar la barrera del mantenimiento, no siendo capaces de embarcarse en nuevos proyectos porque tienen todos sus recursos dedicados al mantenimiento de programas antiguos.

Sobre el software de computadora se llevan a cabo cuatro tipos de mantenimiento. El mantenimiento correctivo actúa para corregir errores que no han sido descubiertos antes de poner en uso el software. El mantenimiento adaptativo se aplica cuando los cambios del entorno externo precipitan las modificaciones del software. El mantenimiento perfectivo incorpora mejoras solicitadas por la comunidad de usuarios. Finalmente, el mantenimiento preventivo mejora la futura facilidad de mantenimiento y la fiabilidad como base para las futuras mejoras.

Los enfoques técnicos y de gestión de la fase de mantenimiento se pueden implementar sin muchos inconvenientes. Sin embargo, las tareas desarrolladas durante el proceso de ingeniería del software definen la facilidad de mantenimiento y tienen un gran impacto sobre el éxito de cualquier método de mantenimiento.

Las herramientas de ingeniería inversa y de reingeniería llegarán a ser una parte importante del proceso de mantenimiento a finales de los 90. La ingeniería inversa extrae información de diseño del código fuente de un programa sin disponer de ninguna otra documentación. La reingeniería toma la información obtenida y reestructura el programa para conseguir una mayor calidad y, por tanto, una mejor facilidad de mantenimiento en el futuro.

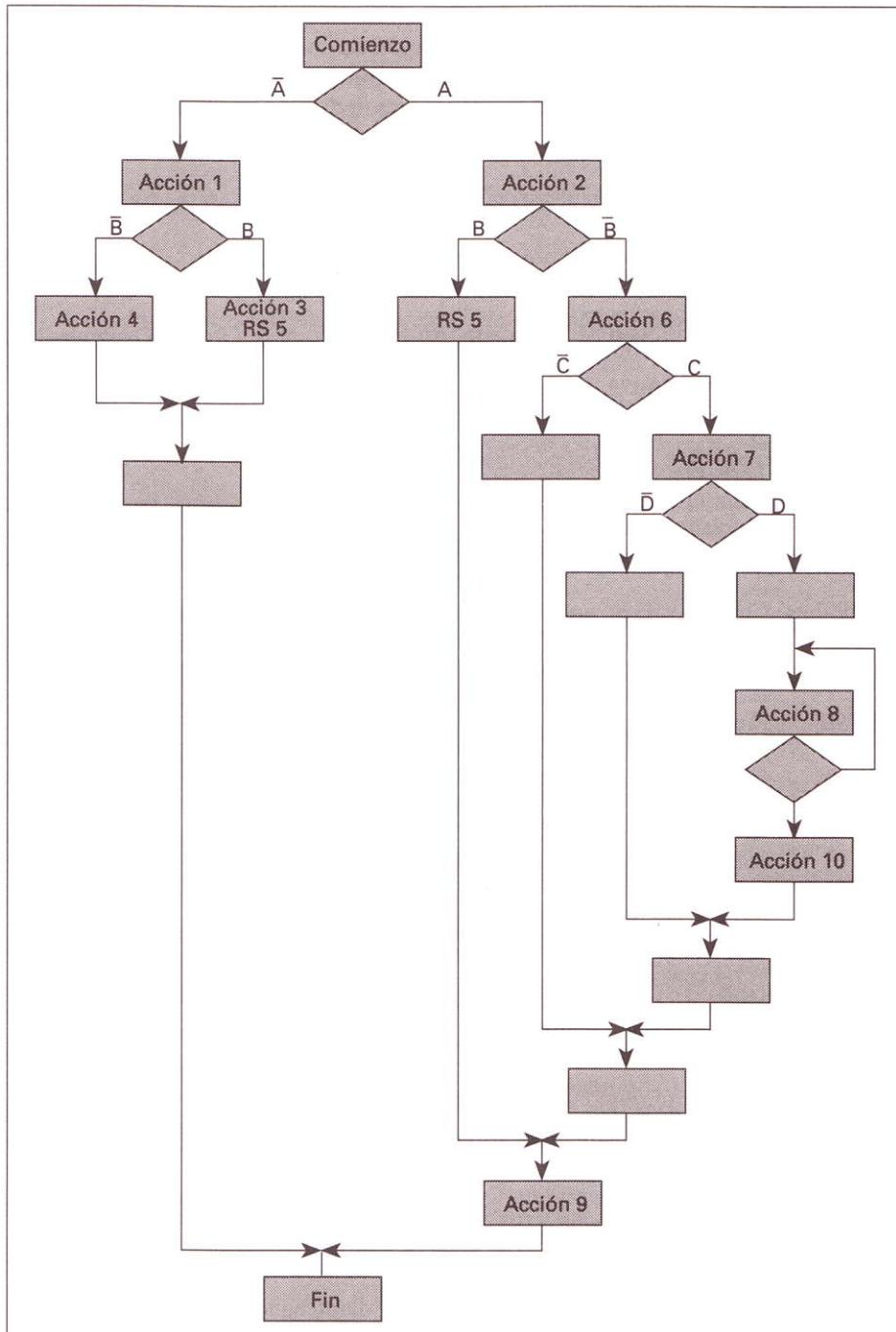


Figura 20.7. Programa estructurado y aumentado. (Reimpreso con permiso de Van Nostrand Reinhold Company. Copyright 1974, *Les Editions d'Organisation*.)

Los métodos y las técnicas presentadas en este libro tienen muchos fines, pero uno de los más importantes es la construcción de software que pueda cambiar. Se emplearán menos personas/hora en cada modificación o en cada petición de mantenimiento. La gente se verá liberada para dedicarse al desarrollo del nuevo software que se requerirá en los sistemas cada vez más complejos del siglo veintiuno.

REFERENCIAS

- [BEL72] Belady, L., and M. Lehman, "An Introduction to Growth Dynamics", in *Statistical Computer Performance Evaluation* (W. Freiberger, ed.), Academic Press, 1972, pp. 503-511.
- [BOE79] Boehm, B. "Software Engineering—R&D Trends and Defense Needs", in *Research Directions in Software Technology* (P. Wegner, ed.), MIT Press, 1979, pp. 44-86.
- [CAN72] Canning, R., "The Maintenance 'Iceberg'", *EDP Analyzer*, vol. 10, no. 10, October 1972.
- [CAS88] "Case Tools for Reverse Engineering", *CASE Outlook*, vol. 2, no. 2, 1988, p. 1-15.
- [CHI90] Chikofsky, E. J., and J. H. Cross, III, "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software*, January 1990, pp. 13-17.
- [CHO90] Choi, S. C., and W. Scacchi, "Extracting and Restructuring the Design of Large Systems", *IEEE Software*, January 1990, pp. 66-71.
- [FRE90] Freedman, D. P., and G. M. Weinberg, *Handbook of Workthroughs, Inspections, and Technical Reviews*, 3d ed., Dorset House, 1990.
- [GIL79] Gilb, T., "A Comment on the Definition of Reliability", *ACM Software Engineering Notes*, vol. 4, no. 3, July 1979.
- [KAF87] Kafura, D., and G. R. Reddy, "The Use of Software Complexity Metrics in Software Maintenance", *IEEE Trans. Software Engineering*, vol. SE-13, no. 3, March 1987, pp. 335-343.
- [KOP79] Kopetz, H., *Software Reliability*. Springer-Verlag, 1979, p. 93.
- [MCC80] McCracken, D., "Software in the 80s—Perils and Promises", *Computerworld* (special edition), vol. 14, no. 38, September 17, 1980, p. 5.
- [MIL81] Miller, J., in *Techniques of Program and System Maintenance* (G. Parikh, ed.), Winthrop Publishers, 1981.
- [OSB90] Osborne, W. M., and E. J. Chikofsky, "Fitting Pieces to the Maintenance Puzzle", *IEEE Software*, January 1990, pp. 10-11.
- [ROM87] Rombach, H. D., "A Controlled Experiment on the Impact of Software Structure on Maintainability", *IEEE Trans. Software Engineering*, vol. SE-13, no. 3, March 1987, pp. 344-354.
- [SCH85] Schaefer, H., "Metrics for Optimal Maintenance Management", *Proc. conf. Software Maintenance-1985*, IEEE, November 1985, pp. 114-119.
- [SCH87] Schmeidewind, N. F., "The State of Software Maintenance", *IEEE Trans. Software Engineering*, vol. SE-13, no. 3, March 1987, pp. 303-310.
- [SWA76] Swanson, E. B., "The Dimensions of Maintenance", *Proc. 2nd Intl. Conf. Software Engineering*, IEEE, October 1976, pp. 492-497.
- [VES83] Vessey, I., and R. Weber, "Some Factors Affecting Program Repair Maintenance", *CACM*, vol. 26, no. 2, February 1983, pp. 128-134.

- [WAR74] Warnier, J. D., *Logical Construction of Programs*, Van Nostrand Reinhold, 1974.
- [YOU75] Yourdon, E., *Techniques of Program Structure and Design*, Prentice-Hall, 1975, p. 24.

PROBLEMAS Y PUNTOS A CONSIDERAR

- 20.1. Su profesor seleccionará uno de los programas que haya desarrollado cada persona en clase durante este curso. Intercambie su programa aleatoriamente con cualquier otro miembro de la clase. *No explique ni inspeccione el programa.* Ahora, implemente una mejora (especificada por el profesor) en el programa que le haya tocado.
- (a) Lleve a cabo todas las tareas de ingeniería del software, incluyendo una breve inspección (pero no con el autor del programa).
 - (b) Lleve a cabo un cuidadoso seguimiento de todos los errores encontrados durante la prueba.
 - (c) Explique sus experiencias en clase.
- 20.2. Intente desarrollar un sistema de tasación de software que se pueda aplicar a programas existentes para encontrar candidatos para el mantenimiento preventivo.
- 20.3. ¿Es lo mismo el mantenimiento correctivo que la depuración? Explique su respuesta.
- 20.4. Discuta el impacto de los lenguajes de alto nivel sobre el mantenimiento adaptativo. ¿Es siempre posible adaptar un programa?
- 20.5. ¿Se deben incorporar los costes de mantenimiento en el paso de planificación? ¿Lo están?
- 20.6. La duración global del ciclo de vida del software, ¿aumentará o disminuirá en la próxima década? Discuta este asunto en clase.
- 20.7. Proyecto en equipo: desarrolle una herramienta automática que permita a un gestor recolectar y analizar datos cuantitativos de mantenimiento. Use las Secciones 20.3.2 y 20.4.4 como fuentes de requisitos del software.
- 20.8. Alguna gente cree que la inteligencia artificial mejorará el nivel de abstracción del proceso de ingeniería inversa. Investigue un poco sobre este asunto (es decir, el uso de la I.A. en la ingeniería inversa) y escriba un breve artículo que siente las bases sobre este asunto.
- 20.9. ¿Por qué es más difícil conseguir la completitud a medida que aumenta el nivel de abstracción?
- 20.10. ¿Por qué se debe aumentar la interactividad para que aumente la completitud?
- 20.11. Consiga folletos de tres herramientas de ingeniería inversa y exponga sus características en clase.
- 20.12. Discuta la viabilidad de una estrategia de “piezas de repuesto”. Considere tanto los aspectos técnicos como los económicos.
- 20.13. Para el experimentado: relate una “historia de miedo” sobre el mantenimiento.
- 20.14. Investigue en la bibliografía para intentar encontrar artículos y libros recientemente publicados sobre mantenimiento de software e ingeniería inversa, poniendo la mayor atención sobre los datos cuantitativos. Escriba un artículo a partir de lo que encuentre.

OTRAS LECTURAS

La literatura sobre mantenimiento de software creció significativamente durante principios de los años 80, pero pocos son los libros que se han publicado recientemente sobre este tema. La mayoría de los libros sobre mantenimiento se centran en las implicaciones de los procedimientos y los métodos de la ingeniería del software sobre el mantenimiento. Martin y McClure (*Software Maintenance*, Prentice-Hall, 1983) tratan el impacto de las técnicas de cuarta generación sobre el proceso de mantenimiento. Parikh (*Handbook of Software Maintenance*, Wiley-Interscience, 1986) trata el mantenimiento del software con un formato efectivo de preguntas y respuestas. Glass y Noiseux (*Software Maintenance Guidebook*, Prentice-Hall, 1981) sigue siendo un tratamiento valioso de la materia. Los datos recogidos por Lientz y Swanson (*Software Maintenance Management*, Addison-Wesley, 1980) siguen constituyendo el estudio más amplio sobre el mantenimiento que se ha publicado hasta la fecha.

La antología de Parikh (*Techniques of Program and System Maintenance*, Winthrop Publishers, 1981) y otra antología de Parikh y Zvezintsov (*Software Maintenance*, IEEE Computer Society Press, 1983) contienen colecciones de artículos sobre mantenimiento. Un número especial del *IEEE Transactions of Software Engineering* (vol. SE-13, nº 3, Marzo de 1987) está dedicado al mantenimiento del software y contiene seis artículos excelentes sobre el tema. Cada año hay varias conferencias sobre mantenimiento patrocinadas por la ACM y el IEEE, y una revista comercial, *Software Management News* (editada por N. Zvezintsov, Staten Island, NY), constituye un útil forum para las nuevas ideas.

La ingeniería inversa y la reingeniería son temas de extremada "actualidad" en la comunidad de la ingeniería del software, pero sólo ahora está empezando a existir bibliografía sobre ellos. Un número especial de la revista *IEEE Software* (enero de 1990) está dedicado al mantenimiento y a la ingeniería inversa y contiene un profundo tratamiento de la materia. *CASE Outlook*, *CASE Trends*, *CASE Strategies* y otras revistas comerciales mantienen un continuo flujo de información sobre las herramientas de ingeniería inversa y de reingeniería. Un número de la revista *Software Magazine* (mayo de 1990) contiene una lista de herramientas populares de ingeniería inversa.