Rellena lo huecos

1 Introducción

Los programas en ejecución fallan, por ello es importante hacer [**pruebas**]. Las [**pruebas**] deben descubrir fallos fruto de errores: de programación, de instalación, de configuración, en el análisis de requisitos, en el diseño del programa… Los [**fallos**] detectados hay que solucionarlos de forma rápida y fácil. Para ello es importante que el programa esté bien [**documentado**], lo que facilita analizar y retocar el programa.

2 Calidad del software

Las aplicaciones desarrolladas deben tener como objetivo la [**calidad**]. La [**calidad**] vista por el [**usuario**] es la corrección, es decir, que el programa haga lo que se ha pedido. Desde el punto de vista del programador la [**calidad**] se logra si el programa es mantenible y ampliable.

3 Planificación de pruebas

El objetivo de las [**pruebas**] es la detección de fallos. Descubrir un fallo es el [**éxito**] de una prueba. Por desgracia la prueba exhaustiva del software es [**imposible**], es decir, no se pueden probar todas las posibilidades de su funcionamiento. Por ello hay que establecer un [**plan**] de pruebas. Este plan se debe establecer desde el principio, antes incluso de haber producido código alguno, porque cuanto antes se detecte un fallo menos [**costoso**] será corregir el error que lo produce. Vamos a ver, a continuación, que hay que probar y a después como se diseñan los casos a probar.

3.1 Qué probar

Según lo que se esté probando existen diferentes tipos de [**pruebas**]:

1. Funcionales: Verifican si la salida es apropiada.
2. De seguridad: Comprueban la vulnerabilidad ante [**amenazas**].
3. De recuperación: Verifican la capacidad del sistema de recuperarse ante [**fallos**].
4. De rendimiento: Comprueban la adecuación de los tiempos de respuesta.
5. De carga: Verifican si se soporta el volumen de trabajo esperado.
6. De compatibilidad: Determinan si todo [**funciona**] correctamente en el entorno de operación esperado.
7. De escalabilidad: Determinan cuales son los niveles de crecimiento que puede soportar la [**aplicacion**].

3.1.1 Diseño de casos de pruebas

Se llama caso de prueba a cada [**prueba**] particular. Un caso de prueba se compone de un conjunto de entradas, las condiciones de ejecución y el resultado esperado. Como es [**imposible**] probar el 100% de los casos hay que buscar un compromiso entre el número de [**casos**] de prueba diseñados y la probabilidad de que [**solucionen**] los fallos existentes. En la búsqueda de ese [**compromiso**] existen dos enfoques.

3.1.1.1 Pruebas de caja blanca

En las pruebas de caja blanca, también llamadas [**pruebas estructurales**], se diseñan los [**casos**] de [**prueba**] teniendo en cuenta la implementación de software a probar. Para lograr el [**compromiso**] entre el número de casos de prueba diseñados y la probabilidad de detectar los fallos existen varios criterios: cobertura de sentencias, cobertura de [**errores**] y cobertura de caminos. El número de caminos independientes en un fragmento de código, independientemente del lenguaje empleado, se obtiene mediante la técnica de [**McCabe**]. Este número, también conocido como complejidad [**ciclómatica**], determina la cota superior del número de pruebas que se deben [**diseñar**] para asegurar que cada sentencia se ejecutará al menos [**una**] vez. Para obtener el número de [**caminos**] representamos el código mediante un grafo de flujo de control y empleamos cualquiera de las fórmulas representadas en la figura.

3.1.1.2 Pruebas de caja negra

En el diseño de los [**casos**] de [**prueba**] sólo se tiene en cuenta lo que hace el programa, cómo si sólo se conociera el interfaz. Por ejemplo, si se prueba una clase el interfaz serán los métodos. Si se prueba una aplicación el interfaz será el entorno de usuario. Para lograr el [**compromisos**] entre casos de [**entrada**] mínimos y cubrimiento máximo agrupamos las posibles entradas mediante clases de equivalencia y también probamos los valores límite de dichas [**clases**]. Las [**clases**] de equivalencia se establecen a partir de la funcionalidad conocida de lo que se va a probar.

3.2 Cuando probar

Existen diferentes tipos de [**pruebas**] dependiendo del momento en el que se realicen.

3.2.1 Pruebas de unidad

Son las primeras comprobaciones de la [ ] y en ellas se prueban por separado los diferentes módulos o unidades que la forman. Normalmente se utiliza la técnica de [**caja**] blanca, aunque también se pueden diseñar los casos de [**prueba**] antes de la creación del módulo a partir de las especificaciones (caja [**negra**]). Para realizar estas [**pruebas**] hay que crear un programa de prueba que interactúe con el [**modulo**] en pruebas y, si es necesario, módulos inferiores virtuales. Es importante sistematizarlas y documentarlas por si se hacen versiones utilizando herramientas al efecto.

3.2.2 Pruebas de integración

Comprueban el ensamblaje de las [**unidades**] ya probadas. Hay tres tipos de integración.

1. Súbita: No adecuada. Con la [**integración**] súbita pueden salir problemas muy tarde.
2. Ascendente: Se asciende en la jerarquía de [**módulos**].
3. Descendente: Se prueba contra módulos virtuales. La [**integracion**] descendente es aconsejable desde el punto de vista del cliente porque desde el [**inicio**] de las pruebas conoce el [**estado**] de la aplicación.

3.2.3 Pruebas de regresión

Estas [**pruebas**] intentan descubrir las causas de nuevos [**fallos**] inducidos por cambios realizados. Por ello se realizan después de algún [**cambio**]. Lo ideal es repetir todas las [**pruebas**] ya realizadas debidamente

guardadas y documentadas. Es recomendable sistematizarlas mediante herramientas. Debido a los [**cambios**] podría ser necesario actualizar las pruebas.

3.2.4 Pruebas de sistema

Se realizan después de las [**pruebas**] de integración. Se hacen pruebas de carga, rendimiento, seguridad y recuperación. Si como resultado de estas pruebas hay que hacer cambios, después de realizarlos, se pasan pruebas de [**regresión**].

3.2.5 Pruebas de validación

Normalmente las realiza el cliente y se utiliza la técnica de caja [**negra**].

4 Control de versiones

Se llama control de versiones a la gestión de los diversos [ ] que se realizan en el sistema. Los sistemas de [ ] de versiones facilitan la administración de las distintas [ ]. Aunque un sistema de control de versiones puede realizarse de forma manual, es muy aconsejable disponer de [ ] que faciliten esta gestión como CVS, Subversion, SourceSafe, ClearCase, Darcs, Bazaar, Plastic SCM, Git, Mercurial…

4.1 Características

Un sistema de control de [ ] debe proporcionar:

1. Mecanismo de [ ] del código y documentación que deba gestionar.
2. Posibilidad de realizar [ ] sobre los elementos almacenados.
3. Registro histórico de las acciones realizadas con cada elemento almacenado pudiendo volver o extraer un estado [ ]. Aunque no es estrictamente necesario, suele ser muy útil la generación de [ ] con los cambios introducidos entre dos versiones, informes de estado, marcado con nombre identificativo de la versión de un conjunto de ficheros…

4.2 Funcionamiento

Todos los sistemas de control de [ ] se basan en disponer de un [ ], que es el conjunto de información gestionada por el sistema. Este [ ] contiene el historial de versiones de todos los elementos gestionados. Cada uno de los usuarios puede crearse una copia local duplicando el contenido del [ ] para permitir su uso. Es posible duplicar la última versión o cualquier versión [ ] en el historial. Este proceso se suele conocer como “check out” o desproteger. Tras realizar la [ ] es necesario actualizar el [ ] con los cambios realizados. Habitualmente este proceso se denomina [ ], “commit”, “check in” o proteger.

4.3 Clasificación

La principal clasificación que se puede establecer en los sistemas de control de versiones está basada en el almacenamiento:

1. Centralizados: existe un repositorio [ ] de todo el código y documentación, del cual es responsable un único usuario (o conjunto de ellos). Con este sistema se facilitan las tareas [ ] a cambio de reducir flexibilidad, pues todas las decisiones fuertes necesitan la aprobación del responsable. CVS y Subversion son ejemplos de sistemas centralizados.

2. Distribuidos: cada usuario tiene su propio [ ]. No es necesario tomar decisiones centralizadamente. Los distintos repositorios pueden [ ] y mezclar revisiones entre ellos. Git y Mercurial son ejemplos de sistemas distribuidos.