

## CAPÍTULO

# 8

## GARANTÍA DE CALIDAD DEL SOFTWARE (SQA/GCS)\*

**E**l enfoque de ingeniería del software descrito en este libro se dirige hacia un solo objetivo: producir software de alta calidad. Pero a muchos lectores les inquietará la pregunta: «¿Qué es la calidad del software?»

Philip Crosby [CR079], en su famoso libro sobre calidad, expone esta situación:

El problema de la gestión de la calidad no es lo que la gente no sabe sobre ella. El problema es lo que creen que saben...

A este respecto, la calidad tiene mucho en común con el sexo. Todo el mundo lo quiere. (Bajo ciertas condiciones, por supuesto.) Todo el mundo cree que lo conoce. (Incluso aunque no quiera explicarlo.) Todo el mundo piensa que su ejecución sólo es cuestión de seguir las inclinaciones naturales. (Después de todo, nos las arreglamos de alguna forma.) Y, por supuesto, la mayoría de la gente piensa que los problemas en estas áreas están producidos por otra gente. (Como si sólo ellos se tomaran el tiempo para hacer las cosas bien.)

Algunos desarrolladores de software continúan creyendo que la calidad del software es algo en lo que empiezan a preocuparse una vez que se ha generado el código. ¡Nada más lejos de la realidad! La garantía de calidad del software (SQA, Software Quality Assurance GCS, Gestión de calidad del software) es una actividad de protección (Capítulo 2) que se aplica a lo largo de todo el proceso del software. La SQA engloba: (1) un enfoque de gestión de calidad; (2) tecnología de ingeniería del software efectiva (métodos y herramientas); (3) revisiones técnicas formales que se aplican durante el proceso del software; (4) una estrategia de prueba multiescalada; (5) el control de la documentación del software y de los cambios realizados; (6) un procedimiento que asegure un ajuste a los estándares de desarrollo del software (cuando sea posible), y (7) mecanismos de medición y de generación de informes.

En este capítulo nos centraremos en los aspectos de gestión y en las actividades específicas del proceso que permitan a una organización de software asegurar que hace «las cosas correctas en el momento justo y de la forma correcta».

### VISTAZO RÁPIDO

**¿Qué es?** No es suficiente hablar por hablar diciendo que la calidad del software es importante, tienes que: (1) definir explícitamente lo que significa «calidad del software»; (2) crear un conjunto de actividades que ayuden a garantizar que todo producto de la ingeniería del software presenta alta calidad; (3) llevar a cabo actividades de garantía de calidad en cada proyecto de software; (4) utilizar métricas para desarrollar estrategias que mejoren el proceso del software y, como consecuencia, mejoren la calidad del producto final.

**¿Quién lo hace?** Todo el que esté relacionado con el proceso de ingeniería del software es responsable de la calidad.

**¿Por qué es importante?** Lo puedes hacer correctamente o lo puedes

repetir. Si un equipo de software aplica la calidad a todas las actividades de la ingeniería del software, reducirá la cantidad de trabajo repetido que deba realizar. Esto supondrá costes más bajos y, lo que es más importante, mejorará el tiempo de llegada al mercado.

**¿Cuáles son los pasos?** Antes de que se puedan iniciar las actividades de garantía de calidad del software, es importante definir la «calidad del software» a un número diferente de niveles de abstracción. Una vez que comprendes lo que es la calidad, un equipo de software debe identificar un conjunto de actividades de garantía de calidad del software que eliminarán los errores de los productos realizados antes de que ocurran.

**¿Cuál es el producto obtenido?** Para definir una estrategia de SQA para el equipo de software se ha creado un plan de garantía de calidad del software. Durante el análisis, diseño y codificación, el producto principal de SQA es un breve informe de la revisión técnica formal. Durante las pruebas, se realizan los planes y procedimientos de prueba. También se pueden generar otros productos de trabajo relacionados con la mejora del proceso.

**¿Cómo puedo asegurar que lo he hecho correctamente?** Encontrar los errores antes que pasen a ser defectos!. Esto es, trabajar para mejorar tu eficiencia en la eliminación de errores (Capítulos 4 y 7), reduciendo de este modo la cantidad de trabajo repetido que tiene que hacer tu equipo de software.

\* N. del T.: En español, GCS. Se conservan las siglas SQA en inglés por estar muy extendido su uso para la jerga informática. A veces se suelen traducir estas siglas también por «Aseguramiento de la Calidad del Software».

## 8.1 CONCEPTOS DE CALIDAD<sup>1</sup>

Se dice que dos copos de nieve no son iguales. Ciertamente cuando se observa caer la nieve, es difícil imaginar que son totalmente diferentes, por no mencionar que cada copo posee una estructura única. Para observar las diferencias entre los copos de nieve, debemos examinar los especímenes muy de cerca, y quizás con un cristal de aumento. En efecto, cuanto más cerca los observemos, más diferencias podremos detectar.

Este fenómeno, *variación entre muestras*, se aplica a todos los productos del hombre así como a la creación natural. Por ejemplo, si dos tarjetas de circuito «idénticas» se examinan muy de cerca, podremos observar que las líneas de cobre sobre las tarjetas difieren ligeramente en la geometría, colocación y grosor. Además, la localización y el diámetro de los orificios de las tarjetas también varían.



**La gente olvida cómo de rápido hiciste un trabajo pero siempre recuerda cómo de bien lo hiciste.**

**Howard Newton**

*El control de variación* es el centro del control de calidad. Un fabricante quiere reducir la variación entre los productos que se fabrican, incluso cuando se realiza algo relativamente sencillo como la duplicación de discos. Seguramente, esto puede no ser un problema —la duplicación de discos es una operación de fabricación trivial y podemos garantizar que se crean duplicados exactos de software—.

¿Podemos? Necesitamos asegurar que las pistas se sitúen dentro de una tolerancia específica para que la gran mayoría de las disqueteras puedan leer los discos. Además, necesitamos asegurar que el flujo magnético para distinguir un cero de un uno sea suficiente para que los detecten las cabezas de lectura/escritura. Las máquinas de duplicación de discos aceptan o rechazan la tolerancia. Por consiguiente, incluso un proceso «simple», como la duplicación, puede encontrarse con problemas debidos a la variación entre muestras.



Controlar la variación es la clave de un producto de alta calidad. En el contexto del software, nos esforzamos en controlar la variación en el proceso que aplicamos, recursos que consumimos y los atributos de calidad del producto final.

¿Cómo se aplica esto al software? ¿Cómo puede una organización de desarrollo de software necesitar controlar la variación? De un proyecto a otro, queremos reducir la diferencia entre los recursos necesarios planificados para terminar un proyecto y los recursos reales utilizados, entre los que se incluyen personal, equipo y tiempo. En general, nos gustaría asegurarnos de que nuestro programa de pruebas abarca un porcentaje conocido del software de una entrega a otra. No sólo queremos reducir el número de defectos que se extraen para ese campo, sino también nos gustaría asegurarnos de que los errores ocultos también se reducen de una entrega a otra. (Es probable que nuestros clientes se molesten si la tercera entrega de un producto tiene diez veces más defectos que la anterior.) Nos gustaría reducir las diferencias en velocidad y precisión de nuestras respuestas de soporte a los problemas de los clientes. La lista se podría ampliar más y más.

### 8.1.1. Calidad

El *American Heritage Dictionary*, define la calidad como «una característica o atributo de algo». Como un atributo de un elemento, la calidad se refiere a las características mensurables —cosas que se pueden comparar con estándares conocidos como longitud, color, propiedades eléctricas, maleabilidad, etc.—. Sin embargo, el software en su gran extensión, como entidad intelectual, es más difícil de caracterizar que los objetos físicos.

No obstante, sí existen las medidas de características de un programa. Entre estas propiedades se incluyen complejidad ciclomática, cohesión, número de puntos de función, líneas de código y muchas otras estudiadas en los Capítulos 19 y 24. Cuando se examina un elemento según sus características mensurables, se pueden encontrar dos tipos de calidad: calidad del diseño y calidad de concordancia.



**Llevo menos tiempo hacer una cosa bien**

**que explicar por qué se hizo mal.**

**Henry Wadsworth Longfellow**

La *calidad de diseño* se refiere a las características que especifican los ingenieros de software para un elemento. El grado de materiales, tolerancias y las especificaciones del rendimiento contribuyen a la calidad del diseño. Cuando se utilizan materiales de alto grado y se

<sup>1</sup> Esta sección, escrita por Michael Cioveky, se ha adaptado de «(Fundamentals of ISO 9000», un libro de trabajo desarrollado para Essential Software Engineering, un video curriculum desarrollado por R.S. Pressman & Asociados, Inc. Reimpresión autorizada.

especifican tolerancias más estrictas y niveles más altos de rendimiento, la calidad de diseño de un producto aumenta, si el producto se fabrica de acuerdo con las especificaciones.

La *calidad de concordancia* es el grado de cumplimiento de las especificaciones de diseño durante su realización. Una vez más, cuanto mayor sea el grado de cumplimiento, más alto será el nivel de calidad de concordancia.

En el desarrollo del software, la calidad de diseño comprende los requisitos, especificaciones y el diseño del sistema. La calidad de concordancia es un aspecto centrado principalmente en la implementación. Si la implementación sigue el diseño, y el sistema resultante cumple los objetivos de requisitos y de rendimiento, la calidad de concordancia es alta.

Pero, ¿son la calidad del diseño y la calidad de concordancia los únicos aspectos que deben considerar los ingenieros de software? Robert Glass [GLA98] establece para ello una relación más «intuitiva»:

$$\text{satisfacción del usuario} = \text{productos satisfactorio} + \text{buena calidad} + \text{entrega dentro de presupuesto y del tiempo establecidos}$$

En la última línea, Glass afirma que la calidad es importante, pero si el usuario no queda satisfecho, ninguna otra cosa realmente importa. DeMarco [DEM99] refuerza este punto de vista cuando afirma: «La calidad del producto es una función de cuánto cambia el mundo para mejor.» Esta visión de la calidad establece que si el producto de software proporciona un beneficio sustancial a los usuarios finales, pueden estar dispuestos para tolerar problemas ocasionales del rendimiento o de fiabilidad.

### 8.1.2. Control de calidad

El control de cambios puede equiparse al control de calidad. Pero, ¿cómo se logra el control de calidad? *El control de calidad* es una serie de inspecciones, revisiones y pruebas utilizados a lo largo del proceso del software para asegurar que cada producto cumple con los requisitos que le han sido asignados. El control de calidad incluye un bucle de realimentación (feedback) del proceso que creó el producto. La combinación de medición y realimentación permite afinar el proceso cuando los productos de trabajo creados fallan al cumplir sus especificaciones. Este enfoque ve el control de calidad como parte del proceso de fabricación.



¿Qué es el control de calidad del software?

Las actividades de control de calidad pueden ser manuales, completamente automáticas o una combinación de herramientas automáticas e interacción humana. Un concepto clave del control de calidad es que se hayan definido todos los productos y las espe-

cificaciones mensurables en las que se puedan comparar los resultados de cada proceso. El bucle de realimentación es esencial para reducir los defectos producidos.

### 8.1.3. Garantía de calidad

La *garantía de calidad* consiste en la auditoría y las funciones de información de la gestión. El objetivo de la garantía de calidad es proporcionar la gestión para informar de los datos necesarios sobre la calidad del producto, por lo que se va adquiriendo una visión más profunda y segura de que la calidad del producto está cumpliendo sus objetivos. Por supuesto, si los datos proporcionados mediante la garantía de calidad identifican problemas, es responsabilidad de la gestión afrontar los problemas y aplicar los recursos necesarios para resolver aspectos de calidad.



**Referencia Web**  
Se puede encontrar una gran variedad de recursos de calidad del software en  
[www.qualitytree.com/links/links.htm](http://www.qualitytree.com/links/links.htm)

### 8.1.4. Coste de calidad

El *coste de calidad* incluye todos los costes acarreados en la búsqueda de la calidad o en las actividades relacionadas en la obtención de la calidad. Se realizan estudios sobre el coste de calidad para proporcionar una línea base del coste actual de calidad, para identificar oportunidades de reducir este coste, y para proporcionar una base normalizada de comparación. La base de normalización siempre tiene un precio. Una vez que se han normalizado los costes de calidad sobre un precio base, tenemos los datos necesarios para evaluar el lugar en donde hay oportunidades de mejorar nuestros procesos. Es más, podemos evaluar cómo afectan los cambios en términos de dinero.

Los *costes de calidad* se pueden dividir en costes asociados con la prevención, la evaluación y los fallos. Entre *los costes de prevención* se incluyen:

- planificación de la calidad,
- revisiones técnicas formales,
- equipo de pruebas,
- formación.



¿Cuáles son los componentes del coste de calidad?

Entre *los costes de evaluación* se incluyen actividades para tener una visión más profunda de la condición del producto «la primera vez a través de» cada proceso. A continuación se incluyen algunos ejemplos de costes de evaluación:

- inspección en el proceso y entre procesos,
- calibrado y mantenimiento del equipo,
- pruebas.



*No tengo miedo de incurrir en costes significativos de prevención. Esté segura de que su inversión le proporcionará un beneficio excelente.*

Los *costes de fallos* son los costes que desaparecerían si no surgieran defectos antes del envío de un producto a los clientes. Estos costes se pueden subdividir en costes de fallos internos y costes de fallos externos. Los *internos* se producen cuando se detecta un error en el producto antes de su envío. Entre estos se incluyen:

- retrabajo (revisión),
- reparación,
- análisis de las modalidades de fallos.

Los *costes de fallos externos* son los que se asocian a los defectos encontrados una vez enviado el producto al cliente. A continuación se incluyen algunos ejemplos de costes de fallos externos:

- resolución de quejas,
- devolución y sustitución de productos,
- soporte de línea de ayuda,
- trabajo de garantía.

Como es de esperar, los costes relativos para encontrar y reparar un defecto aumentan dramáticamente a medida que se cambia de prevención a detección y desde el fallo interno al externo. La Figura 8.1, basada en datos recopilados por Boehm [BOE81], ilustra este fenómeno.

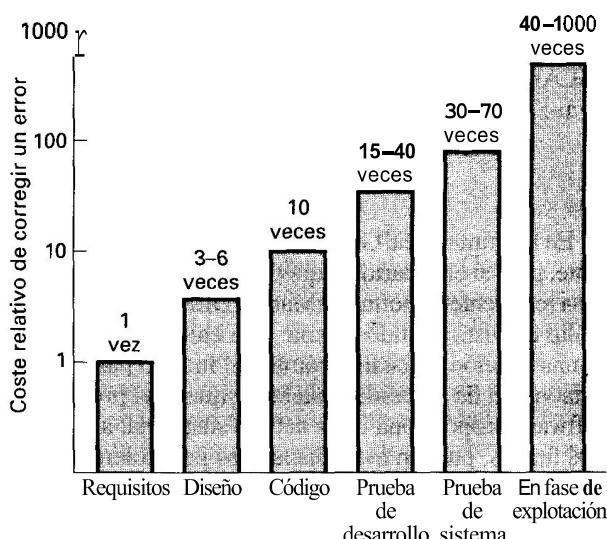


*las pruebas son necesarias, pero también es una forma costosa de encontrar errores. Gaste el tiempo en encontrar errores al comienzo del proceso y podrá reducir significativamente los costes de pruebas y depuración.*

Kaplan y sus colegas [KAP95] refuerzan las estadísticas de costes anteriores informando con datos anecdoticos basados en un trabajo realizado en las instalaciones de desarrollo de IBM en Rochester:

Se han dedicado 7.053 horas inspeccionando 200.000 líneas de código con el resultado de 3.112 errores potenciales descubiertos. Dando por sentado un coste de programador de 40 dólares por hora, el coste de eliminar 3.112 defectos ha sido de 282.120 dólares, o aproximadamente unos 91 dólares por defecto.

Compare estos números con el coste de eliminación de defectos una vez que el producto se ha enviado al cliente. Suponga que no ha habido inspecciones, pero que los programadores han sido muy cuidadosos y solamente se ha escapado un defecto por 1.000 líneas de código [significativamente mejor que la media en industrial] en el producto enviado. Eso significaría que se tendrían que corregir todavía 200 defectos en la casa del cliente o después de la entrega. A un coste estimado de 25.000 dólares por reparación de campo, el coste sería de 5 millones de dólares, o aproximadamente 18 veces más caro que el coste total del esfuerzo de prevención de defectos.



**FIGURA 8.1.** Coste relativo de corregir un error.

Es verdad que IBM produce software utilizado por cientos de miles de clientes y que sus costes por reparación en casa del cliente o después de la entrega pueden ser más altos que los de organizaciones de software que construyen sistemas personalizados. Esto, de ninguna manera, niega los resultados señalados anteriormente. Aunque la organización media de software tiene costes de reparación después de la entrega que son el 25 por 100 de los de IBM (¡la mayoría no tienen ni idea de cuales son sus costes!), se están imponiendo ahorros en el coste asociados con actividades de garantía y control de calidad.

## 8.2 LA TENDENCIA DE LA CALIDAD

Hoy en día los responsables de compañías de todo el mundo industrializado reconocen que la alta calidad del producto se traduce en ahorro de coste y en una mejora general. Sin embargo, esto no era siempre el caso.

La tendencia de la calidad comenzó en los años cuarenta con el influyente trabajo de W. Edwards Deming [DEM86], y se hizo la primera verificación en Japón. Mediante las ideas de Deming como piedra angular, los

japoneses han desarrollado un enfoque sistemático para la eliminación de las causas raíz de defectos en productos. A lo largo de los años setenta y ochenta, su trabajo emigró al mundo occidental y a veces se llama «gestión total de calidad (GTC)<sup>2</sup>. Aunque la terminología difiere según los diferentes países y autores, normalmente se encuentra una progresión básica de cuatro pasos que constituye el fundamento de cualquier programa de GTC.



### CLAVE

Se puede aplicar GTC al software de computadora. El enfoque GTC se centra en la mejora continua del proceso.

El primer paso se llama *kuizan* y se refiere a un sistema de mejora continua del proceso. El objetivo de *kai-zen* es desarrollar un proceso (en este caso, proceso del software) que sea visible, repetible y mensurable.

El segundo paso, invocado sólo una vez que se ha alcanzado *kuizan*, se llama *atuirimae hinshitsu*. Este paso examina lo intangible que afecta al proceso y trabaja para optimizar su impacto en el proceso. Por ejemplo, el proceso de software se puede ver afectado por la alta rotación de personal que ya en sí mismo se ve afectado por reorganizaciones dentro de una compañía. Puede ser que una estructura organizativa estable haga mucho para mejorar la calidad del software. *Atuirimae hinshitsu* llevaría a la gestión a sugerir cambios en la forma en que ocurre la reorganización.

Mientras que los dos primeros pasos se centran en el proceso, el paso siguiente llamado *kansei* (traducido como «los cinco sentidos») se centra en el usuario del producto (en este caso, software). En esencia, examinando la forma en que el usuario aplica el producto, *kunsei* conduce a la mejora en el producto mismo, y potencialmente al proceso que lo creó.

Finalmente, un paso llamado *miryokuteki hinshitsu* amplía la preocupación de la gestión más allá del producto inmediato. Este es un paso orientado a la gestión que busca la oportunidad en áreas relacionadas que se pueden identificar observando la utilización del producto en el mercado. En el mundo del software, *miryokuteki hinshitsu* se podría ver como un intento de detectar productos nuevos y beneficiosos, o aplicaciones que sean una extensión de un sistema ya existente basado en computadora.



### Referencia Web

Se puede encontrar una gran variedad de recursos para la mejora continua del proceso y GTC en [deming.eng.clemson.edu/](http://deming.eng.clemson.edu/)

Para la mayoría de las compañías, *kuizan* debería ser de preocupación inmediata. Hasta que se haya logrado un proceso de software avanzado (Capítulo 2), no hay muchos argumentos para seguir con los pasos siguientes.

## GARANTÍA DE CALIDAD DEL SOFTWARE

Hasta el desarrollador de software más agobiado estará de acuerdo con que el software de alta calidad es una meta importante. Pero, ¿cómo definimos la calidad? Un bromista dijo una vez: «Cualquier programa hace algo bien, lo que puede pasar es que no sea lo que nosotros queremos que haga».



### ¿Cómo se define «calidad del software»?

En los libros se han propuesto muchas definiciones de calidad del software. Por lo que a nosotros respecta, la calidad del software se define como:

Concordancia con los requisitos funcionales y de rendimiento explícitamente establecidos, con los estándares de desarrollo explicitamente documentados, y con las características implícitas que se espera de todo software desarrollado profesionalmente.

No hay duda de que la anterior definición puede ser modificada o ampliada. De hecho, no tendría fin una discusión sobre una definición formal de calidad del soft-

ware. Para los propósitos de este libro, la anterior definición sirve para hacer hincapié en tres puntos importantes:

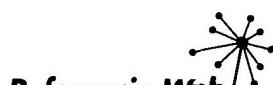
1. Los requisitos del software son la base de las medidas de la calidad. La falta de concordancia con los requisitos es una falta de calidad.
2. Los estándares especificados definen un conjunto de criterios de desarrollo que guían la forma en que se aplica la ingeniería del software. Si no se siguen esos criterios, casi siempre habrá falta de calidad.
3. Existe un conjunto de requisitos implícitos que a menudo no se mencionan (por ejemplo: el deseo por facilitar el uso y un buen mantenimiento). Si el software se ajusta a sus requisitos explícitos pero falla en alcanzar los requisitos implícitos, la calidad del software queda en entredicho.

### 8.3.1. Problemas de fondo

La historia de la garantía de calidad en el desarrollo de software es paralela a la historia de la calidad en la creación de hardware. Durante los primeros años de la infor-

<sup>2</sup> Consulte [ART92] para un estudio más completo del GTC y su uso en un contexto de software, y [KAP95] para un estudio sobre el uso de criterio «Balbridge Award» en el mundo del software.

mática (los años cincuenta y sesenta), la calidad era responsabilidad únicamente del programador. Durante los años setenta se introdujeron estándares de garantía de calidad para el software en los contratos militares para desarrollo de software y se han extendido rápidamente a los desarrollos de software en el mundo comercial [IEE94]. Ampliando la definición presentada anteriormente, la garantía de calidad del software (SQA) es un «patrón de acciones planificado y sistemático»[SCH97] que se requiere para asegurar la calidad del software. El ámbito de la responsabilidad de la garantía de calidad se puede caracterizar mejor parafraseando un popular anuncio de coches: «La calidad es la 1.<sup>a</sup> tarea.» La implicación para el software es que muchos de los que constituyen una organización tienen responsabilidad de garantía de calidad del software —ingenieros de software, jefes de proyectos, clientes, vendedores, y aquellas personas que trabajan dentro de un grupo de SQA—.



Se puede encontrar un tutorial profundo y amplios recursos para la gestión de calidad en [www.management.gov](http://www.management.gov)

El grupo de SQA sirve como representación del cliente en casa. Es decir, la gente que lleva a cabo la SQA debe mirar el software desde el punto de vista del cliente. ¿Satisface de forma adecuada el software los factores de calidad apuntados en el Capítulo 19? ¿Se ha realizado el desarrollo del software de acuerdo con estándares preestablecidos? ¿Han desempeñado apropiadamente sus papeles las disciplinas técnicas como parte de la actividad de SQA? El grupo de SQA intenta responder a estas y otras preguntas para asegurar que se mantiene la calidad del software.

### 8.3.2. Actividades de SQA

La garantía de calidad del software comprende una gran variedad de tareas, asociadas con dos constitutivos diferentes —los ingenieros de software que realizan trabajo técnico y un grupo de SQA que tiene la responsabilidad de la Planificación de garantía de calidad, supervisión, mantenimiento de registros, análisis e informes—.

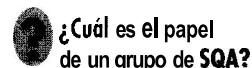
Los ingenieros de software afrontan la calidad (y realizan garantía de calidad) aplicando métodos técnicos sólidos y medidas, realizando revisiones técnicas formales y llevando a cabo pruebas de software bien planificadas. Solamente las revisiones son tratadas en este capítulo. Los temas de tecnología se estudian en las Partes Tercera a Quinta de este libro.

Las reglas del grupo de SQA tratan de ayudar al equipo de ingeniería del software en la consecución de un producto final de alta calidad. El Instituto de Ingeniería del Software [PAU93] recomienda un conjunto de activida-

des de SQA que se enfrentan con la planificación de garantía de calidad, supervisión, mantenimiento de registros, análisis e informes. Éstas son las actividades que realizan (o facilitan) un grupo independiente de SQA:

*Establecimiento de un plan de SQA para un proyecto.* El plan se desarrolla durante la planificación del proyecto y es revisado por todas las partes interesadas. Las actividades de garantía de calidad realizadas por el equipo de ingeniería del software y el grupo SQA son gobernadas por el plan. El plan identifica:

- evaluaciones a realizar,
- auditorías y revisiones a realizar,
- estándares que se pueden aplicar al proyecto,
- procedimientos para información y seguimiento de errores,
- documentos producidos por el grupo SQA,
- realimentación de información proporcionada al equipo de proyecto del software.



*Participación en el desarrollo de la descripción del proceso de software del proyecto.* El equipo de ingeniería del software selecciona un proceso para el trabajo que se va a realizar. El grupo de SQA revisa la descripción del proceso para ajustarse a la política de la empresa, los estándares internos del software, los estándares impuestos externamente (por ejemplo: ISO 9001), y a otras partes del plan de proyecto del software.

*Revisión de las actividades de ingeniería del software para verificar su ajuste al proceso de software definido.* El grupo de SQA identifica, documenta y sigue la pista de las desviaciones desde el proceso y verifica que se han hecho las correcciones.

*Auditoría de los productos de software designados para verificar el ajuste con los definidos como parte del proceso del software.* El grupo de SQA revisa los productos seleccionados; identifica, documenta y sigue la pista de las desviaciones; verifica que se han hecho las correcciones, e informa periódicamente de los resultados de su trabajo al gestor del proyecto.

*Asegurar que las desviaciones del trabajo y los productos del software se documentan y se manejan de acuerdo con un procedimiento establecido.* Las desviaciones se pueden encontrar en el plan del proyecto, en la descripción del proceso, en los estándares aplicables o en los productos técnicos.

*Registrar lo que no se ajuste a los requisitos e informar a sus superiores.* Los elementos que no se ajustan a los requisitos están bajo seguimiento hasta que se resuelven.

Además de estas actividades, el grupo de SQA coordina el control y la gestión de cambios (Capítulo 9) y ayuda a recopilar y a analizar las métricas del software.

## REVISIONES DEL SOFTWARE

Las revisiones del software son un «filtro» para el proceso de ingeniería del software. Esto es, las revisiones se aplican en varios momentos del desarrollo del software y sirven para detectar errores y defectos que pueden así ser eliminados. Las revisiones del software sirven para «purificar» las actividades de ingeniería del software que suceden como resultado del análisis, el diseño y la codificación. Freedman y Weinberg [FRE90] argumentan de la siguiente forma la necesidad de revisiones:

El trabajo técnico necesita ser revisado por la misma razón que los lápices necesitan gomas: errar es humano. La segunda razón por la que necesitamos revisiones técnicas es que, aunque la gente es buena descubriendo algunos de sus propios errores, algunas clases de errores se le pasan por alto más fácilmente al que los origina que a otras personas. El proceso de revisión es, por tanto, la respuesta a la plegaria de Robert Burns:

*¡Qué gran regalo sería poder verlos como nos ven los demás!*

Una revisión —cualquier revisión— es una forma de aprovechar la diversidad de un grupo de personas para:

1. señalar la necesidad de mejoras en el producto de una sola persona o un equipo;
2. confirmar las partes de un producto en las que no es necesaria o no es deseable una mejora; y
3. conseguir un trabajo técnico de una calidad más uniforme, o al menos más predecible, que la que puede ser conseguida sin revisiones, con el fin de hacer más manejable el trabajo técnico.



*Al igual que los filtros de agua, las RTF's tienden a retardar el «flujo» de las actividades de ingeniería del software. Muy pocos y el flujo es «sucio». Muchos y el flujo se reducirá o un goteo. Utilice métricos para determinar qué revisiones son efectivas y cuáles no. Saque los que no sean efectivos fuera del flujo.*

Existen muchos tipos diferentes de revisiones que se pueden llevar adelante como parte de la ingeniería del software. Cada una tiene su lugar. Una reunión informal alrededor de la máquina de café es una forma de revisión, si se discuten problemas técnicos. Una presentación formal de un diseño de software a una audiencia de clientes, ejecutivos y personal técnico es una forma de revisión. Sin embargo, en este libro nos centraremos en la *revisión técnica formal (RTF)* —a veces denominada *inspección*. Una revisión técnica formal es el filtro más efectivo desde el punto de vista de garantía de calidad. Llevada a cabo por ingenieros del software (y otros), la RTF es para ellos un medio efectivo para mejorar la calidad del software.

### 8.4.1. Impacto de los defectos del software sobre el coste

El IEEE Standard Dictionary of Electrical and Electronics Terms (IEEE Standard 100-1992) define un

defecto como una «anomalía del producto». La definición de «fallo» en el contexto del hardware se puede encontrar en IEEE Standard 610. 12-1990:

- (a) Un defecto en un dispositivo de hardware o componente: por ejemplo, un corto circuito o un cable roto.
- (b) Un paso incorrecto, proceso o definición de datos en un programa de computadora. Nota: Esta definición se usa principalmente por la disciplina de tolerancia de fallos. En su uso normal, los términos «error» y «fallo» se utilizan para expresar este significado. Consulte también: fallo sensible al dato; fallo sensible al programa; fallos equivalentes; enmascaramiento de fallos; **fallo** intermitente.

Dentro del contexto del proceso del software, los términos defecto y fallo son sinónimos. Ambos implican un problema de calidad que es descubierto *después* de entregar el software a los usuarios finales (o a otra actividad del proceso del software). En los primeros capítulos, se utilizó el término *error* para representar un problema de calidad que es descubierto por los ingenieros del software (u otros) antes de entregar el software al usuario final (o a otra actividad del proceso del software).

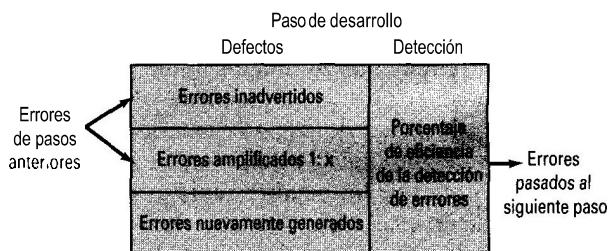


FIGURA 8.2. Modelo de amplificación de defectos.

El objetivo primario de las revisiones técnicas formales es encontrar errores durante el proceso, de forma que se conviertan en defectos después de la entrega del software. El beneficio obvio de estas revisiones técnicas formales es el descubrimiento de errores al principio para que no se propaguen al paso siguiente del proceso del software.



### CLAVE

El objetivo principal de una RTF es encontrar errores antes de pasar a otra actividad de ingeniería del software o de entregar al cliente.

Una serie de estudios (TRW, Nippon Electric y Mitre Corp., entre otros) indican que las actividades del diseño introducen entre el 50 al 65 por ciento de todos los errores (y en último lugar, todos los defectos) durante

el proceso del software. Sin embargo, se ha demostrado que las revisiones técnicas formales son efectivas en un 75 por 100 a la hora de detectar errores [JON86]. Con la detección y la eliminación de un gran porcentaje de errores, el proceso de revisión reduce substancialmente el coste de los pasos siguientes en las fases de desarrollo y de mantenimiento.

Para ilustrar el impacto sobre el coste de la detección anticipada de errores, consideremos una serie de costes relativos que se basan en datos de coste realmente recogidos en grandes proyectos de software [IBM81]<sup>3</sup>. Supongamos que un error descubierto durante el diseño cuesta corregirlo 1,0 unidad monetaria. De acuerdo con este coste, el mismo error descubierto justo antes de que comienza la prueba costará 6,5 unidades; durante la prueba 15 unidades; y después de la entrega, entre 60 y 100 unidades.

#### 8.4.2. Amplificación y eliminación de defectos

Se puede usar un modelo de amplificación de defectos [IBM81] para ilustrar la generación y detección de errores durante los pasos de diseño preliminar, diseño detallado y codificación del proceso de ingeniería del software. En la Figura 8.2 se ilustra esquemáticamente el modelo. Cada cuadro representa un paso en el desarrollo del software. Durante cada paso se pueden generar errores que se pasan inadvertidos. La revisión puede fallar en descubrir nuevos errores y errores de pasos anteriores, produciendo un mayor número de errores que pasan inadvertidos. En algunos casos, los errores que pasan inadvertidos desde pasos anteriores se amplifican (factor de amplificación,  $x$ ) con el trabajo actual. Las subdivisiones de los cuadros representan cada una de estas características y el porcentaje de eficiencia para la detección de errores, una función de la profundidad de la revisión.

La Figura 8.3 ilustra un ejemplo hipotético de la amplificación de defectos en un proceso de desarrollo de software en el que no se llevan a cabo revisiones. Como muestra la figura, se asume que cada paso descubre y

corrige el 50 por 100 de los errores que le llegan, sin introducir ningún error nuevo (una suposición muy optimista). Antes de que comience la prueba, se han amplificado diez errores del diseño preliminar a 94 errores. Al terminar quedan 12 errores latentes. La Figura 8.4 considera las mismas condiciones, pero llevando a cabo revisiones del diseño y de la codificación dentro de cada paso del desarrollo. En este caso los 10 errores del diseño preliminar se amplifican a 24 antes del comienzo de la prueba. Sólo quedan 3 errores latentes. Recordando los costes relativos asociados con el descubrimiento y la corrección de errores, se puede establecer el coste total (con y sin revisiones para nuestro ejemplo hipotético). El número de errores encontrado durante cada paso mostrado en las figuras 8.3 y 8.4 se multiplica por el coste de eliminar un error (1.5 unidades de coste del diseño, 6.5 unidades de coste antes de las pruebas, 15 unidades de coste durante las pruebas, y 67 unidades de coste después de la entrega). Utilizando estos datos, se puede ver que el coste total para el desarrollo y el mantenimiento cuando se realizan revisiones es de 783 unidades. Cuando no hay revisiones, el coste total es de 2.177 unidades —casi tres veces más caro—.



**Cita:**

Algunas enfermedades, como dicen los médicos, en su comienzo son fáciles de curar pero difíciles de reconocer... pero con el paso del tiempo, cuando no se han reconocido y tratado al principio, se vuelven fáciles de reconocer pero difíciles de curar.

Niccolò Machiavelli

Para llevar a cabo revisiones, el equipo de desarrollo debe dedicar tiempo y esfuerzo, y la organización de desarrollo debe gastar dinero. Sin embargo, los resultados del ejemplo anterior no dejan duda de que hemos encontrado el síndrome de «pague ahora o pague, después, mucho más». Las revisiones técnicas formales (para el diseño y otras actividades técnicas) producen un beneficio en coste demostrable. Deben llevarse a cabo.

### 8.5. REVISIONES TÉCNICAS FORMALES

Una revisión técnica formal (RTF) es una actividad de garantía de calidad del software llevada a cabo por los ingenieros del software (y otros). Los objetivos de la RTF son: (1) descubrir errores en la función, la lógica o la implementación de cualquier representación del software; (2) verificar que el software bajo revisión alcanza sus requisitos; (3) garantizar que el software ha sido representado de acuerdo con ciertos estándares predefinidos; (4) conseguir un software

desarrollado de forma uniforme y (5) hacer que los proyectos sean más manejables. Además, la RTF sirve como campo de entrenamiento, permitiendo que los ingenieros más jóvenes puedan observar los diferentes enfoques de análisis, diseño e implementación del software. La RTF también sirve para promover la seguridad y la continuidad, ya que varias personas se familiarizarán con partes del software que, de otro modo, no hubieran visto nunca.

<sup>3</sup> Aunque estos datos son de hace más de 20 años, pueden ser aplicados en un contexto moderno.

La RTF es realmente una clase de revisión que incluye recorridos, inspecciones, revisiones cíclicas y otro pequeño grupo de evaluaciones técnicas del software. Cada RTF se lleva a cabo mediante una reunión y sólo tendrá éxito si es bien planificada, controlada y atendida. En las siguientes secciones se presentan directrices similares a las de las inspecciones [FRE90, GIL93] como representativas para las revisiones técnicas formales.

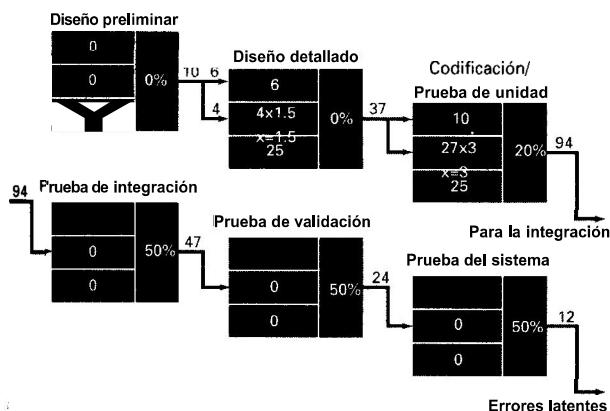


FIGURA 8.3. Amplificación de defectos, sin revisiones.

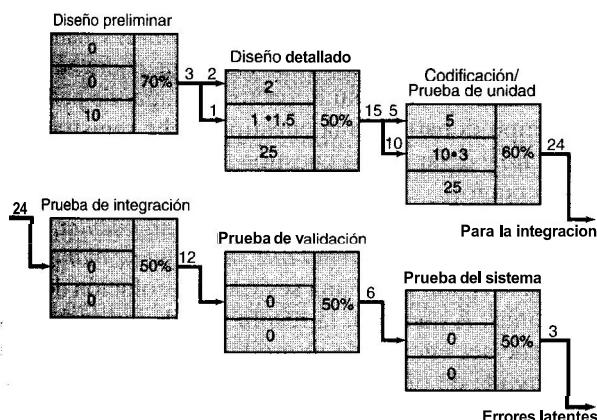


FIGURA 8.4. Amplificación de defectos, llevando a cabo revisiones.

### 8.5.1. La reunión de revisión

Independientemente del formato que se elija para la RTF, cualquier reunión de revisión debe acogerse a las siguientes restricciones:

- deben convocarse para la revisión (normalmente) entre tres y cinco personas;
- se debe preparar por adelantado, pero sin que requiera más de dos horas de trabajo a cada persona; y
- la duración de la reunión de revisión debe ser menor de dos horas.



Cuando realizamos RTF's,  
¿cuáles son nuestros  
objetivos?



**Cita:**

Una reunión es frecuentemente un evento donde se aprovechan unos minutos y se pierden horas.

Author desconocido

Con las anteriores limitaciones, debe resultar obvio que cada RTF se centra en una parte específica (y pequeña) del software total. Por ejemplo, en lugar de intentar revisar un diseño completo, se hacen inspecciones para cada módulo (componente) o pequeño grupo de módulos. Al limitar el centro de atención de la RTF, la probabilidad de descubrir errores es mayor.

El centro de atención de la RTF es un producto de trabajo (por ejemplo, una porción de una especificación de requisitos, un diseño detallado del módulo, un listado del código fuente de un módulo). El individuo que ha desarrollado el producto —el *productor*— informa al jefe del proyecto de que el producto está terminado y que se requiere una revisión. El jefe del proyecto contacta con un *jefe de revisión*, que evalúa la disponibilidad del producto, genera copias del material del producto y las distribuye a dos o tres revisores para que se preparen por adelantado. Cada revisor estará entre una y dos horas revisando el producto, tomando notas y también familiarizándose con el trabajo. De forma concurrente, también el jefe de revisión revisa el producto y establece una agenda para la reunión de revisión que, normalmente, queda convocada para el día siguiente.

### C VE

la RTF se centra en una porción relativamente pequeña de un producto de trabajo.

La reunión de revisión es llevada a cabo por el jefe de revisión, los revisores y el productor. Uno de los revisores toma el papel de registrador, o sea, de persona que registra (de forma escrita) todos los sucesos importantes que se produzcan durante la revisión. La RTF comienza con una explicación de la agenda y una breve introducción a cargo del productor. Entonces el productor procede con el «recorrido de inspección» del producto, explicando el material, mientras que los revisores exponen sus pegas basándose en su preparación previa. Cuando se descubren problemas o errores válidos, el registrador los va anotando.



Puede descargarse el NASA SATC  
Formal Inspection Guidebook de  
[satc.gsfc.nasa.gov/fi/fipage.html](http://satc.gsfc.nasa.gov/fi/fipage.html)

Al final de la revisión, todos los participantes en la **RTF** deben decidir si (1) aceptan el producto sin posteriores modificaciones; (2) rechazan el producto debido a los serios errores encontrados (una vez corregidos, ha de hacerse otra revisión) o (3) aceptan el producto provisionalmente (se han encontrado errores menores que deben ser corregidos, pero sin necesidad de otra posterior revisión). Una vez tomada la decisión, todos los participantes terminan firmando, indicando así que han participado en la revisión y que están de acuerdo con las conclusiones del equipo de revisión.

### 8.5.2. Registro e informe de la revisión

Durante la **RTF**, uno de los revisores (el registrador) procede a registrar todas las pegas que vayan surgiendo. Al final de la reunión de revisión, resume todas las pegas y genera una lista de sucesos de revisión. Además, prepara un informe sumario de la revisión técnica formal. El informe sumario de revisión responde a tres preguntas:

1. ¿Qué fue revisado?
2. ¿Quién lo revisó?
3. ¿Qué se descubrió y cuáles son las conclusiones?

El informe sumario de revisión es una página simple (con posibles suplementos). Se adjunta al registro histórico del proyecto y puede ser enviada al jefe del proyecto y a otras partes interesadas.



Informe Sumario y lista de Sucesos de Revisión Técnica

*La lista de sucesos de revisión* sirve para dos propósitos: (1) identificar áreas problemáticas dentro del producto y (2) servir como lista de comprobación de puntos de acción que guíe al productor para hacer las correcciones. Normalmente se adjunta una lista de conclusiones al informe sumario.

Es importante establecer un procedimiento de seguimiento que asegure que los puntos de la lista de sucesos son corregidos adecuadamente. A menos que se haga así, es posible que las pegas surgenidas «caigan en saco roto». Un enfoque consiste en asignar la responsabilidad del seguimiento al revisor jefe

### 8.5.3. Directrices para la revisión

Se deben establecer de antemano directrices para conducir las revisiones técnicas formales, distribuyéndolas después entre los revisores, para ser consensuadas y, finalmente, seguidas. A menudo, una revisión incontrolada puede ser peor que no hacer ningún tipo de revisión. A continuación se muestra un conjunto mínimo de directrices para las revisiones técnicas formales:

- 1 *Revisar el producto, no al productor.* Una **RTF** involucra gente y egos. Conducida adecuadamente, la **RTF** debe llevar a todos los participantes a un sentimiento agradable de estar cumpliendo su deber. Si se lleva a cabo incorrectamente, la **RTF** puede tomar el aura de inquisición. Se deben señalar los errores educadamente; el tono de la reunión debe ser distendido y constructivo; no debe intentarse dificultar o batallar. El jefe de revisión debe moderar la reunión para garantizar que se mantiene un tono y una actitud adecuados y debe inmediatamente cortar cualquier revisión que haya escapado al control.



*No señale bruscamente los errores. Una forma de ser educada es hacer una pregunta que permita al productor descubrir su propia errar.*

2. *Fijar una agenda y mantenerla.* Un mal de las reuniones de todo tipo es la *deriva*. La **RTF** debe seguir un plan de trabajo concreto. El jefe de revisión es el que carga con la responsabilidad de mantener el **plan** de la reunión y no debe tener miedo de cortar a la gente cuando se empiece a divagar.
3. *Limitar el debate y las impugnaciones.* Cuando un revisor ponga de manifiesto una pega, podrá no haber unanimidad sobre su impacto. En lugar de perder el tiempo debatiendo la cuestión, debe registrarse el hecho y dejar que la cuestión se lleve a cabo en otro momento.
4. *Enunciar áreas de problemas, pero no intentar resolver cualquier problema que se ponga de manifiesto.* Una revisión no es una sesión de resolución de problemas. A menudo, la resolución de los problemas puede ser encargada al productor por sí solo o con la ayuda de otra persona. La resolución de los problemas debe ser pospuesta para después de la reunión de revisión.



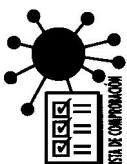
*Es una de las compensaciones más bonitas de la vida, que ningún hombre puede sinceramente intentar ayudar a otro sin ayudarse a sí mismo.*

Ralph Waldo Emerson

5. *Tomar notas escritas.* A veces es buena idea que el registrador tome las notas en una pizarra, de forma que las declaraciones o la asignación de prioridades pueda ser comprobada por el resto de los revisores, a medida que se va registrando la información.
6. *Limitar el número de participantes e insistir en la preparación anticipada.* Dos personas son mejores que una, pero catorce no son necesariamente mejores que cuatro. Se ha de mantener el número de participantes en el mínimo necesario. Además, todos los miembros

del equipo de revisión deben prepararse por anticipado. El jefe de revisión debe solicitar comentarios (que muestren que cada revisor ha revisado el material).

7. *Desarrollar una lista de comprobación para cada producto que haya de ser revisado.* Una lista de comprobaciones ayuda al jefe de revisión a estructurar la reunión de RTF y ayuda a cada revisor a centrarse en los asuntos importantes. Se deben desarrollar listas de comprobaciones para los documentos de análisis, de diseño, de codificación e incluso de prueba.



listas de comprobación de RTF

8. *Disponer recursos y una agenda para las RTF.* Para que las revisiones sean efectivas, se deben planificar como una tarea del proceso de ingeniería del software. Además se debe trazar un plan de actuación para las modificaciones inevitables que aparecen como resultado de una RTF.

9. *Llevar a cabo un buen entrenamiento de todos los revisores.* Por razones de efectividad, todos los participantes en la revisión deben recibir algún entrenamiento formal. El entrenamiento se debe basar en los aspectos relacionados con el proceso, así como las consideraciones de psicología humana que atañen a la revisión. Freedman y Weinberg [FRE90] estiman en un mes la curva de aprendizaje para cada veinte personas que vayan a participar de forma efectiva en las revisiones.

10. *Repasar las revisiones anteriores.* Las sesiones de información pueden ser beneficiosas para descubrir problemas en el propio proceso de revisión. El primer producto que se haya revisado puede establecer las propias directivas de revisión.

Como existen muchas variables (por ejemplo, número de participantes, tipo de productos de trabajo, tiempo y duración, enfoque de revisión específico) que tienen impacto en una revisión satisfactoria, una organización de software debería determinar qué método funciona mejor en un contexto local. Porter y sus colegas [PÓR95] proporcionan una guía excelente para este tipo de experimentos.

## 8 GARANTÍA DE CALIDAD ESTADÍSTICA

*La garantía de calidad estadística* refleja una tendencia, creciente en toda la industria, a establecer la calidad **más** cuantitativamente. Para el software, la garantía de calidad estadística implica los siguientes pasos:

1. Se agrupa y se clasifica la información sobre los defectos del software.
2. Se intenta encontrar la causa subyacente de cada defecto (por ejemplo, no concordancia con la especificación, error de diseño, incumplimiento de los estándares, pobre comunicación con el cliente).



¿Qué pasos se requieren para desarrollar una SQA estadística?

3. Mediante el principio de Pareto (el 80 por 100 de los defectos se pueden encontrar en el 20 por 100 de todas las posibles causas), se aísla el 20 por 100 (los «pocos vitales»).
4. Una vez que se han identificado los defectos vitales, se actúa para corregir los problemas que han producido los defectos.

Este concepto relativamente sencillo representa un paso importante hacia la creación de un proceso de ingeniería del software adaptativo en el cual se realizan cambios para mejorar aquellos elementos del proceso que introducen errores.



**Cita:**  
El 20 por 100 del código tiene el 80 por 100 de los defectos. ¡Encuentrelos, corríjalos!

Jewell Arthur

Para ilustrar el proceso, supongamos que una organización de desarrollo de software recoge información sobre defectos durante un período de un año. Algunos de los defectos se descubren mientras se desarrolla el software. Otros se encuentran después de que el software se haya distribuido al usuario final. Aunque se descubren cientos de errores diferentes, todos se pueden encontrar en una (o más) de las siguientes causas:

- Especificación incompleta o errónea (EIE).
- Mala interpretación de la comunicación del cliente (MCC).
- Desviación deliberada de la especificación (DDE).
- Incumplimiento de los estándares de programación (IEP).
- Error en la representación de los datos (ERD).
- Interfaz de módulo inconsistente (IMI).
- Error en la lógica de diseño (ELD).
- Prueba incompleta o errónea (PIE).
- Documentación imprecisa o incompleta (DII).
- Error en la traducción del diseño al lenguaje de programación (TLP).

- Interfaz hombre-máquina ambigua o inconsistente (IHM).
- Varios (VAR).



La Asociación China de Calidad del Software presenta uno de los sitios web más completos para la calidad en [www.cosq.org](http://www.cosq.org)

Para aplicar la SQA estadística se construye la Tabla 8.1. La tabla indica que EIE, MCC y ERD son las *causas vitales* que contabilizan el 53 por 100 de todos los errores. Sin embargo, debe observarse que si sólo se consideraran errores serios, se seleccionarían las siguientes causas vitales: EIE, ERD, TLPy ELD. Una vez determinadas las causas vitales, la organización de desarrollo de software puede comenzar la acción correctiva. Por ejemplo, para corregir la MCC, el equipo de desarrollo del software podría implementar técnicas que facilitaran la especificación de la aplicación (Capítulo 11) para mejorar la calidad de la especificación y la comunicación con el cliente. Para mejorar el ERD, el equipo de desarrollo del software podría adquirir herramientas CASE para la modelización de datos y realizar revisiones del diseño de datos más rigurosas.

Es importante destacar que la acción correctiva se centra principalmente en las causas vitales. Cuando éstas son corregidas, nuevas candidatas saltan al principio de la lista.

Se han mostrado las técnicas de garantía de calidad del software estadísticas para proporcionar una mejora sustancial en la calidad [ART97]. En algunos casos, las organizaciones de software han conseguido una reducción anual del 50 por 100 de los errores después de la aplicación de estas técnicas.

Junto con la recopilación de información sobre defectos, los equipos de desarrollo del software pueden calcular un *índice de errores* (IE) para cada etapa principal del proceso de ingeniería del software [IEE94]. Después del análisis, el diseño, la codificación, la prueba y la entrega, se recopilan los siguientes datos:

$E_i$  = número total de defectos descubiertos durante la etapa i-ésima del proceso de ingeniería del software;

$S_i$  = número de defectos graves;

$M_i$  = número de defectos moderados;

$T_i$  = número de defectos leves;

$PS$  = tamaño del producto (LDC, sentencias de diseño, páginas de documentación) en la etapa i-ésima.

$W_s, W_m, W_t$  = factores de peso de errores graves, moderados, y leves, en donde los valores recomendados son  $W_s = 10, W_m = 3, W_t = 1$ . Los factores de peso de cada fase deberían agrandarse a medida que el desarrollo evoluciona. Esto favorece a la organización que encuentra los errores al principio.

En cada etapa del proceso de ingeniería del software se calcula un *índice defase*,  $IF_i$ :

$$IF_i = W_s (S_i/E_i) + W_m (M_i/E_i) + W_t (T_i/E_i)$$

El *índice de errores* (IE) se obtiene mediante el cálculo del defecto acumulativo de cada  $IF_i$ , asignando más peso a los errores encontrados más tarde en el proceso de ingeniería del software, que a los que se encuentran en las primeras etapas:

$$\begin{aligned} IE &= \sum (i \times IF_i) / PS \\ &= (IF_1 + 2IF_2 + 3IF_3 + \dots + iIF_i) / PS \end{aligned}$$

| Error   | Total |      | Grave |      | Moderado |      | Leve |      |
|---------|-------|------|-------|------|----------|------|------|------|
|         | No.   | %    | No.   | %    | No.      | %    | No.  | %    |
| IIE     | 205   | 22%  | 34    | 27%  | 68       | 18%  | 103  | 24%  |
| MCC     | 156   | 17%  | 12    | 9%   | 68       | 18%  | 76   | 17%  |
| DDE     | 48    | 5%   | 1     | 1%   | 24       | 6%   | 23   | 5%   |
| IEP     | 25    | 3%   | 0     | 0%   | 15       | 4%   | 10   | 2%   |
| ERD     | 130   | 14%  | 26    | 20%  | 68       | 18%  | 36   | 8%   |
| IMI     | 58    | 6%   | 9     | 7%   | 18       | 5%   | 31   | 7%   |
| ELD     | 45    | 5%   | 14    | 11%  | 12       | 3%   | 19   | 4%   |
| PIE     | 95    | 10%  | 12    | 9%   | 35       | 9%   | 48   | 11%  |
| DII     | 36    | 4%   | 2     | 2%   | 20       | 5%   | 14   | 3%   |
| TLP     | 60    | 6%   | 15    | 12%  | 19       | 5%   | 26   | 6%   |
| IHM     | 28    | 3%   | 3     | 2%   | 17       | 4%   | 8    | 2%   |
| VAR     | 56    | 6%   | 0     | 0%   | 15       | 4%   | 41   | 9%   |
| Totales | 942   | 100% | 128   | 100% | 379      | 100% | 435  | 100% |

TABLA 8.1. Recolección de datos para la SQA estadística

Se puede utilizar el índice de errores junto con la información recogida en la Tabla 8.1, para desarrollar una indicación global de la mejora en la calidad del software.

La aplicación de la SQA estadística y el principio de Pareto se pueden resumir en una sola frase: *j Utilizar el tiempo para centrarse en cosas que realmente interesan, pero primero asegurarse que se entiende qué es lo que realmente interesa!*

Un estudio exhaustivo de la SQA estadística se sale del alcance de este libro. Los lectores interesados deberían consultar [SCH98], [KAP95] y [KAN95].

## FIABILIDAD DEL SOFTWARE

No hay duda de que la fiabilidad de un programa de computadora es un elemento importante de su calidad general. Si un programa falla frecuente y repetidamente en su funcionamiento, no importa si el resto de los factores de calidad son aceptables.



### Referencia Web

El Centro de Análisis de Fiabilidad proporciona mucha información útil sobre la fiabilidad, mantenibilidad, soporte y calidad en [rac.iitri.org](http://rac.iitri.org)

La fiabilidad del software, a diferencia de otros factores de calidad, puede ser medida o estimada mediante datos históricos o de desarrollo. La **fiabilidad del software** se define en términos estadísticos como «la probabilidad de operación libre de fallos de un programa de computadora en un entorno determinado y durante un tiempo específico» [MUS87]. Por ejemplo, un programa X puede tener una fiabilidad estimada de 0,96 durante un intervalo de proceso de ocho horas. En otras palabras, si se fuera a ejecutar el programa X 100 veces, necesitando ocho horas de tiempo de proceso (tiempo de ejecución), lo probable es que funcione correctamente (sin fallos) 96 de cada 100 veces.

Siempre que se habla de fiabilidad del software, surge la pregunta fundamental: ¿Qué se entiende por el término fallo? En el contexto de cualquier discusión sobre calidad y fiabilidad del software, el fallo es cualquier falta de concordancia con los requisitos del software. Incluso en esta definición existen grados. Los fallos pueden ser simplemente desconcertantes o ser catastróficos. Puede que un fallo sea corregido en segundos mientras que otro lleve semanas o incluso meses. Para complicar más las cosas, la corrección de un fallo pue-

llevar a la introducción de otros errores que, finalmente, lleven a más fallos.

### 7.1. Medidas de fiabilidad y de disponibilidad

Los primeros trabajos sobre fiabilidad intentaron extraer las matemáticas de la teoría de fiabilidad del hardware (por ejemplo: [ALV64]) a la predicción de la fiabilidad del software. La mayoría de los modelos de fiabilidad relativos al hardware van más orientados a los fallos debidos al desajuste que a los fallos debidos a defectos de diseño. En el hardware, son más probables los fallos debidos al desgaste físico (por ejemplo: el efecto de la temperatura, de la corrosión y los golpes) que los fallos relativos al diseño. Desgraciadamente, para el software lo que ocurre es lo contrario. De hecho, todos los fallos del software, se producen por problemas de diseño o de implementación; el desajuste (consulte el Capítulo 1) no entra en este panorama.

### CUADRO CLAVE

Los problemas de la fiabilidad del software se deben casi siempre a errores en el diseño o en la implementación.

Todavía se debate sobre la relación entre los conceptos clave de la fiabilidad del hardware y su aplicabilidad al software (por ejemplo, [LIT89], [ROO90]). Aunque aún falta por establecer un nexo irrefutable, merece la pena considerar unos cuantos conceptos que conciernen a ambos elementos de los sistemas.

Considerando un sistema basado en computadora, una sencilla medida de la fiabilidad es el tiempo medio entre fallos (TMEF), donde;

$$\text{TMEF} = \text{TMDF} + \text{TMDR}$$

Las siglas TMDF y TMDR corresponden a tiempo medio de fallo y tiempo medio de reparación, respectivamente.



¿Por qué es TMEF una métrica más útil que errores/LDC?

Muchos investigadores argumentan que el TMDF es, con mucho, una medida más útil que los defectos/KLDC o defectos/PF. Sencillamente, el usuario final se enfrenta a los fallos, no al número total de errores. Como cada error de un programa no tiene la misma tasa de fallo, la cuenta total de errores no es una buena indicación de la fiabilidad de un sistema. Por ejemplo, consideremos un programa que ha estado funcionando durante 14 meses. Muchos de los errores del programa pueden pasar desapercibidos durante décadas antes de que se detecten. El TMEF de esos errores puede ser de 50 e incluso de 100 años. Otros errores, aunque no se hayan descubierto aún, pueden tener una tasa de fallo de 18 ó 24 meses. Incluso aunque se eliminan todos los errores de la primera categoría (los que tienen un gran TMEF), el impacto sobre la fiabilidad del software será muy escaso.

Además de una medida de la fiabilidad debemos obtener una medida de la disponibilidad. La disponibilidad del software es la probabilidad de que un programa funcione de acuerdo con los requisitos en un momento dado, y se define como:

$$\text{Disponibilidad} = [\text{TMDF}/(\text{TMDF} + \text{TMDR})] \times 100\%$$

La medida de fiabilidad TMEF es igualmente sensible al TMDF que al TMDR. La medida de disponibilidad es algo más sensible al TMDR, una medida indirecta de la facilidad de mantenimiento del software.

### 8.7.2. Seguridad del software

Leveson [LEV86] discute el impacto del software en sistemas críticos de seguridad, diciendo:

Antes de que se usara software en sistemas críticos de seguridad, normalmente éstos se controlaban mediante dispositivos electrónicos y mecánicos convencionales (no programables). Las técnicas de seguridad de sistemas se diseñaban para hacer frente a fallos aleatorios en esos dispositivos [no programables]. Los errores humanos de diseño no se consideraban porque se suponía que todos los defectos producidos por los errores humanos se podían evitar o eliminar completamente antes de su distribución y funcionamiento.



#### Cita:

No puedo imaginar cualquier condición que podría causar el hundimiento de este barco. La construcción naval moderna ha ido más allá de eso.

E. J. Smith, Capitán del Titanic

Cuando se utiliza el software como parte del sistema de control, la complejidad puede aumentar en un orden de magnitud o más. Los defectos sutiles de diseño, producidos por un error humano —algo que se puede descubrir y eliminar en el control convencional basado en el hardware— llegan a ser mucho más difíciles de descubrir cuando se utiliza el software.

La *seguridad del software* es una actividad de garantía de calidad del software que se centra en la identificación y evaluación de los riesgos potenciales que pueden producir un impacto negativo en el software y hacer que falle el sistema completo. Si se pueden identificar pronto los riesgos en el proceso de ingeniería del software podrán especificarse las características del diseño del software que permitan eliminar o controlar los riesgos potenciales.

Como parte de la seguridad del software, se puede dirigir un proceso de análisis y modelado. Inicialmente, se identifican los riesgos y se clasifican por **su** importancia y **su** grado de riesgo. Por ejemplo, algunos de los riesgos asociados con el control basado en computadora del sistema de conducción de un automóvil podrían ser:

- Produce una aceleración incontrolada que no se puede detener.
- No responde a la presión del pedal del freno (deteniéndose).
- No responde cuando se activa el contacto.
- Pierde o gana velocidad lentamente.

Cuando se han identificado estos riesgos del sistema, se utilizan técnicas de análisis para asignar su gra-



#### Referencia Web

Se pueden encontrar documentos sobre la seguridad del software (y un glosario detallado) que merecen la pena en [www.rstcorp.com/hotlist/topics-safety.html](http://www.rstcorp.com/hotlist/topics-safety.html)

vedad y **su** probabilidad de ocurrencia<sup>4</sup>. Para que sea efectivo, se tiene que analizar el software en el contexto del sistema completo. Por ejemplo, puede que un sutil error en la entrada del usuario (las personas son componentes del sistema) se magnifique por un fallo del software que producen los datos de control que actúan de forma inadecuada sobre un dispositivo mecánico. Si se dan un conjunto de condiciones externas del entorno (y sólo si se dan), la situación inadecuada del dispositivo mecánico producirá un fallo desastroso. Se pueden usar técnicas de análisis, como el *análisis del árbol de fallos* [VES81], la *lógica de tiempo real* [JAN86] o los *modelos de redes de Petri* [LEV87], para predecir la cadena de sucesos que pueden producir los riesgos y la probabilidad de que se dé cada uno de los sucesos que componen la cadena.

Una vez que se han identificado y analizado los riesgos, se pueden especificar los requisitos relacionados con la seguridad para el software. Esto es, la especificación puede contener una lista de eventos no deseables y las respuestas del sistema a estos eventos. El papel del software en la gestión de eventos no deseados es entonces apropiado.



#### ¿Cuál es la diferencia entre fiabilidad del software y seguridad del software?

Aunque la fiabilidad y la seguridad del software están bastante relacionadas, es importante entender la sutil diferencia que existe entre ellas. La fiabilidad del software utiliza el análisis estadístico para determinar la probabilidad de que pueda ocurrir un fallo del software. Sin embargo, la ocurrencia de un fallo no lleva necesariamente a un riesgo o a un accidente. La seguridad del software examina los modos según los cuales los fallos producen condiciones que pueden llevar a accidentes. Es decir, los fallos no se consideran en vacío, sino que se evalúan en el contexto de un completo sistema basado en computadora.

Un estudio completo sobre seguridad del software y análisis del riesgo va más allá del ámbito de este libro. Para aquellos lectores que estén más interesados, deberían consultar el libro de Leveson [LEV95] sobre este tema.

<sup>4</sup> Este método es análogo al método de análisis de riesgos descrito para la gestión del proyecto de software en el capítulo 6. La principal diferencia es el énfasis en aspectos de tecnología frente a temas relacionados con el proyecto.

## 8 PRUEBA DE ERRORES PARA SOFTWARE

Si William Shakespeare hubiera escrito sobre las condiciones del ingeniero de software moderno, podría perfectamente haber escrito: «Errar es humano, encontrar el error rápida y correctamente es divino.» En los años sesenta, un ingeniero industrial japonés, Shigeo Shingo [SHI86], que trabajaba en Toyota, desarrolló una técnica de garantía de calidad que conducía a la prevención y/o a la temprana corrección de errores en el proceso de fabricación. Denominado *poka-yoke* (prueba de errores), el concepto de Shingo hacía uso de dispositivos *poka-yoke* —mecanismos que conducen (1) a la prevención de un problema de calidad potencial antes de que éste ocurra, o (2) a la rápida detección de problemas de calidad si se han introducido ya—. Nosotros encontramos dispositivos *poka-yoke* en nuestra vida cotidiana (incluso si nosotros no tenemos conciencia de este concepto). Por ejemplo, el interruptor de arranque de un coche no trabaja si está metida una marcha en la transmisión automática (un *dispositivo de prevención*); un pitido de aviso del coche sonará si los cinturones de seguridad no están bien sujetos (un *dispositivo de detección de fallos*).

Un dispositivo de *poka-yoke* presenta un conjunto de características comunes:

- Es simple y barato —si un dispositivo es demasiado complicado y caro, no será efectivo en cuanto a costo—;
- Es parte del proceso —esto es, el dispositivo *poka-yoke* está integrado en una actividad de ingeniería—;
- Está localizado cerca de la tarea del proceso donde están ocurriendo los errores —por consiguiente proporciona una realimentación rápida en cuanto a la corrección de errores se refiere—.



### Referencia Web

Se puede obtener una colección completa de recursos de *poka-yoke* en  
[www.campbell.berry.edu/faculty/igrout/pokayoke.shtml](http://www.campbell.berry.edu/faculty/igrout/pokayoke.shtml)

Aunque el *poka-yoke* fue originariamente desarrollado para su uso en «control de calidad cero» [SHI86] para el hardware fabricado, puede ser adaptado para su uso en ingeniería del software. Para ilustrar esto, consideremos el problema siguiente [ROB97]:

Una compañía de productos de software vende el software de aplicación en el mercado internacional. Los menús desplegables y todos los códigos asociados proporcionados con cada aplicación deben reflejar el lenguaje que se emplea en el lugar donde se usa. Por ejemplo, el elemento del menú del lenguaje inglés para «Close», tiene el mnemónico «C» asociado con ello. Cuando la aplicación se vende en un país de habla francesa, el mismo elemento del menú es «Fermer» con el mnemónico «F».

Para llevar a cabo la entrada adecuada del menú para cada aplicación, un «localizador» (una persona que habla en el idioma local y con la terminología de ese lugar) traduce los menús de acuerdo con el idioma en uso. El problema es asegurar que (1) cada entrada de menú (pueden existir cientos de ellas) se adapte a los estándares apropiados y que no existan conflictos, independientemente del lenguaje que se está utilizando.

El uso del *poka-yoke* para la prueba de diversos menús de aplicación desarrollados en diferentes lenguajes, tal y como se ha descrito anteriormente, se discute en un artículo de Harry Robinson [ROB97]:

Nosotros primero decidimos dividir el problema de pruebas de menús en partes que puedan ser resueltas más fácilmente. Nuestro primer avance para la resolución del problema fue el comprender que existían dos aspectos separados para los catálogos de mensaje. Había por una parte el aspecto de contenidos: las traducciones de texto muy simples tales como cambiar meramente «Close» por la palabra «Fermer». Puesto que el equipo que realizaba las comprobaciones no hablaba de forma fluida el lenguaje en el que se pretendía trabajar, teníamos que dejar estos aspectos a traductores expertos del lenguaje.

El segundo aspecto de los catálogos del mensaje era la estructura, las reglas sintácticas que debía obedecer un catálogo de objetivos adecuadamente construido. A diferencia del contenido, en este caso sí que era posible para el equipo de comprobación el verificar los aspectos estructurales de los catálogos.

Como un ejemplo de lo que significa estructura, consideremos las etiquetas y los mnemónicos de un menú de aplicación. Un menú está constituido por etiquetas y sus mnemónicos (abreviaturas) asociados. Cada menú, independientemente de su contenido o su localización, debe obedecer las siguientes reglas listadas en la Guía de Estilo Motif

- Cada nemotécnico debe estar contenido en su etiqueta asociada;
- Cada nemotécnico debe ser único dentro del menú;
- Cada nemotécnico debe ser un carácter Único;
- Cada nemotécnico debe estar en ASCII.

Estas reglas son invariantes a través de las localizaciones, y pueden ser utilizadas para verificar que un menú está correctamente construido en la localización objetivo.

Existen varias posibilidades para realizar la prueba de errores de los mnemónicos del menú:

*Dispositivo de prevención.* Nosotros podemos escribir un programa para generar los mnemónicos automáticamente, dada una lista de etiquetas en cada menú. Este enfoque evitaría errores, pero el problema es que escoger un mnemónico adecuado es difícil, y el esfuerzo requerido para escribir el programa no estaría justificado con el beneficio obtenido.

*Dispositivo de prevención.* Podríamos escribir un programa que prevendría al localizador de elegir unos mnemónicos que no satisfagan el criterio. Este enfoque también evitaría errores, pero el beneficio obtenido sería mínimo: los mnemónicos incorrectos son lo suficiente-

mente fáciles de detectar y corregir después de que aparezcan.

**Dispositivo de detección.** Nosotros podríamos proporcionar un programa para verificar que las etiquetas del menú escogidas y los mnemónicos satisfacen los criterios anteriores. Nuestros localizadores podrían ejecutar los programas sobre los catálogos de mensaje traducidos antes de enviarnos a nosotros dichos catálogos. Este enfoque proporcionaría una realimentación rápida sobre los errores y será, probablemente, un paso a dar en el futuro.

**Dispositivo de detección.** Nosotros podríamos escribir un programa que verificase las etiquetas y mnemónicos del menú, y ejecutara el programa sobre catálogos de mensajes después de que nos los hayan devuelto los localizadores. Este enfoque es el camino que actualmente estamos tomando. No es tan eficiente como alguno de los métodos anteriormente mencionados y puede requerir que se establezca una comunicación fluida hacia delante y hacia atrás con los localizadores, pero los errores detectados son incluso fáciles de corregir en este punto.

Varios textos pequeños de poka-yoke se utilizaron como dispositivos poka-yoke para validar los aspectos

estructurales de los menús. Un pequeño «script» *poka-yoke* leería la tabla, recuperaría los mnemónicos y etiquetas a partir del catálogo de mensajes, y compararía posteriormente las cadenas recuperadas con el criterio establecido descrito anteriormente.

Los «scripts» *poka-yoke* eran pequeños (aproximadamente 100 líneas), fáciles de escribir (algunos de los escritos estaban en cuanto al tiempo por debajo de una hora) y fáciles de ejecutar. Nosotros ejecutábamos nuestros «scripts» *poka-yoke* sobre 16 aplicaciones en la ubicación en inglés por defecto y en 11 ubicaciones extranjeras. Cada ubicación contenía 100 menús para un total de 1.200 menús. Los dispositivos *poka-yoke* encontraron 311 errores en menús y mnemónicos. Pocos de los problemas que nosotros descubrimos eran como muy llamativos, pero en total habrían supuesto una gran preocupación en la prueba y ejecución de nuestras aplicaciones localizadas.

El ejemplo descrito antes representa un dispositivo *poka-yoke* que ha sido integrado en la actividad de pruebas de ingeniería del software. La técnica *poka-yoke* puede aplicarse a los niveles de diseño, codificación y pruebas y proporciona un filtro efectivo de garantía de calidad.

## 8.9 EL ESTÁNDAR DE CALIDAD ISO 9001

Esta sección contiene varios objetivos, siendo el principal describir el cada vez más importante estándar internacional ISO 9001. El estándar, que ha sido adoptado por más de 130 países para su uso, se está convirtiendo en el medio principal con el que los clientes pueden juzgar la competencia de un desarrollador de software. Uno de los problemas con el estándar ISO 9001 está en que no es específico de la industria: está expresado en términos generales, y puede ser interpretado por los desarrolladores de diversos productos como cojinetes de bolas (rodamientos), secadores de pelo, automóviles, equipamientos deportivos y televisiones, así como por desarrolladores de software. Se han realizado muchos documentos que relacionan el estándar con la industria del software, pero no entran en una gran cantidad de detalles. El objetivo de esta sección es describir lo que significa el ISO 9001 en términos de elementos de calidad y técnicas de desarrollo.

Para la industria del software los estándares relevantes son:

- ISO 9001. Quality Systems-Model for Quality Assurance in Design, Development, Production, Installation and Servicing. Este es un estándar que describe el sistema de calidad utilizado para mantener el desarrollo de un producto que implique diseño.
- ISO 9000-3. Guidelines for Application of ISO 9001 to the Development, Supply and Maintenance of Software. Este es un documento específico que interpreta el ISO 9001 para el desarrollador de software.

- ISO 9004-2. Quality Management and Quality System Elements —Part 2—. Este documento proporciona las directrices para el servicio de facilidades del software como soporte de usuarios.

Los requisitos se agrupan bajo 20 títulos:

- Responsabilidad de la gestión.
- Inspección, medición y equipo de pruebas.
- Sistema de calidad.
- Inspección y estado de pruebas.
- Revisión de contrato.
- Acción correctiva.
- Control de diseño.
- Control de producto no aceptado.
- Control de documento.
- Tratamiento, almacenamiento, empaquetamiento y entrega.
- Compras.
- Producto proporcionado al comprador.
- Registros de calidad.
- Identificación y posibilidad de seguimiento del producto, Auditorías internas de calidad.
- Formación
- Control del proceso
- Servicios.
- Inspección y estado de prueba.
- Técnicas estadísticas.

Merece la pena ver un pequeño extracto de la ISO 9001. Este le dará al lector una idea del nivel con el que la ISO 9001 trata la garantía de calidad y el proceso de

desarrollo. El extracto elegido proviene de la sección 1.11:

#### **4.11. El equipo de Inspección, medición y pruebas.**

**El suministrador debe controlar, calibrar y mantenerla inspección, medir y probar el equipo, ya sea el dueño el suministrador, prestado o proporcionado por el comprador, para demostrar la conformidad del producto con los requisitos específicos. El equipo debe utilizarse de un modo que asegure que se conoce la incertidumbre de la medición y que es consistente con la capacidad de medición requerida.**

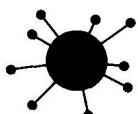
Lo primero a destacar es su generalidad; se puede aplicar al desarrollador de cualquier producto. Lo segundo a considerar es la dificultad en la interpreta-

ción del párrafo —es pretendido obviamente por los procesos estándar de ingeniería donde equipos tales como indicadores de calibración y potenciómetros son habituales—.

Una interpretación del párrafo anterior es que el distribuidor debe asegurar que cualquiera de las herramientas de software utilizadas para las pruebas tiene, por lo menos, la misma calidad que el software a desarrollar, y que cualquier prueba del equipo produce valores de medición, por ejemplo, los monitores del rendimiento, tienen una precisión aceptable cuando se compara con la precisión especificada para el rendimiento en la especificación de los requisitos.

## **10 EL PLAN DE SQA**

El *plan de SQA* proporciona un mapa para institucionalizar la garantía de calidad del software. El plan, desarrollado por un grupo de SQA, sirve como plantilla para actividades de SQA instituidas para cada proyecto de software.



**El Plan GCS**

El IEEE [IEEE94] ha recomendado un estándar para los planes de SQA. Las secciones iniciales describen el propósito y el alcance del documento e indican aquellas actividades del proceso del software cubiertas por la garantía de calidad. Se listan todos los documentos señalados en el *plan de SQA* y se destacan todos los estándares aplicables. La sección de *Gestión* del plan describe la situación de la SQA dentro de la estructura organizativa; las tareas y las actividades de SQA y su emplazamiento a lo largo del proceso del software; así como los papeles y responsabilidades organizativas relativas a la calidad del producto.

La sección de *Documentación* describe (por referencia) cada uno de los productos de trabajo producidos como parte del proceso de software. Entre estos se incluyen:

- documentos del proyecto (por ejemplo: plan del proyecto),
- modelos (por ejemplo: DERs, jerarquías de clases),
- documentos técnicos (por ejemplo: especificaciones, planes de prueba),

- documentos de usuario (por ejemplo: archivos de ayuda).

Además, esta sección define el conjunto mínimo de productos de trabajo que se pueden aceptar para lograr alta calidad.

Los *estándares, prácticas y convenciones* muestran todos los estándares/prácticas que se aplican durante el proceso de software (por ejemplo: estándares de documentos, estándares de codificación y directrices de revisión). Además, se listan todos los proyectos, procesos y (en algunos casos) métricas de producto que se van a recoger como parte del trabajo de ingeniería del software.

La sección *Revisões y Auditorías* del plan identifica las revisiones y auditorías que se van a llevar a cabo por el equipo de ingeniería del software, el grupo de SQA y el cliente. Proporciona una visión general del enfoque de cada revisión y auditoría.

La sección *Prueba* hace referencia al *Plan y Procedimiento de Pruebas del Software* (Capítulo 18). También define requisitos de mantenimiento de registros de pruebas. La *Información sobre problemas y acción correctiva* define procedimientos para informar, hacer seguimiento y resolver errores y defectos, e identifica las responsabilidades organizativas para estas actividades.

El resto del *plan de SQA* identifica las herramientas y métodos que soportan actividades y tareas de SQA; hace referencia a los procedimientos de gestión de configuración del software para controlar el cambio; define un enfoque de gestión de contratos; establece métodos para reunir, salvaguardar y mantener todos los registros; identifica la formación que se requiere para cumplir las necesidades del plan y define métodos para identificar, evaluar, supervisar y controlar riesgos.

## RESUMEN

La garantía de calidad del software es una «actividad de protección» que se aplica a cada paso del proceso del software. La **SQA** comprende procedimientos para la aplicación efectiva de métodos y herramientas, revisiones técnicas formales, técnicas y estrategias de prueba, dispositivos *poku-yoke*, procedimientos de control de cambios, procedimientos de garantía de ajuste a los estándares y mecanismos de medida e información.

La **SQA** es complicada por la compleja naturaleza de la calidad del software —un atributo de los programas de computadora que se define como «concordancia con los requisitos definidos explícita e implícitamente»—. Cuando se considera de forma más general, la calidad del software engloba muchos factores de proceso y de producto diferentes con sus métricas asociadas.

Las revisiones del software son una de las actividades más importantes de la **SQA**. Las revisiones sirven como filtros durante todas las actividades de ingeniería del software, eliminando defectos mientras que no son relativamente costosos de encontrar y corregir. La revisión técnica formal es una revisión específica que se ha

mostrado extremadamente efectiva en el descubrimiento de errores.

Para llevar a cabo adecuadamente una garantía de calidad del software, se deben recopilar, evaluar y distribuir todos los datos relacionados con el proceso de ingeniería del software. La **SQA** estadística ayuda a mejorar la calidad del producto y la del proceso de software. Los modelos de fiabilidad del software amplían las medidas, permitiendo extrapolar los datos recogidos sobre los defectos, a predicciones de tasas de fallo y de fiabilidad.

Resumiendo, recordemos las palabras de Dunn y Ullman [DUN82]: «La garantía de calidad del software es la guía de los preceptos de gestión y de las disciplinas de diseño de la garantía de calidad para el espacio tecnológico y la aplicación de la ingeniería del software.) La capacidad para garantizar la calidad es la medida de madurez de la disciplina de ingeniería. Cuando se sigue de forma adecuada esa guía anteriormente mencionada, lo que se obtiene *es* la madurez de la ingeniería del software.

## REFERENCIAS

- [ALV64] von Alvin, W. H. (ed.), *Reliability Engineering*, Prentice-Hall, 1964.
- [ANS87] ANSI/ASQC A3-1987, Quality Systems Terminology, 1987.
- [ART92] Arthur, L. J., *Improving Software Quality: An Insider's Guide to TQM*, Wiley, 1992.
- [ART97] Arthur, L. J., «Quantum Improvements in Software System Quality», *CACM*, vol. 40, n.º 6, Junio 1997, pp. 47-52.
- [BOE81] Boehm, B., *Software Engineering Economics*, Prentice-Hall, 1981.
- [CR075] Crosby, P., *Quality is Free*, McGraw-Hill, 1975.
- [CR079] Crosby, P., *Quality is Free*, McGraw-Hill, 1979.
- [DEM86] Deming, W. W., *Out of the Crisis*, MIT Press, 1986.
- [DEM99] DeMarco, T., «Management Can Make Quality (Im)possible», presentación de Cutter Summit '99, Boston, MA, 26 de Abril 1999.
- [DIJ76] Dijkstra, E., *A Discipline of Programming*, Prentice-Hall, 1976.
- [DUN82] Dunn, R., y R. Ullman, *Quality Assurance for Computer Software*, McGraw-Hill, 1982.
- [FRE90] Freedman, D. P., y G. M. Weinberg, *Handbook of Walkthroughs, Inspections and Technical Reviews*, 3.ª ed., Dorset House, 1990.
- [GIL93] Gilb, T., y D. Graham, *Software Inspections*, Addison-Wesley, 1993.
- [GLA98] Glass, R., «Defining Quality Intuitively», *IEEE Software*, Mayo 1998, pp. 103-104 y 107.
- [HOY98] Hoyle, D., *Iso 9000 Quality Systems Development Handbook: A Systems Engineering Approach*, Butterworth-Heinemann, 1998.
- [IBM81] «Implementing Software Inspections», notas del curso, IBM Systems Sciences Institute, IBM Corporation, 1981.
- [IEE90] *Software Quality Assurance: Model Procedures*, Institution of Electrical Engineers, 1990.
- [IEE94] *Software Engineering Standards*, ed. de 1994, IEEE Computer Society, 1994
- [JAN86] Jahanian, F., y A. K. Mok, «Safety Analysis of Timing Properties of Real Time Systems», *IEEE Trans. Software Engineering*, vol. SE-12, n.º 9, Septiembre 1986, pp. 890-904.
- [JON86] Jones, T. C., *Programming Productivity*, McGraw-Hill, 1986.
- [KAN95] Kan, S. H., *Metrics and Models in Software Quality Engineering*, Addison Wesley, 1995.
- [KAP95] Kaplan, C., R. Clark y V. Tang, *Secrets of Software Quality: 40 Innovations from IBM*, McGraw-Hill, 1995.
- [LEV86] Leveson, N.G., «Software Safety: Why, What, and How», *ACM Computing Surveys*, vol. 18, n.º 2, Junio 1986, pp. 125-163.
- [LEV87] Leveson, N. G., y J.L. Stolzy, «Safety Analysis using Petri Nets», *IEEE Trans. Software Engineering*, vol. SE-13, n.º 3, Marzo 1987, pp. 386-397.

- [LEV95] Leveson, N.G., *Safeware: System Safety and Computers*, Addison-Wesley, 1995.
- [LIN79] Linger, R., H. Mills y B. Witt, *Structured Programming*, Addison-Wesley, 1979.
- [LIT89] Littlewood, B., «Forecasting Software Reliability», en *Software Reliability: Modelling and Identification* (S. Bittanti, ed.), Springer-Verlag, 1989, pp. 141-209.
- [MAN96] Manns, T. y M. Coleman, *Software Quality Assurance*, MacMillan Press, 1996.
- [MUS87] Musa, J. D., A. Iannino y K. Okumoto, *Engineering and Managing Software with Reliability Measures*, McGraw-Hill, 1987.
- [PAU93] Paulk, M., et al., *Capability Maturity Model for Software*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1993.
- [POR95] Porter, A., H. Siv, C. A. Toman, y L. G. Votta, «An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development», *Proc. Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Washington DC, Octubre 1996, ACM Press, pp. 92-103.
- [ROB97] Robinson, H., «Using Poka-Yoke Techniques for Early Error Detection», *Proc. Sixth International Conference on Software Testing Analysis and Review (STAR'97)*, 1997, pp. 119-142.
- [ROO90] Rook, J., *Software Reliability Handbook*, Elsevier, 1990.
- [SCH98] Schulmeyer, G. C., y J.I. McManus (eds.), *Handbook of Software Quality Assurance*, Prentice-Hall, 3.<sup>a</sup> ed., 1998.
- [SCH94] Schmauch, C.H., *ISO 9000 for Software Developers*, ASQC Quality Press, Milwaukee, WI, 1994.
- [SCH97] Schoonmaker, S.J., *ISO 9000 for Engineers and Designers*, McGraw Hill, 1997.
- [SHI86] Shigeo Shingo, *Zero Quality Control: Source Inspection and the Poka-Yoke System*, Productivity Press, 1986.
- [SOM96] Somerville, I., *Software Engineering*, 5.<sup>a</sup> ed., Addison-Wesley, 1996.
- [TIN96] Tingey, M., *Comparing ISO 9000*, Malcolm Baldrige y el SEJ CMM para Software, Prentice-Hall, 1996.
- [TRI97] Tricker, R., *ISO 9000 for Small Businesses*, Butterworth-Heinemann, 1997.
- [VES81] Vesely, W.E., et al., *Fault Tree Handbook*, U.S Nuclear Regulatory Commission, NUREG-0492, Enero 1981.
- [WI494] Wilson, L. A., *8 steps to Successful ISO 9000*, Cambridge Interactive Publications, 1996.

## PROBLEMAS Y PUNTOS A CONSIDERAR

**8.1.** Al principio del capítulo se señaló que «el control de variación está en el centro del control de calidad». Como todos los programas que se crean son diferentes unos de otros, ¿cuáles son las variaciones que se buscan y cómo se controlan?

**8.2.** ¿Es posible evaluar la calidad del software si el cliente no se pone de acuerdo sobre lo que se supone que ha de hacer?

**8.3.** La calidad y la fiabilidad son conceptos relacionados, pero son fundamentalmente diferentes en varias formas. Discútalas.

**8.4.** ¿Puede un programa ser correcto y aún así no ser fiable? Explique por qué.

**8.5.** ¿Puede un programa ser correcto y aun así no exhibir una buena calidad? Explique por qué.

**8.6.** ¿Por qué a menudo existen fricciones entre un grupo de ingeniería del software y un grupo independiente de garantía de calidad del software? ¿Es esto provechoso?

**8.7.** Si se le da la responsabilidad de mejorar la calidad del software en su organización. ¿Qué es lo primero que haría? ¿Qué sería lo siguiente?

**8.8.** Además de los errores, ¿hay otras características claras del software que impliquen calidad? ¿Cuáles son y cómo se pueden medir directamente?

**8.9.** Una revisión técnica formal sólo es efectiva si todo el mundo se la prepara por adelantado. ¿Cómo descubriría que uno de los participantes no se la ha preparado? ¿Qué haría si fuera el jefe de revisión?

**8.10.** Algunas personas piensan que una RTF debe evaluar el estilo de programación al igual que la corrección. ¿Es una buena idea? ¿Por qué?

**8.11.** Revise la Tabla 8.1 y seleccione las cuatro causas vitales de errores serios y moderados. Sugiera acciones correctoras basándose en la información presentada en otros capítulos.

**8.12.** Una organización utiliza un proceso de ingeniería del software de cinco pasos en el cual los errores se encuentran de acuerdo a la siguiente distribución de porcentajes:

| Paso | Porcentaje de errores encontrados |
|------|-----------------------------------|
| 1    | 20 %                              |
| 2    | 15 %                              |
| 3    | 15 %                              |
| 4    | 40 %                              |
| 5    | 10%                               |

Mediante la información de la Tabla 8.1 y la distribución de porcentajes anterior, calcule el índice de defectos global para dicha organización. Suponga que PS = 100.000.

**8.13.** Investigue en los libros sobre fiabilidad del software y escriba un artículo que explique un modelo de fiabilidad del software. Asegúrese de incluir un ejemplo.

**8.14.** El concepto de TMEF del software sigue abierto a debate. ¿Puede pensar en algunas razones para ello?

**8.15.** Considere dos sistemas de seguridad crítica que estén controlados por una computadora. Liste al menos tres peligros para cada uno de ellos que se puedan relacionar directamente con los fallos del software.

**8.16.** Utilizando recursos web y de impresión, desarrolle un tutorial de 20 minutos sobre *poka-yoke* y expóngaselo a su clase.

**8.17.** Sugiera unos cuantos dispositivos *poka-yoke* que pueden ser usados para detectar y/o prevenir errores que son encontrados habitualmente antes de «enviar» un mensaje por e-mail.

**8.18.** Adquiera una copia de ISO 9001 e ISO 9000-3. Prepare una presentación que trate tres requisitos ISO 9001 y cómo se aplican en el contexto del software.

## OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Los libros de Moriguchi (*Software Excellence: A Total Quality Management Guide*, Productivity Press, 1997), Horch (*Practical Guide to Software Quality Management*, Artech Publishing, 1996) son unas excelentes presentaciones a nivel de gestión sobre los beneficios de los programas formales de garantía de calidad para el software de computadora. Los libros de Deming [DEM86] y Crosby [CRO79], aunque no se centran en el software, ambos libros son de lectura obligada para los gestores senior con responsabilidades en el desarrollo de software. Gluckman y Roome (*Everyday Heroes of the Quality Movement*, Dorset House, 1993) humaniza los aspectos de calidad contando la historia de los participantes en el proceso de calidad. Kan (*Metrics and Models in Software Quality Engineering*, Addison-Wesley, 1995) presenta una visión cuantitativa de la calidad del software. Manns (*Software Quality Assurance*, Macmillan, 1996) es una excelente introducción a la garantía de calidad del software.

Tingley (*Comparing ISO 9000, Malcolm Baldrige, and the SEI CMM for Software*, Prentice-Hall, 1996) proporciona una guía útil para las organizaciones que se esfuerzan en mejorar sus procesos de gestión de calidad. Oskarsson (*An ISO 9000 Approach to Building Quality Software*, Prentice-Hall, 1995) estudia cómo se aplica el estándar ISO al software.

Durante los últimos años se han escrito docenas de libros sobre aspectos de garantía de calidad. La lista siguiente es una pequeña muestra de fuentes útiles:

Clapp, J. A., et al., *Software Quality Control, Error Analysis and Testing*, Noyes Data Corp.. Park Ridge, NJ, 1995.

Dunn, R. H., y R.S. Ullman, *TQM for Computer Software*, McGrawHill, 1994.

Fenton, N., R. Whitty y Y. Iizuka, *Software Quality Assurance and Measurement: Worldwide Industrial Applications*, Chapman & Hall, 1994.

Ferdinand, A. E., *Systems, Software, and Quality Engineering*, Van Nostrand Reinhold, 1993.

Ginac, F. P., *Customer Oriented Software Quality Assurance*, Prentice-Hall, 1998.

Ince, D., *ISO 9001 and Software Quality Assurance*, McGraw-Hill, 1994.

Ince, D., *An Introduction to Software Quality Assurance and Implementation*, McGraw-Hill, 1994.

Jarvis, A., y V. Crandall, *Inroads to Software Quality: «How To» Guide and Toolkit*, Prentice-Hall, 1997.

Sanders, J., *Software Quality: A Framework for Success in Software Development*, Addison-Wesley, 1994.

Sumner, F. H. *Software Quality Assurance*, MacMillan, 1993.

Wallmuller, E., *Software Quality Assurance: A Practical Approach*, Prentice-Hall, 1995.

Weinberg, G. M., *Quality Software Management*, 4 vols., Dorset House, 1992, 1993, 1994 y 1996.

Wilson, R. C., *Software Rx: Secrets of Engineering Quality Software*, Prentice-Hall, 1997.

Una antología editada por Wheeler, Bryczynsky y Messon (*Software Inspection: Industry Best Practice*, IEEE Computer Society Press, 1996) presenta información útil sobre esta importante actividad de GCS. Friedman y Voas (*Software Assessment*, Wiley, 1995) estudian los soportes y métodos prácticos para asegurar la fiabilidad y la seguridad de programas de computadora.

Musa (*Software Reliability Engineering: More Reliable Software, Faster Development and Testing*, McGraw-Hill, 1998) ha escrito una guía práctica para aplicar a las técnicas de fiabilidad del software. Han sido editadas antologías de artículos importantes sobre la fiabilidad del software por Kapur y otros (*Contributions to Hardware and Software Reliability Modelling*, World Scientific Publishing Co., 1999), Gritzails (*Reliability, Quality and Safety of Software-Intensive Systems*, Kluwer Academic Publishing, 1997), y Lyu (*Handbook of Software Reliability Engineering*, McGraw-Hill, 1996). Storey (*Safety-Critical Computer Systems*, Addison-Wesley, 1996) y Leveson [LEV95] continúan siendo los estudios más completos sobre la seguridad del software publicados hasta la fecha.

Además de [SHI86], la técnica *poka-yoke* para el software de prueba de errores es estudiada por Shingo (*The Shingo Production Management System: Improving Product Quality by Preventing Defects*, Productivity Press, 1989) y por Shimbun (*Poka-Yoke: Improving Product Quality by Preventing Defects*, Productivity Press, 1989).

En Internet están disponibles una gran variedad de fuentes de información sobre la garantía de calidad del software, fiabilidad del software y otros temas relacionados. Se puede encontrar una lista actualizada con referencias a sitios (páginas) web que son relevantes para la calidad del software en <http://www.pressman5.com>.

## CAPÍTULO

# 9

# GESTIÓN DE LA CONFIGURACIÓN DEL SOFTWARE (GCS/SCM)\*

**C**UANDO se construye software de computadora, los cambios son inevitables. Además, los cambios aumentan el grado de confusión entre los ingenieros del software que están trabajando en el proyecto. La confusión surge cuando no se han analizado los cambios antes de realizarlos, no se han registrado antes de implementarlos, no se les han comunicado a aquellas personas que necesitan saberlo o no se han controlado de manera que mejoren la calidad y reduzcan los errores. Babich [BAB86] se refiere a este asunto cuando dice:

El arte de coordinar el desarrollo de software para minimizar...la confusión, se denomina gestión de configuración. La gestión de configuración es el arte de identificar, organizar y controlar las modificaciones que sufre el software que construye un equipo de programación. La meta es maximizar la productividad minimizando los errores.

La gestión de configuración del software (GCS) es una actividad de autoprotección que *se aplica* durante el proceso del software. Como el cambio se puede producir en cualquier momento, las actividades de GCS sirven para (1) identificar el cambio, (2) controlar el cambio, (3) garantizar que el cambio se implementa adecuadamente y (4) informar del cambio a todos aquellos que puedan estar interesados.

Es importante distinguir claramente entre el mantenimiento del software y la gestión de configuración del software. El mantenimiento es un conjunto de actividades de ingeniería del software que se producen después de que el software se haya entregado al cliente y esté en funcionamiento. La gestión de configuración del software es un conjunto de actividades de seguimiento y control que comienzan cuando se inicia el proyecto de ingeniería del software y termina sólo cuando el software queda fuera de la circulación.

## VISTAZO RÁPIDO

**¿Qué es?** Cuando construimos software de computadora, surgen cambios. Debido a esto, necesitamos controlarlos eficazmente. La gestión de la configuración del software (GCS) es un conjunto de actividades diseñadas para controlar el cambio identificando los productos del trabajo que probablemente cambien, estableciendo relaciones entre ellos, definiendo mecanismos para gestionar distintas versiones de estos productos, controlando los cambios realizados, y auditando e informando de los cambios realizados.

**¿Quién lo hace?** Todos aquellos que están involucrados en el proceso de ingeniería de software están relacionados con la GCS hasta cierto punto, pero las posiciones de mantenimiento especializadas **son** creadas a veces para la gestión del proceso de GCS.

**¿Por qué es importante?** Si no controlamos el cambio, él nos controlará a nosotros. Y esto nunca es bueno. Es muy fácil para un flujo de cambios incontrolados llevar al caos a un proyecto de software correcto. Por esta razón, la GCS es una parte esencial de una buena gestión del proyecto y una práctica formal de la ingeniería del software.

**¿Cuáles son los pasos?** Puesto que muchos productos de trabajo se obtienen cuando se construye el software, cada uno debe estar identificado únicamente. Una vez que esto se ha logrado, se pueden establecer los mecanismos para el control del cambio y de las versiones. Para garantizar que se mantiene la calidad mientras se realizan los cambios, se audita el proceso, y para asegurar que aquellos

que necesitan conocer los cambios son informados, se realizan los informes.

**¿Cuál es el producto obtenido?** El Plan de Gestión de la Configuración del Software define la estrategia del proyecto para la GCS. Además, cuando se realiza la GCS formal, el **proceso de control del cambio** provoca peticiones de cambio del software e informes de órdenes de cambios de ingeniería.

**¿Cómo puedo estar seguro de que lo he hecho correctamente?** Cuando cualquier producto de trabajo puede ser estimado para ser supervisado y controlado: cuando cualquier cambio pueda ser seguido y analizado; cuando cualquiera que necesite saber algo sobre algún cambio ha sido informado, lo habremos realizado correctamente.

\* En inglés, Software Configuration Management.

## 9.1 GESTIÓN DE LA CONFIGURACIÓN DEL SOFTWARE

El resultado del proceso de ingeniería del software es una información que se puede dividir en tres amplias categorías: (1) programas de computadora (tanto en forma de código fuente como ejecutable), (2) documentos que describen los programas de computadora (tanto técnicos como de usuario) y (3) datos ( contenidos en el programa o externos a él). Los elementos que componen toda la información producida como parte del proceso de ingeniería del software se denominan colectivamente configuración del software.

A medida que progresá el proceso del software, el número de *elementos de configuración del software* (*ECSs*) crece rápidamente. Una especificación del sistema produce un plan del proyecto del software y una especificación de requisitos del software (así como otros documentos relativos al hardware). A su vez, éstos producen otros documentos para crear una jerarquía de información. Si simplemente cada ECS produjera otros ECSs, no habría prácticamente confusión. Desgraciadamente, en el proceso entra en juego otra variable —el cambio—. El cambio se puede producir en cualquier momento y por cualquier razón. De hecho, la Primera Ley de la Ingeniería de Sistemas [BER80] establece: Sin importar en qué momento del ciclo de vida del sistema nos encontremos, el sistema cambiará y el deseo de cambiarlo persistirá a lo largo de todo el ciclo de vida.



No hay nada permanente excepto el cambio  
Heráclito, 500 AdC.

¿Cuál es el origen de estos cambios? La respuesta a esta pregunta es tan variada como los cambios mismos. Sin embargo, hay cuatro fuentes fundamentales de cambios:

- nuevos negocios o condiciones comerciales que dictan los cambios en los requisitos del producto o en las normas comerciales;
- nuevas necesidades del cliente que demandan la modificación de los datos producidos por sistemas de información, funcionalidades entregadas por productos o servicios entregados por un sistema basado en computadora;
- reorganización o crecimiento/reducción del negocio que provoca cambios en las prioridades del proyecto o en la estructura del equipo de ingeniería del software;
- restricciones presupuestarias o de planificación que provocan una redefinición del sistema o producto.

La gestión de configuración del software (GCS) es un conjunto de actividades desarrolladas para gestionar los cambios a lo largo del ciclo de vida del software de computadora.



*La mayoría de los cambios son justificados. No lamente las cambios. Mejor dicho, esté seguro que tiene los mecanismos preparados para realizarlos.*

### 9.1.1. Líneas base

Una línea base es un concepto de gestión de configuración del software que nos ayuda a controlar los cambios sin impedir seriamente los cambios justificados. La IEEE (Estándar IEEE 610.12-1990) define una línea base como:

Una especificación o producto que se ha revisado formalmente y sobre los que se ha llegado a un acuerdo, y que de ahí en adelante sirve como base para un desarrollo posterior que puede cambiarse solamente a través de procedimientos formales de control de cambios.

Una forma de describir la línea base es mediante una analogía:

Considere las puertas de la cocina en un gran restaurante. Para evitar colisiones, una puerta está marcada como SALIDA y la otra como ENTRADA. Las puertas tienen topes que hacen que sólo se puedan abrir en la dirección apropiada.

Si un camarero recoge un pedido en la cocina, lo coloca en una bandeja luego se da cuenta de que ha cogido un plato equivocado, puede cambiar el plato correcto rápidamente e informalmente antes de salir de la cocina.

Sin embargo, si abandona la cocina, le da el plato al cliente y luego se le informa de su error, debe seguir el siguiente procedimiento: (1) mirar en la orden de pedido si ha habido algún error; (2) disculparse insistente; (3) volver a la cocina por la puerta de ENTRADA; (4) explicar el problema, etc.



Un producto de trabajo de la ingeniería del software se convierte en una línea base, solamente después de haber sido revisado y aprobado.

Una línea base es análoga a la cocina de un restaurante. Antes de que un elemento de configuración de software se convierta en una línea base, el cambio se puede llevar a cabo rápida e informalmente. Sin embargo, una vez que se establece una línea base, pasamos, de forma figurada, por una puerta de un solo sentido. Se pueden llevar a cabo los cambios, pero se debe aplicar un procedimiento formal para evaluar y verificar cada cambio.

En el contexto de la ingeniería del software, definimos una *línea base* como un punto de referencia en el desarrollo del software que queda marcado por el envío de uno o más elementos de configuración del software y la aprobación del ECS obtenido mediante una revisión técnica formal (Capítulo 8). Por ejemplo, los ele-

mentos de una *Especificación de Diseño* se documentan y se revisan. Se encuentran errores y se corrigen. Cuando todas las partes de la especificación se han revisado, corregido y aprobado, la *Especificación de Diseño* se convierte en una línea base. Sólo se pueden realizar cambios futuros en la arquitectura del software (documentado en la *Especificación de Diseño*) tras haber sido evaluados y aprobados. Aunque se pueden definir las líneas base con cualquier nivel de detalle, las líneas base más comunes son las que se muestran en la Figura 9.1.



Asegúrese que la base de datos del proyecto se mantiene sobre un entorno controlado, centralizado.

La progresión de acontecimientos que conducen a un línea base está también ilustrada en la Figura 9.1. Las tareas de la ingeniería del software producen uno o más ECSs. Una vez que un ECS se ha revisado y aprobado, se coloca en una *base de datos del proyecto* (también denominada *biblioteca del proyecto* o *depósito de software*). Cuando un miembro del equipo de ingeniería del software quiere hacer modificaciones en un ECS de línea base, se copia de la base de datos del proyecto a un área de trabajo privada del ingeniero. Sin embargo, este ECS extraído puede modificarse sólo si se siguen los controles GCS (tratados más adelante en este capítulo). Las flechas punteadas de la Figura 9.1 muestran el camino de modificación de una línea base ECS.

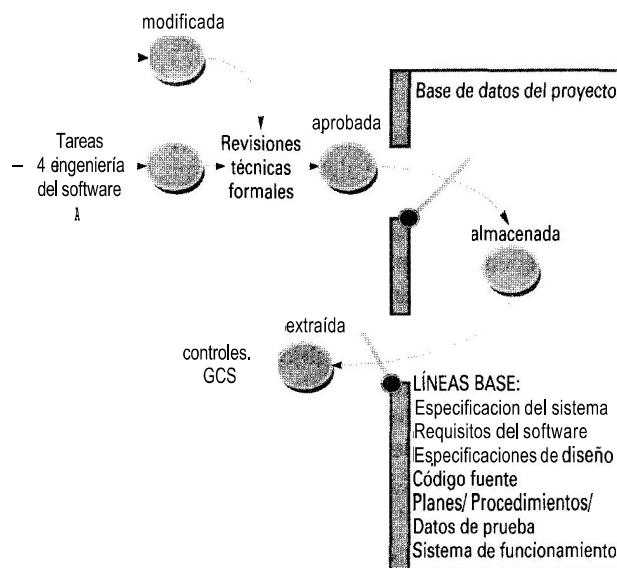


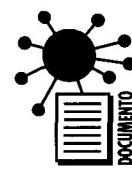
FIGURA 9.1. ECS de línea base y base de datos del proyecto.

### 9.1.2. Elementos de configuración del software

Ya hemos definido un elemento de configuración del software como la información creada como parte del proce-

so de ingeniería del software. Llevado al extremo, se puede considerar un ECS como una sección individual de una gran especificación o cada caso de prueba de un gran conjunto de pruebas. De forma más realista, un ECS es un documento, un conjunto completo de casos de prueba o un componente de un programa dado (p. ej., una función de C++ o un paquete de ADA).

En realidad, los ECSs se organizan como *objetos de configuración* que han de ser catalogados en la base de datos del proyecto con un nombre único. Un objeto de configuración tiene un nombre y unos atributos y está «conectado» a otros objetos mediante relaciones. De acuerdo con la Figura 9.2, los objetos de configuración, **Especificación de Diseño**, **modelo de datos**, **componente N**, **código fuente** y **Especificación de Prueba**, están definidos por separado. Sin embargo, cada objeto está relacionado con otros como muestran las flechas. Una flecha curvada representa una *relación de composición*. Es decir, **modelo de datos** y **componente N** son parte del objeto **Especificación de Diseño**. Una flecha recta con dos puntas representa una interrelación. Si se lleva a cabo un cambio sobre el objeto **código fuente**, las interrelaciones permiten al ingeniero de software determinar qué otros objetos (y ECSs) pueden verse afectados<sup>1</sup>.



Elementos de configuración del software.

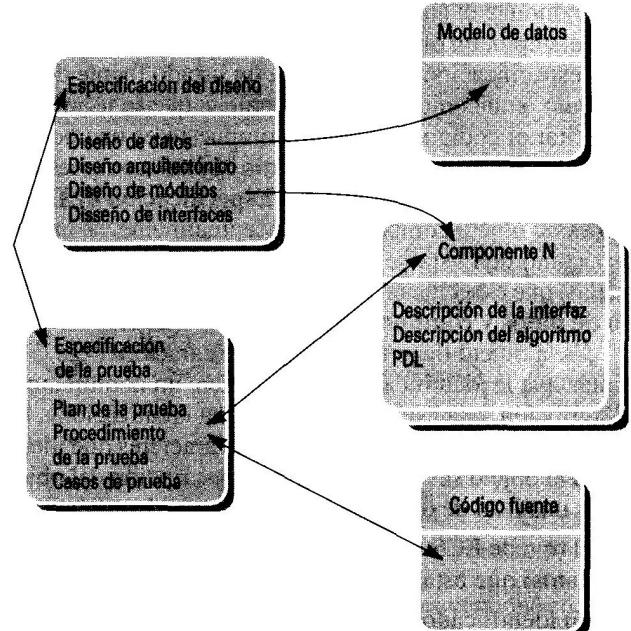


FIGURA 9.2. Objetos de configuración.

<sup>1</sup> Estas relaciones se tratan en la Sección 9.2.1 y la estructura de la base de datos del proyecto se trata con detalle en el Capítulo 31.

## 9.2 EL PROCESO DE GCS

La gestión de configuración del software es un elemento importante de garantía de calidad del software. Su responsabilidad principal es el control de cambios. Sin embargo, la GCS también es responsable de la identificación de ECSs individuales y de las distintas versiones del software, de las auditorías de la configuración del software para asegurar que se desarrollan adecuadamente y de la generación de informes sobre todos los cambios realizados en la configuración.



### Referencia Web

Los páginas amarillas de gestión de configuración contienen el listado más complejo de los recursos de GCS en la web. Para más información, consultar:  
[www.cs.colorado.edu/users/andre/configuration-management.html](http://www.cs.colorado.edu/users/andre/configuration-management.html)

Cualquier estudio de la GCS plantea una serie de preguntas complejas:

- ¿Cómo identifica y gestiona una organización las diferentes versiones existentes de un programa (y su documentación) de forma que se puedan introducir cambios eficientemente?
- ¿Cómo controla la organización los cambios antes y después de que el software sea distribuido al cliente?
- ¿Quién tiene la responsabilidad de aprobar y de asignar prioridades a los cambios?
- ¿Cómo podemos garantizar que los cambios se han llevado a cabo adecuadamente?
- ¿Qué mecanismo se usa para avisar a otros de los cambios realizados?

Estas cuestiones nos llevan a la definición de cinco tareas de GCS: *Identificación, control de versiones, control de cambios, auditorías de configuración y generación de informes*.

## 9.3 IDENTIFICACIÓN DE OBJETOS EN LA CONFIGURACIÓN DEL SOFTWARE

Para controlar y gestionar los elementos de configuración, se debe identificar cada uno de forma única y luego organizarlos mediante un enfoque orientado a objetos. Se pueden identificar dos tipos de objetos [CHO89]: *objetos básicos y objetos compuestos*<sup>2</sup>. Un objeto básico es una «unidad de texto» creada por un ingeniero de software durante el análisis, diseño, codificación o pruebas. Por ejemplo, un objeto básico podría ser una sección de una especificación de requisitos, un listado fuente de un módulo o un conjunto de casos prueba que se usan para ejercitarse el código. Un objeto compuesto es una colección de objetos básicos y de otros objetos compuestos. De acuerdo con la Figura 9.2, la **Especificación de Diseño** es un objeto compuesto. Conceptualmente, se puede ver como una lista de referencias con nombre (identificadas) que especifican objetos básicos, tales como **modelo de datos** y **componente N**.

Cada objeto tiene un conjunto de características distintas que lo identifican de forma única: un nombre, una descripción, una lista de recursos y una «realización». El nombre del objeto es una cadena de caracteres que identifica al objeto sin ambigüedad. La descripción del objeto es una lista de elementos de datos que identifican:

- el tipo de ECS (por ejemplo: documento, programa, datos) que está representado por el objeto;
- un identificador del proyecto;
- la información de la versión y/o el cambio;

### PUNTO CLAVE

Las relaciones establecidas entre los objetos de configuración permiten al ingeniero del software evaluar el impacto del cambio.

Los recursos son «entidades que proporciona, procesa, referencia o son, de alguna otra forma, requeridas por el objeto». [CHO89], por ejemplo, los tipos de datos, las funciones específicas e incluso los nombres de las variables pueden ser considerados recursos de objetos. La realización es una referencia a la «unidad de texto» para un objeto básico y nulo para un objeto compuesto.

La identificación del objeto de configuración también debe considerar las relaciones existentes entre los objetos identificados. Un objeto puede estar identificado como <parte-de> un objeto compuesto. La relación <parte-de> define una jerarquía de objetos. Por ejemplo, utilizando esta sencilla notación

Diagrama E-R 1.4 <parte-de> modelo de datos;  
 Modelo de datos <parte-de> especificación de diseño;

creamos una jerarquía de ECSs.

No es realista asumir que la única relación entre los objetos de la jerarquía de objetos se establece mediante largos caminos del árbol jerárquico. En muchos casos, los objetos están interrelacionados entre ramas de la

<sup>2</sup> Como mecanismos para representar una versión completa de la configuración del software se ha propuesto el concepto de objeto agregado [GUS89]

jerarquía de objetos. Por ejemplo, el modelo de datos está interrelacionado con los diagramas de flujo de datos (suponiendo que se usa el análisis estructurado) y también está interrelacionado con un conjunto de casos de prueba para una clase particular de equivalencia. Las relaciones a través de la estructura se pueden representar de la siguiente forma:

Modelo de datos <interrelacionado> modelo de flujo de datos;

Modelo de datos <interrelacionado> caso de prueba de la clase m;

#### Referencia cruzada

tos modelos de datos y los diagramas de flujo de datos se tratan en el Capítulo 12.

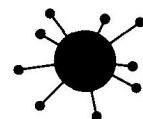
En el primer caso, la interrelación es entre objetos compuestos, mientras que en el segundo caso, la relación es entre un objeto compuesto (**modelo de datos**) y un objeto básico (**caso de prueba de la clase m**).

Las interrelaciones entre los objetos de configuración se pueden representar con un *lenguaje de interconexión de módulos (LIM)* [NAR87]. Un LIM describe las interdependencias entre los objetos de configuración y permite construir automáticamente cualquier versión de un sistema.

El esquema de identificación de los objetos de software debe tener en cuenta que los objetos evolucionan a lo largo del proceso de ingeniería del software. Antes de que un objeto se convierta en una línea base, puede cambiar varias veces, e incluso cuando ya es una línea base, los cambios se presentan con bastante frecuencia. Se puede crear un *grafo de evolución*

[GUS89] para cualquier objeto. El grafo de evolución describe la historia de los cambios de un objeto y aparece ilustrado en la Figura 9.3. El objeto de configuración 1.0 pasa la revisión y se convierte en el objeto 1.1. Algunas correcciones y cambios mínimos producen las versiones 1.1.1 y 1.1.2, que van seguidas de una actualización importante, resultando el objeto 1.2. La evolución de objeto 1.0 sigue hacia 1.3 y 1.4, pero, al mismo tiempo, una gran modificación del objeto produce un nuevo camino de evolución, la versión 2.0. A estas dos versiones se les sigue dando soporte.

Se pueden realizar cambios en cualquier versión, aunque no necesariamente en todas. ¿Cómo puede el desarrollador establecer las referencias de todos los componentes, documentos, casos de prueba de la versión 1.4? ¿Cómo puede saber el departamento comercial los nombres de los clientes que en la actualidad tienen la versión 2.1? ¿Cómo podemos estar seguros de que los cambios en el código fuente de la versión 2.1 han sido reflejados adecuadamente en la correspondiente documentación de diseño? Un elemento clave para responder a todas estas preguntas es la identificación.



Herramientas CASE-GCS

Se han desarrollado varias herramientas automáticas para ayudaren la tarea de identificación(y otras de GCS). En algunos casos, se diseña la herramienta para mantener copias completas de las versiones más recientes.

## 9.4 CONTROL DE VERSIONES

El *control de versiones* combina procedimientos y herramientas para gestionar las versiones de los objetos de configuración creados durante el proceso del software. Clemm [CLE89] describe el control de versiones en el contexto de la GCS:



Elesquema que Vd. estableció por los ECS debería incorporar el número de versión.

La gestión de configuración permite a un usuario especificar configuraciones alternativas del sistema de software mediante la selección de las versiones adecuadas. Esto se puede gestionar asociando atributos a cada versión del software y permitiendo luego especificar [y construir]una configuración describiendo el conjunto de atributos deseado.

Los «atributos» que se mencionan pueden ser tan sencillos como un número específico de versión asociado a cada objeto o tan complejos como una cadena de variables lógicas (indicadores) que especifiquen

tipos de cambios funcionales aplicados al sistema [LIE89].

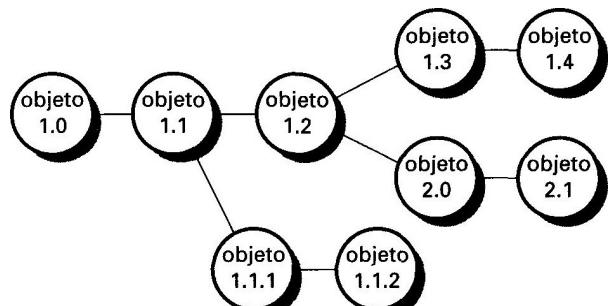
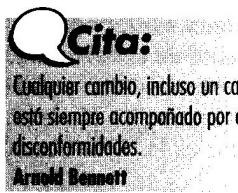


FIGURA 9.3. Grafo de evolución.

Una representación de las diferentes versiones de un sistema es el grafo de evolución mostrado en la Figura 9.3. Cada nodo del grafo es un objeto compuesto, es decir, una versión completa del software. Cada versión del software es una colección de ECSs (código fuente, documentos, datos) y cada versión puede estar com-

puesta de diferentes variantes. Para ilustrar este concepto, consideremos una versión de un sencillo programa que está formado por los componentes 1, 2, 3, 4 y 5<sup>3</sup>. El componente 4 sólo se usa cuando el software se implementa para monitores de color. El componente 5 se implementa cuando se dispone de monitor monocromo. Por tanto, se pueden definir dos variantes de la versión: (1) componentes 1, 2, 3 y 4; (2) componentes 1, 2, 3 y 5.

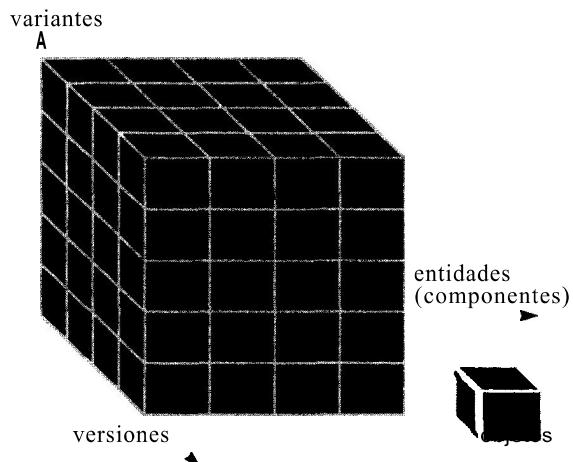
Para construir la *variante* adecuada de una determinada versión de un programa, a cada componente se le asigna una «tupla de atributos» —una lista de características que definen si se ha de utilizar el componente cuando se va a construir una determinada versión del software—. A cada variante se le asigna uno o más atributos. Por ejemplo, se podría usar un atributo color para definir qué componentes se deben incluir para soporte de monitores en color.



Otra forma de establecer los conceptos de la relación entre componentes, variantes y versiones (revisiones) es representarlas como un *fondo de objetos* [REI89]. De acuerdo con la Figura 9.4, la relación entre los objetos de configuración y los componentes, las variantes y las versiones se pueden representar como un espacio tridimensional. Un componente consta de una colección de

objetos del mismo nivel de revisión. Una variante es una colección diferente de objetos del mismo nivel de revisión y, por tanto, coexiste en paralelo con otras variantes. Una nueva versión se define cuando se realizan cambios significativos en uno o más objetos.

En la pasada década se propusieron diferentes enfoques automatizados para el control de versiones. La principal diferencia entre los distintos enfoques está en la sofisticación de los atributos que se usan para construir versiones y variantes específicas de un sistema y en la mecánica del proceso de construcción.



**FIGURA 9.4.** Representación en fondo de objetos de los componentes, variantes y versiones [REI89].

## 9.5 CONTROL DE CAMBIOS

La realidad del *control de cambio* en un contexto moderno de ingeniería de software ha sido bien resumida por James Bach [BAC98] :

El control de cambio es vital. Pero las fuerzas que lo hacen necesario también lo hacen molesto. Nos preocupamos por el cambio porque una diminuta perturbación en el código puede crear un gran fallo en el producto. Pero también puede reparar un gran fallo o habilitar excelentes capacidades nuevas. Nos preocupamos por el cambio porque un desarrollador pícaro puede hacer fracasar el proyecto; sin embargo las brillantes ideas nacidas en la mente de estos pícaros, y un pesado proceso de control de cambio pueden disuadirle de hacer un trabajo creativo.

Bach reconoce que nos enfrentamos a una situación a equilibrar. Mucho control de cambio y crearemos problemas. Poco, y crearemos otros problemas.

En un gran proyecto de ingeniería de software, el cambio incontrolado lleva rápidamente al caos. Para estos pro-



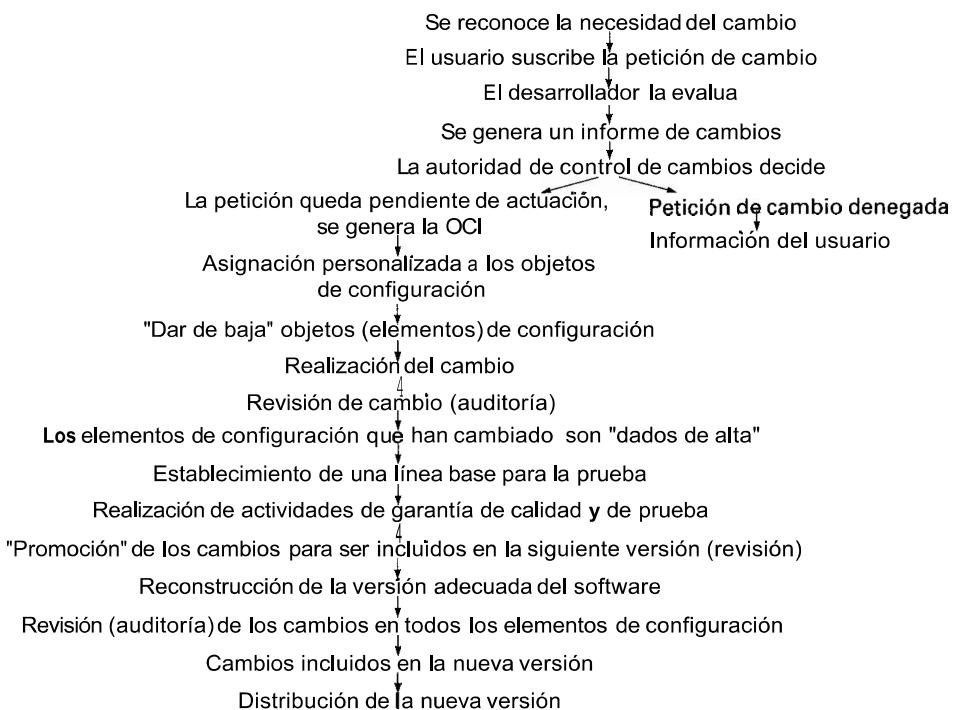
yectos, el control de cambios combina los procedimientos humanos y las herramientas automáticas para proporcionar un mecanismo para el control del cambio. El proceso de control de cambios está ilustrado esquemáticamente en la Figura 9.5. Se hace una petición de cambio<sup>4</sup> y se evalúa para calcular el esfuerzo técnico, los posibles efectos secundarios, el impacto global sobre otras funciones del sistema y sobre otros objetos de la configuración. Los resultados de la evaluación se presentan

<sup>3</sup> En este contexto, el término «componente» se refiere a todos los objetos compuestos y objetos básicos de un ESC de línea base. Por ejemplo, un componente de «entrada» puede estar constituido por seis componentes de software distintos, cada uno responsable de una subfunción de entrada.

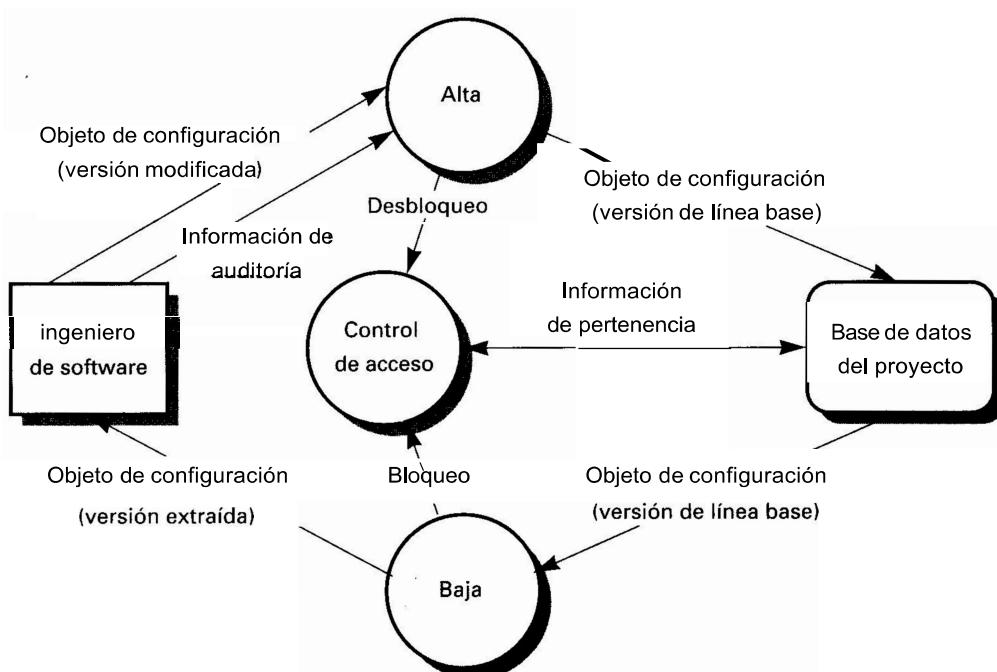
<sup>4</sup> Aunque muchas de las peticiones de cambio se reciben durante la fase de mantenimiento, en este estudio tomamos un punto de vista más amplio. Una petición de cambio puede aparecer en cualquier momento durante el proceso del software.

como un informe de cambios a la *autoridad de control de cambios (ACC)* —una persona o grupo que toma la decisión final del estado y la prioridad del cambio—. Para cada cambio aprobado se genera una *orden de cambio de ingeniería (OCI)*. La OCI describe el cambio a realizar, las restricciones que se deben respetar y los criterios de revisión

y de auditoría. El objeto a cambiar es «dato de baja» de la base de datos del proyecto; se realiza el cambio y se aplican las adecuadas actividades de **SQA**. Luego, el objeto es «dato de alta» en la base de datos y se usan los mecanismos de control de versiones apropiados (Sección 9.4) para crear la siguiente versión del software.



**FIGURA 9.5.** El proceso de control de cambios.



**FIGURA 9.6.** Control de acceso y de sincronización.



*La confusión conduce a los errores -algunas de ellas muy serias—. Los controles de acceso y de sincronización evitan la confusión. Implemente ambos, incluso si su enfoque tiene que ser simplificado para adaptarlo a su cultura de desarrollo.*

Los procesos de «alta» y «baja» implementan dos elementos importantes del control de cambios —control de acceso y control de sincronización—. El *control de acceso* gobierna los derechos de los ingenieros de software a acceder y modificar objetos de configuración concretos. El *control de sincronización* asegura que los cambios en paralelo, realizados por personas diferentes, no se sobreescreiben mutuamente [HAR89].

El flujo de control de acceso y de sincronización están esquemáticamente ilustrados en la Figura 9.6. De acuerdo con una petición de cambio aprobada y una OCI, un ingeniero de software *da de baja* a un objeto de configuración. Una función de control de acceso comprueba que el ingeniero tiene autoridad para dar de baja el objeto, y el control de sincronización *bloquea* el objeto en la base de datos del proyecto, de forma que no se puedan hacer más actualizaciones hasta que se haya reemplazado con la nueva versión. Fíjese que se pueden dar de baja otras copias, pero no se podrán hacer otras actualizaciones. El ingeniero de software modifica una copia del objeto de línea base, denominada *versión extraída*. Tras la SQA y la prueba apropiada, se *da de alta* la versión modificada del objeto y se desbloquea el nuevo objeto de línea base.

Puede que algunos lectores empiecen a sentirse incómodos con el nivel de burocracia que implica el proceso anterior. Esta sensación es normal. Sin la protección adecuada, el control de cambios puede ralentizar el progreso y crear un papeleo innecesario. La mayoría de los desarrolladores de software que disponen de mecanismos de control de cambios (desgraciadamente la mayoría no tienen ninguno) han creado varios niveles de control para evitar los problemas mencionados anteriormente.



*Opte por un poco más de control de cambio del que pensaba necesitar en un principio. Es probable que mucha pueda ser la cantidad apropiada.*

Antes de que un ECS se convierta en una línea base, sólo es necesario aplicar *un control de cambios informal*. El que haya desarrollado el ECS en cuestión podrá hacer cualquier cambio justificado por el proyecto y por los requisitos técnicos (siempre que los cambios no impacten en otros requisitos del sistema más amplios que queden fuera del ámbito de trabajo del encargado del desarrollo). Una vez que el objeto ha pasado la revisión técnica formal y ha sido aprobado, se crea la línea base. Una vez que el ECS se convierte en una línea base, aparece el *control de cambios a nivel de proyecto*. Ahora, para hacer un cambio, el encargado del desarrollo debe recibir la aprobación del gestor del proyecto (si el cambio es «local») o de la ACC si el cambio impacta en otros ECSs. En algunos casos, se dispensa de generar formalmente las peticiones de cambio, los informes de cambio y las OCI. Sin embargo, hay que hacer una evaluación de cada cambio así como un seguimiento y revisión de los mismos.

Cuando se distribuye el producto de software a los clientes, se instituye el *control de cambios formal*. El procedimiento de control de cambios formal es el que aparece definido en la Figura 9.5.

La autoridad de control de cambios (ACC) desempeña un papel activo en el segundo y tercer nivel de control. Dependiendo del tamaño y de las características del proyecto de software, la ACC puede estar compuesta por una persona —el gestor del proyecto— o por varias personas (por ejemplo, representantes del software, del hardware, de la ingeniería de las bases de datos, del soporte o del departamento comercial, etc.). El papel de la ACC es el de tener una visión general, o sea, evaluar el impacto del cambio fuera del ECS en cuestión. ¿Cómo impactará el cambio en el hardware? ¿Cómo impactará en el rendimiento? ¿Cómo alterará el cambio la percepción del cliente sobre el producto? ¿Cómo afectará el cambio a la calidad y a la fiabilidad? La ACC se plantea estas y otras muchas cuestiones.



*El cambio es inevitable, excepto para las máquinas de venta.*

*Bumper Sticker*

## 9.6 AUDITORÍA DE LA CONFIGURACIÓN

La identificación, el control de versiones y el control de cambios ayudan al equipo de desarrollo de software a mantener un orden que, de otro modo, llevaría a una situación caótica y sin salida. Sin embargo, incluso los mecanismos más correctos de control de cambios hacen un seguimiento al cambio sólo hasta que se ha generado la OCI. ¿Cómo podemos asegurar que el cambio se

ha implementado correctamente? La respuesta es doble: (1) revisiones técnicas formales y (2) auditorías de configuración del software.

Las revisiones técnicas formales (presentadas en detalle en el Capítulo 8) se centran en la corrección técnica del elemento de configuración que ha sido modificado. Los revisores evalúan el ECS para determinar la con-

sistencia con otros ECSs, las omisiones o los posibles efectos secundarios. Se debe llevar a cabo una revisión técnica formal para cualquier cambio que no sea trivial.

Una *auditoría de configuración del software* complementa la revisión técnica formal al comprobar características que generalmente no tiene en cuenta la revisión. La auditoría se plantea y responde las siguientes preguntas:



**¿Cuáles son las principales preguntas que hacemos en una auditoría de configuración?**

1. ¿Se ha hecho el cambio especificado en la OCI? ¿Se han incorporado modificaciones adicionales?
2. ¿Se ha llevado a cabo una revisión técnica formal para evaluar la corrección técnica?

3. ¿Se ha seguido el proceso del software y se han aplicado adecuadamente los estándares de ingeniería del software?
4. ¿Se han «resaltado» los cambios en el ECS? ¿Se han especificado la fecha del cambio y el autor? ¿Reflejan los cambios los atributos del objeto de Configuración?
5. ¿Se han seguido procedimientos de GCS para señalar el cambio, registrarlo y divulgarlo?
6. ¿Se han actualizado adecuadamente todos los ECSs relacionados?

En algunos casos, las preguntas de auditoría se incluyen en la revisión técnica formal. Sin embargo, cuando la GCS es una actividad formal, la auditoría de la GCS se lleva a cabo independientemente por el grupo de garantía de calidad.

## 9.7 INFORMES DE ESTADO

La generación de *informes de estado de la configuración* (a veces denominada *contabilidad de estado*) es una tarea de GCS que responde a las siguientes preguntas: (1) ¿Qué pasó? (2) ¿Quién lo hizo? (3) ¿Cuándo pasó? (4) ¿Qué más se vio afectado?

En la Figura 9.5 se ilustra el flujo de información para la generación de *informes de estado de la configuración (IEC)*. Cada vez que se asigna una nueva identificación a un ECS o una identificación actualizada se genera una entrada en el IEC. Cada vez que la ACC aprueba un cambio (o sea, se expide una OCI), se genera una entrada en el IEC. Cada vez que se lleva a cabo una auditoría de configuración, los resultados aparecen como parte de una tarea de generación de un IEC. La salida del IEC se puede situar en una base de datos interactiva [TAY85] de forma que los encargados del desarrollo o del mantenimiento del software puedan acceder a la información de cambios por categorías clave. Además, se genera un IEC regularmente con intención de mantener a los gestores y a los profesionales al tanto de los cambios importantes.



*Desarrolle una «lista que se necesita conocer» para cada ECS y manténgala actualizada. Cuando se realice un cambio, asegúrese que se informa a todos los que están en la lista.*

La generación de informes de estado de la configuración desempeña un papel vital en el éxito del proyecto de desarrollo de software. Cuando aparece involucrada mucha gente es muy fácil que se dé el síndrome de que «la mano izquierda ignora lo que hace la mano derecha». Puede que dos programadores intenten modificar el mismo ECS con intenciones diferentes y conflictivas. Un equipo de ingeniería del software puede emplear meses de esfuerzo en construir un software a partir de unas especificaciones de hardware obsoletas. Puede que la persona que descubra los efectos secundarios seños de un cambio propuesto no esté enterada de que el cambio se está realizando. El IEC ayuda a eliminar esos problemas, mejorando la comunicación entre todas las personas involucradas.

## RESUMEN

La gestión de configuración del software es una actividad de protección que se aplica a lo largo de todo el proceso del software. La GCS identifica, controla, audita e informa de las modificaciones que invariablemente se dan al desarrollar el software una vez que ha sido distribuido a los clientes. Cualquier información producida como parte de la ingeniería del software se convierte en parte de una configuración del software. La configuración se organiza de tal forma que sea posible un control organizado de los cambios.

La configuración del software está compuesta por un conjunto de objetos interrelacionados, también denominados elementos de configuración del software, que se producen como resultado de alguna actividad de ingeniería del software. Además de los documentos, los programas y los datos, también se puede poner bajo control de configuración el entorno de desarrollo utilizado para crear el software.

Una vez que se ha desarrollado y revisado un objeto de configuración, se convierte en una línea base. Los

cambios sobre un objeto de línea base conducen a la creación de una nueva versión del objeto. La evolución de un programa se puede seguir examinando el histórico de las revisiones de todos los objetos de su configuración. Los objetos básicos y los compuestos forman un asociación de objetos que refleja las variantes y las versiones creadas. El control de versiones es un conjunto de procedimientos y herramientas que se usan para gestionar el uso de los objetos.

El control de cambios es una actividad procedimental que asegura la calidad y la consistencia a medi-

da que se realizan cambios en los objetos de la configuración. El proceso de control de cambios comienza con una petición de cambio, lleva a una decisión de proseguir o no con el cambio y culmina con una actualización controlada del ECS que se ha de cambiar.

La auditoría de configuración es una actividad de SQA que ayuda a asegurar que se mantiene la calidad durante la realización de los cambios. Los informes de estado proporcionan información sobre cada cambio a aquellos que tienen que estar informados.

## REFERENCIAS

- [ADA89] Adams, E., M. Honda y T. Miller, «Object Management in a CASE Environment», *Proc. 11 th Intl. Conf Software Engineering*, IEEE, Pittsburg, PA, Mayo 1989, pp. 154-163.
- [BAC98] Bach, J., «The Highs and Lows of Change Control», *Computer*, vol. 31, n.º 8, Agosto 1998, pp. 113-115.
- [BER80] Bersoff, E.H., V.D. Henderson y S. G. Siegel, *Software Configuration Management*, Prentice-Hall, 1980.
- [BRY80] Bryan, W., C. Chadboume, y S. Siegel, *Software Configuration Management*, IEEE Computer Society Press, 1980.
- [CHO89] Choi, S. C., y W. Scacchi, «Assuring the Correctness of a Configured Software Description», *Proc. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, Octubre 1989, pp. 66-75.
- [CLE89] Clemm, G. M., «Replacing Version Control with Job Control», *Proc. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, Octubre 1989, pp. 162-169.
- [GUS89] Gustavsson, A., «Maintaining the Evaluation of Software Objects in an Integrated Environment», *Proc. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, Octubre 1989, pp. 114-117.
- [HAR89] Harter, R., «Configuration Management», *HP Professional*, vol. 3, n.º 6, Junio 1989.
- [IEE94] *Software Engineering Standards*, edición de 1994, IEEE Computer Society, 1994.
- [LIE89] Lie, A. et al., «Change Oriented Versioning in a Software Engineering Database», *Proc. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, Octubre 1989, pp. 56-65
- [NAR87] Narayanaswamy, K., y W. Scacchi, «Maintaining Configurations of Evolving Software Systems», *IEEE Trans. Software Engineering*, vol. SE-13, n.º 3, Marzo 1987, pp. 324-334.
- [REI89] Reichenberger, C., «Orthogonal Version Management», *Proc. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, Octubre 1989, pp. 137-140.
- [ROC75] Rochkind, M., «The Source Code Control System», *IEEE Trans. Software Engineering*, vol. SE-1, n.º 4, Diciembre 1975, pp. 364-370.
- [TAY85] Taylor, B., «A Database Approach to Configuration Management for Large Projects», *Proc. Conf. Software Maintenance-1985*, IEEE, Noviembre 1985, pp. 15-23.
- [TIC82] Tichy, W. F., «Design, Implementation and Evaluation of a Revision Control System», *Proc. 6<sup>th</sup> Intl. Conf. Software Engineering*, IEEE, Tokyo, Septiembre 1982, pp. 58-67.

## PROBLEMAS Y PUNTOS A CONSIDERAR

**9.1.** ¿Por qué es cierta la primera ley de la ingeniería de sistemas? ¿Cómo afecta a nuestra percepción de los paradigmas de la ingeniería del software?

**9.2.** Exponga las razones de la existencia de líneas base con sus propias palabras.

**9.3.** Asuma que es el gestor de un pequeño proyecto. ¿Qué líneas base definiría para el proyecto y cómo las controlaría?

**9.4.** Diseñe un sistema de base de datos que permita a un ingeniero del software guardar, obtener referencias de forma cruzada, buscar, actualizar, cambiar, etc., todos los elementos de la configuración del software importantes. ¿Cómo manejaría la base de datos de diferentes versiones de un mis-

mo programa? ¿Se manejaría de forma diferente el código fuente que la documentación? ¿Cómo se evitaría que dos programadores hicieran cambios diferentes sobre el mismo ECS al mismo tiempo?

**9.5.** Investigue un poco sobre bases de datos orientadas a objetos y escriba un artículo que describa cómo se podrían usar en el contexto de la GCS.

**9.6.** Utilice un modelo E-R (Capítulo 12) para describir las interrelaciones entre los ECS (objetos) de la Sección 9.1.2.

**9.7.** Investigue sobre herramientas de GCS existentes y describa cómo implementan el control de versiones, de cambios

**9.8.** Las relaciones «parte-de» e «interrelacionado» representan relaciones sencillas entre los objetos de configuración. Describa cinco relaciones adicionales que pudieran ser útiles en el contexto de la base de datos del proyecto.

**9.9.** Investigue sobre una herramienta de GCS existente y describa cómo implementa los mecanismos de control de versiones. Alternativamente, lea dos o tres de los artículos a los que se hace referencia en este capítulo e investigue en las estructuras de datos y los mecanismos que se usan para el control de versiones.

**9.10.** Utilizando la Figura 9.5 como guía, desarrolle un esquema de trabajo más detallado aún para el control de cambios. Describa el papel de la ACC y sugiera formatos para la petición de cambio, el informe de cambios e IEC.

**9.11.** Desarrolle una lista de comprobaciones que se pueda utilizar en las auditorías de configuración.

**9.12.** ¿Cuál es la diferencia entre una auditoría de GCS y una revisión técnica formal? ¿Se pueden juntar sus funciones en una sola revisión? ¿Cuáles son los pros y los contras?

## OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Uno de los pocos libros escritos sobre la GCS en los últimos años lo realizaron Brown et al. (*AntiPatterns and Patterns in Software Configuration Management*, Wiley, 1989).

Los autores tratan las cosas que no hay que hacer (antipatrones) cuando se implementa un proceso de GCS y entonces consideran sus remedios.

Lyon (*Practical CM: Best Configuration Management Practices for the 21<sup>st</sup> Century*, Raven Publishing, 1999) y Mikkelsen y Pherigo (*Practical Software Configuration Management: The Latenight Developer's Handbook*, Allyn & Bacon, 1997) proporcionan tutoriales prácticos importantes de GCS. Ben-Menachem (*Software Configuration Management Guidebook*, McGraw-Hill, 1994), Vacca (*Implementing a Successful Configuration Change Management Program*, I.S. Management Group, 1993), y Ayer y Patrinostro (*Software Configuration Management*, McGrawHill, 1992) presentan correctas visiones para aquellos que todavía no se han introducido en la materia. Berlack (*Software Configuration Management*, Wiley, 1992, presenta una estudio útil de conceptos de la GCS, haciendo incapié en la importancia del diccionario de datos (repository) y de las herramientas en la gestión del cambio. Babich [BAB86] es un tratado abreviado, aunque eficaz, de temas prácticos sobre la gestión de configuración del software.

Buckley (*Implementing Configuration Management*, IEEE Computer Society Press, 1993) estudia enfoques de gestión de configuración para todos los elementos de un sistema (hardware, software y firmware con unos detallados tratamientos

de las principales actividades de GC). Rawlings (*SCM for Network Development Environments*, McGraw-Hill, 1994) es el primer libro de GCS en tratar el asunto con un énfasis específico en el desarrollo de software en un entorno de red. Whitgift (*Methods and Tools for Software Configuration Management*, Wiley, 1991) contiene una razonable cobertura de todos los temas importantes de GCS, pero se distingue por su estudio del diccionario de datos y tratamiento de aspectos de entornos CASE. Arnold y Bohner (*Software Change Impact Analysis*, IEEE Computer Society Press, 1996) han editado una antología que estudia cómo analizar el impacto del cambio dentro de sistemas complejos basados en software.

Puesto que la GCS identifica y controla documentos de ingeniería del software, los libros de Nagle (*Handbook for Preparing Engineering Documents: From Concept to Completion*, IEEE, 1996), Watts (*Engineering Documentation Control Handbook: Configuration Management for Industry*, Noyes Publication, 1993). Ayer y Patinostro (*Documenting the Software Process*, McGraw-Hill, 1992) proporciona un complemento con textos más centrados en la GCS. La edición de Marzo de 1999 de *Crosstalk* contiene varios artículos útiles de la GCS.

En Internet están disponibles una gran variedad de fuentes de información relacionadas con temas de gestión y de software. Se puede encontrar una lista actualizada con referencias a sitios (páginas) web que son relevantes para el Software en <http://www.pressman5.com>.





## MÉTODOS CONVENCIONALES PARA LA INGENIERÍA DEL SOFTWARE

**E**N esta parte de *Ingeniería del Software Un Enfoque Práctico* consideramos los conceptos técnicos, métodos y mediciones que son aplicables al análisis, diseño y pruebas del software. En los capítulos siguientes, aprenderás las respuestas a las siguientes cuestiones:

- ¿Cómo se define el software dentro del contexto de un gran sistema y qué papel juega la ingeniería de sistemas?
- ¿Cuáles son los principios y conceptos básicos que se aplican al análisis de requisitos del software?
- ¿Qué es el análisis estructurado y cómo sus diferentes modelos te permiten entender las funciones, datos y comportamientos?
- ¿Cuáles son los conceptos básicos y los principios que se aplican en la actividad de diseño del software?
- ¿Cómo se realiza el diseño de los modelos de datos; arquitectura, interfaces y componentes?
- ¿Cuáles son los conceptos básicos, principios y estrategias que se aplican a las pruebas del software?
- ¿Cómo se utilizan los métodos de prueba de caja negra y caja blanca para diseñar eficientes casos de prueba?
- ¿Qué métricas técnicas están disponibles para valorar la calidad de los modelos de análisis y diseño, código fuente y casos de prueba?

Una vez que estas cuestiones sean contestadas, comprenderás cómo construir software utilizando un enfoque disciplinado de ingeniería.

