3

Diseño y realización de pruebas

Objetivos

- ✓ Lo que se busca con este capítulo es que conozcas la importancia que tiene para un software la fase de pruebas, que, lejos de ser un mero trámite, ha de tenerse muy en cuenta.
- ✓ Es importante que comprendas los conceptos de procedimientos y casos de pruebas, así como los tipos de pruebas de un software (regresión, funcionales, estructurales, etc.).
- ✓ También es fundamental para un desarrollador conocer la diferencia entre una prueba de caja blanca y otra de caja negra, así como conocer herramientas de depuración de código, y para eso se ha incorporado JUnit y Selenium/Katalon al capítulo.
- ✓ Por último, y para tener una visión más completa de un desarrollo software, tendrás que aprender los conceptos de *planificación de pruebas* y *calidad del software*.

Mapa conceptual

Glosario

Caso de prueba. Conjunto de entradas, condiciones de ejecución y salidas esperadas diseñadas para un objetivo concreto. Los casos de pruebas forman parte de las pruebas a un software.

Fiabilidad. Probabilidad de que un software cumpla con su función sin errores durante un tiempo determinado. Cuanto más fiable sea un software, mejor.

MD5. Sigla del inglés *message digest algorithm 5* (algoritmo de reducción criptográfico). También llamado *función hash*, sirve para saber si algún dato, archivo o información han sido modificados.

Prueba de regresión. Prueba automatizada que puede servir para corroborar, tras la modificación de un software, que todo va a funcionar como debería.

Release. Versión definitiva de un software que puede comercializarse o distribuirse.

RTF. Sigla de revisión técnica formal. Revisión realizada por el auditor y el desarrollador de software a un programa o sistema.

SQA. Sigla del inglés *software quality assurance*. Más conocido como *control de calidad*.

***Tester* o ingeniero de pruebas.** Profesional independiente del equipo de desarrollo, que suele ser programador y poseer amplios conocimientos en informática. Su función es participar en la fase de pruebas de un sistema.

Versión alfa. Primera fase de la versión de un software. Cuando está en fase alfa, el software se prueba en un entorno y características determinadas y controladas por el desarrollador. No es la versión definitiva.

Versión beta. Versión próxima a la versión definitiva. El software no se prueba en el entorno de desarrollo como en la versión alfa, sino que puede probarlo el cliente en su ubicación. La versión beta sirve para detectar anomalías no detectadas durante las pruebas de la versión alfa. Una versión beta puede probarse por muchas personas (pueden llegar a ser miles), mientras que una alfa se prueba solamente por un grupo reducido de personas. Tras probar la versión beta y corregir los errores, se liberará una *release* o versión definitiva.

3.1. Introducción

Siendo realistas, es prácticamente imposible realizar pruebas exhaustivas a un programa. Ya lo decía Edsger Dijkstra en su momento, y es que las pruebas generalmente son demasiado costosas. Salvo que el programa sea tan importante como para realizarlas, lo que se hace es llegar a un punto intermedio en el cual se garantiza que no va a haber defectos importantes o muchos defectos y la aplicación está completamente operativa con un funcionamiento aceptable.

El objetivo de las pruebas es convencer, tanto a los usuarios como a los propios desarrolladores, de que el software es lo suficientemente robusto como para poder trabajar con él de forma productiva.

Cuando un software supera unas pruebas exhaustivas, las probabilidades de que ese software dé problemas en producción se atenúan y, por tanto, su fiabilidad aumenta.

Investiga

¿Quién fue Edsger Dijkstra y por qué tienen tanta importancia sus aseveraciones?

3.2. Procedimientos de pruebas y casos de prueba

Un procedimiento de prueba es la definición del objetivo que desea conseguirse con las pruebas, qué es lo que va a probarse y cómo.

El objetivo de las pruebas no siempre es detectar errores. Muchas veces lo que quiere conseguirse es que el sistema ofrezca un rendimiento determinado, que la interfaz tenga una apariencia y cumpla unas características determinadas, etc.

Por lo tanto, la ausencia de errores en las pruebas nunca significa que el software las supere, pues hay muchos parámetros en juego.

Cuando se diseñan los procedimientos, se deciden las personas que hacen las pruebas y bajo qué parámetros van a realizarse.

No siempre tienen que ser los programadores los que hacen las pruebas. No obstante, siempre tiene que haber personal externo al equipo de desarrollo, puesto que los propios programadores solo prueban las cosas que funcionan (si supieran dónde están los errores, los corregirían).

Hay que tener en cuenta que es imposible probar todo, la prueba exhaustiva no existe. Muchos errores del sistema saldrán en producción cuando el software ya esté implantado, pero siempre se intentará que sea el mínimo número de ellos.

En los planes de pruebas (es un documento que detalla en profundidad las pruebas que se vayan a realizar), generalmente, se cubren los siguientes aspectos:

1. *Introducción*. Breve introducción del sistema describiendo objetivos, estrategia, etc.
2. *Módulos o partes del software por probar.* Detallar cada una de estas partes o módulos.
3. *Características del software por probar.* Tanto individuales como conjuntos de ellas.
4. *Características del software que no ha de probarse.*
5. *Enfoque de las pruebas.* En el que se detallan, entre otros, las personas responsables, la planificación, la duración, etc.
6. *Criterios de validez o invalidez del software.* En este apartado, se registra cuando el software puede darse como válido o como inválido especificando claramente los criterios.
7. *Proceso de pruebas.* Se especificará el proceso y los procedimientos de las pruebas por ejecutar.
8. *Requerimientos del entorno.* Incluyendo niveles de seguridad, comunicaciones, necesidades hardware y software, herramientas, etc.
9. *Homologación o aprobación del plan*. Este plan deberá estar firmado por los interesados o sus responsables.

Las demás fases del proceso de pruebas, como puede entenderse, son el mero desarrollo del plan de pruebas anterior.

PARA SABER MÁS

Edsger Dijkstra (1930-2002), excelente científico holandés que destacó por sus algoritmos, daba soluciones a problemas de una forma sólida y eficiente. Entre sus algoritmos, destacan el algoritmo de Dijkstra, el problema de la cena de los filósofos, los comandos guardados, el algoritmo del banquero o el de *shunting yard*.

Una de sus frases quedará para la historia: "Las pruebas solo pueden demostrar la presencia de errores, no su ausencia". Como Dijkstra sabía bien, cuando se prueba un software, pueden pasarse por alto defectos o errores que saldrán más delante o incluso nunca llegarán a conocerse.

3.2.1. Casos de prueba

En la fase de pruebas, se diseñan y preparan los casos de prueba, que se crean con el objetivo de encontrar fallos.

Por experiencia, no hay que probar los programas de forma redundante. Si se prueba un software y funciona, la mayoría de las veces no hace falta probar lo mismo. Hay que crear otro tipo de pruebas, no repetirlas.

Hay que tener en cuenta que la prueba no debe ser muy sencilla ni muy compleja. Si es muy sencilla, no va a aportar nada y, si es muy compleja, quizá, sea difícil encontrar el origen de los errores.

RECUERDA

✓ Probar es ejecutar casos de prueba uno a uno, pero que un software pase todos los casos de prueba no quiere decir que el programa esté exento de fallos.

Como se ha observado, las pruebas solo encuentran o tratan de encontrar aquellos errores que van buscando, luego, es muy importante realizar un buen diseño de las pruebas con buenos casos de prueba, puesto que se aumenta de esta manera la probabilidad de encontrar fallos.

Ejemplo

Caso de pruebas

Imagínese que se tiene la ventana anterior y desea realizarse un caso de pruebas. La descripción de un caso de pruebas Caso 1 para esta aplicación sería el que se observa en la figura 3.1.

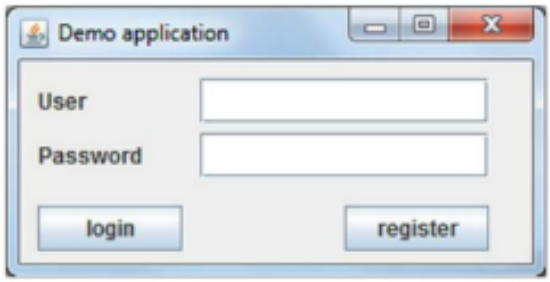

Figura 3.1
Ventana de acceso a una aplicación.

- ✓ *Objetivo:* comprobar que un usuario correcto entra en el sistema y la fecha y hora de entrada queda registrada.
- ✓ *Entrada de datos:* en el campo User, se introduce "myfpschool" y, en el campo Password, "Troconne77".
- ✓ *Condiciones:* en la tabla Usuarios, existe el usuario myfpschool con la contraseña Troconne77 encriptada en MD5.
- ✓ *Resultado:* el usuario entra en el sistema y, en la tabla Log, deja un registro ("myfpschool",fecha,hora).
- ✓ *Procedimiento de la prueba:*

1. Se comprueba en la tabla Usuarios que existe el usuario por introducir.
2. Se comprueba que la contraseña esté codificada en MD5 correctamente.
3. En los campos User y Password, se teclean los datos "myfpschool" y "Troconne77".
4. Se pulsa Aceptar y se comprueba que se accede al sistema correctamente.
5. Se revisa la tabla Log y se comprueba que se ha registrado el usuario, la fecha y la hora actual.

Actividades propuestas

Indica si las siguientes afirmaciones son verdaderas o falsas y razona tus respuestas:

V F 3.1. Muchas veces no se hacen todas las pruebas que se desearían porque son muy costosas.

V F 3.2. Lo más normal es que las pruebas las realicen los programadores porque son los que mejor conocen la aplicación.

Verificar

3.2.2. Codificación y ejecución de las pruebas

Una vez diseñados los casos de prueba, hay que generar las condiciones necesarias para poder ejecutar dichos casos de prueba. Habrá que codificarlos en muchos casos generando set o conjuntos de datos. En estos set de datos, hay que incluir tanto datos válidos e inválidos como algunos datos fuera de rango o disparatados.

SABÍAS QUE...

El *beta testing*

Muchas empresas de videojuegos, software, etc., ofrecen una primera versión o *release* a una comunidad determinada de usuarios para que la prueben. El objetivo de esta versión beta es descubrir el mayor número de errores posibles. Es más fácil (y más barato) descubrir errores por parte de decenas, centenas o miles de usuarios que por un reducido número de ingenieros de calidad.

También habrá que preparar las máquinas sobre las que van a hacerse las pruebas instalando el software necesario, los usuarios de sistema, realizar carga del sistema, etc.

FUNDAMENTAL

Ejecución de las pruebas

Una vez definidos los casos de prueba y establecido el entorno de las pruebas, es el momento de su ejecución.

Irán ejecutándose los casos de prueba uno a uno y, cuando se detecte algún error, hay que aislarlo y anotar la acción que estaba probándose, el caso, el módulo, la fecha, la hora, los datos utilizados, etc. De esa manera, intentará documentarse lo más detalladamente posible el error.

En el caso de que se produzcan errores aleatorios, también hay que registrarlos anotando este hecho.

3.3. Tipos de pruebas: funcionales, estructurales y regresión

El lector ya conoce qué son las pruebas y qué objetivos tienen. En cuanto al tipo de pruebas por realizar, existen muchas categorías. A continuación, se repasan las más frecuentes.

En primer lugar, existen las *pruebas funcionales,* que, como su nombre indica, buscan que los componentes software diseñados cumplan con la función con la que fueron diseñados y desarrollados. Estas pruebas buscan lo que el sistema hace, más que cómo lo hace.

Todos los sistemas tienen una serie de funcionalidades o características y esas son las que van a testearse.

RECUERDA

✓ Las pruebas funcionales buscan comprobar la funcionalidad del sistema.

El *tester* (o ingeniero de pruebas), para la realización de las pruebas, se basa en la documentación existente (manual de usuario y otros manuales). También suelen realizarse pruebas en conjunto con los usuarios, puesto que ellos saben cómo tiene que funcionar el sistema.

Estas pruebas suelen considerarse como pruebas de *caja negra*. No se evalúa cómo el sistema funciona internamente, pero sí qué es lo que hace.

También existe otro tipo de pruebas como pueden ser las de seguridad, en las que se evalúan aspectos de seguridad, o las de interoperabilidad, cuando existen interfaces entre el sistema y otros. Se prueba la compatibilidad entre el sistema y los demás sistemas con los que interactúa.

Las *pruebas no funcionales* son aquellas pruebas más técnicas que se realizan al sistema. Estas siguen siendo de caja negra, puesto que nunca se examina la lógica interna de la aplicación.

Suelen ser pruebas no funcionales las pruebas de carga, pruebas de estrés, pruebas de rendimiento, pruebas de fiabilidad, etc.

Entre las pruebas que examinan de forma más detallada la arquitectura de la aplicación, están las *pruebas estructurales,* que son de caja blanca, puesto que, en algún momento, se utilizan técnicas de análisis del código. Generalmente, para este tipo de pruebas, se utilizan herramientas especializadas.

Otro tipo de pruebas son las *pruebas de regresión* o *pruebas repetidas*. No suele probarse lo que ya se ha probado, pero, en el caso de que el software haya sido modificado, generalmente, se realiza este tipo de pruebas.

Estas pruebas intentan descubrir si existe algún error tras las modificaciones o si se encuentra algún tipo de problema que no se había descubierto previamente.

Solamente se realizarán estas pruebas en el caso de que haya una modificación de software.

En muchos casos, este tipo de pruebas no sirven para descubrir nuevos errores, sino para certificar su ausencia, pues que no aparezcan errores durante estas pruebas no significa que el software esté exento de ellos.

Este tipo de pruebas de regresión suelen automatizarse y se agrupan en conjuntos llamados *regression test suites* (conjuntos de pruebas de regresión).

Actividades propuestas

Indica si las siguientes afirmaciones son verdaderas o falsas y razona tus respuestas:

V F 3.3. Una versión alfa está más terminada que una versión beta.

V F 3.4. En las pruebas funcionales, se verifica que el software técnicamente funciona de forma correcta y que el rendimiento es el esperado.

3.4. Pruebas de caja blanca

En las pruebas de caja blanca, se conoce o se tiene en cuenta el código que quiere probarse. Se denomina también *clear box testing* porque la persona que realiza las pruebas está en contacto con el código fuente. Su objetivo es probar el código, cada uno de sus elementos.

Existen algunas clases de pruebas de este tipo como, por ejemplo:

- Pruebas de cubrimiento.
- Pruebas de condiciones.
- Pruebas de bucles.

Se verá a continuación en qué consiste cada una de estas pruebas.

3.4.1. Pruebas de cubrimiento

En este tipo de pruebas, el objetivo es ejecutar, al menos una vez, todas las sentencias o líneas del programa.

En ocasiones, es imposible cubrir el 100% del código porque puede haber condiciones que nunca se cumplan:

```
if (a > 20 && a < 10) { …. }
```

O también puede haber excepciones o notificaciones de error en un código que nunca va a fallar (código sin errores).

TOMA NOTA

El número ciclomático es el que indica el número de circuitos que existen en un grafo (circuito). Indica el número de maneras que hay de ir desde un nodo hasta sí mismo sin pasar dos veces por el mismo arco.

Para realizar las pruebas, habrá que generar el suficiente número de casos de prueba para poder cubrir los distintos caminos independientes del código. En cada condición, deberá cumplirse en un caso y en otro no.

```
int isFreaky(int videogames,
    int manga, int technology){
    if(videogames>0){
        if(manga>0){
            if(technology>0){
            return 1;
            }
            else{
            return 0;
            }
        else{
        return 0;
        }
    else{
    return 0;
    }
}
```

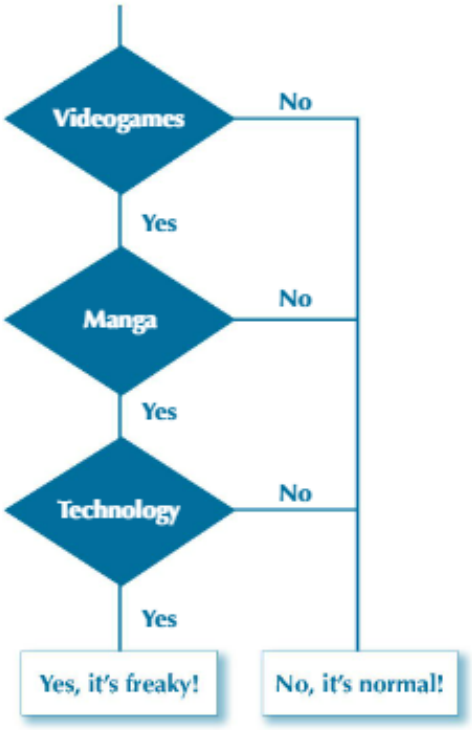

Figura 3.2
Diagrama de flujo de una función.

En la figura 3.2, puede observarse cómo una función determinada, para averiguar si una persona es un poco friki, puede generar un grafo con varias posibilidades.

Para hacer la prueba de cubrimiento, existen dos opciones. La primera es crear casos de pruebas para ejecutar las líneas de código al menos una vez y, por lo tanto, los conjuntos de datos de pruebas *({videogames=0,manga=0,technology=0}, {videogames=1,manga=0,technology=0}, {videogames=1,manga=1,technology=0}, {videogames=1,manga=1,technology=1})* serían válidos.

Si lo que quiere hacerse es comprobar todas y cada una de las combinaciones de caminos, el número de casos de prueba se dispara. En esta función no habría muchos caminos, pero en un programa más extenso el número de combinaciones sería muy elevado.

Actividades propuestas

Indica si las siguientes afirmaciones son verdaderas o falsas y razona tus respuestas:

V F 3.5. Existen pruebas que están pensadas para repetirlas un número indefinido de veces. Estas, generalmente, son pruebas automatizadas.

V F 3.6. Las pruebas de regresión, generalmente, son pruebas de caja blanca.

Verificar

3.4.2. Prueba de condiciones

En este caso, se necesitarán varios casos de prueba. En una condición, puede haber varias condiciones simples y habrá que generar un caso de pruebas por cada operando lógico o comparación. La idea es que, en cada expresión, se cumpla en un caso y en otro no.

Véase un ejemplo:

```
if (videogames=1 && manga=1 && technology=1){ freaky = 1}
```

En el caso anterior, deberán comprobarse todas y cada una de las combinaciones de las tres variables anteriores. En esta ocasión, son variables, pero podrían ser otro tipo de expresiones más complejas.

De esa manera, habrá que cerciorarse de que cualquiera de las combinaciones de valores de la condición funcionará tal y como el programa fue concebido.

3.4.3. Prueba de bucles

Los bucles son estructuras que se basan en la repetición, por lo tanto, la prueba de bucles se basará en la repetición de un número especial de veces.

En el caso de un *bucle simple,* los casos de prueba deberían contemplar lo siguiente:

a) Repetir el máximo, máximo −1 y máximo +1 veces el bucle para ver si el resultado del código es el esperado.
b) Repetir el bucle cero y una vez.
c) Repetir el bucle un número determinado de veces.

En el caso de *bucles anidados,* existirán bucles internos y externos. Sería bueno realizar la prueba de bucles simple para los bucles internos ejecutando el bucle externo un número determinado de veces y, luego, realizar la prueba contraria. El bucle interno se ejecuta un número determinado de veces y el externo se prueba con las pruebas anteriores de bucle simple.

3.5. Pruebas de caja negra

Entre las pruebas de caja negra (aquellas que simplemente prueban la interfaz sin tener en cuenta el código), pueden citarse las siguientes:

- Pruebas de cubrimiento.
- Pruebas de clases de equivalencia de datos.
- Pruebas de valores límite.

3.5.1. Prueba de clases de equivalencia de datos

Imagínese que está probándose una interfaz y debe generarse un código de usuario y una clave.

El sistema dice que el código de usuario tendrá que tener mayúsculas y minúsculas, no puede tener caracteres que no sean alfabéticos y ha de tener, al menos, 6 letras (máxi-

mo 12). Las contraseñas tendrán, al menos, 8 caracteres (máximo 10) y contendrán letras y números.

Para testear esta interfaz, lo que debe hacerse es establecer clases de equivalencia para cada uno de los campos. Tendrán que crearse clases válidas y clases inválidas por cada uno de los campos. Por ejemplo:

1. *Usuario:*

   — *Clases válidas:* "Pelegrino" y "Rocinante".
   — *Clases inválidas:* "marrullero44", "nene", "Portaavionesgigante", "Z&aratustra" y "Ventajoso12".

2. *Contraseña:*

   — *Clases válidas:* "5Entrevias" y "s8brino".
   — *Clases inválidas:* "corta", "muyperoquemuylarguisima", "oletugarbo" y "999999999".

RECUERDA

- ✓ El objetivo de esta prueba es comprobar todas las clases válidas y las inválidas al menos una vez.
- ✓ Cada vez que se diseña un caso de prueba con datos inválidos, se introducirá solamente una clase inválida. De esa manera, se conocerá si el programa está funcionando correctamente.
- ✓ Muchas veces, al utilizar varias clases inválidas, los errores se enmascaran y no puede conocerse si todas las clases funcionan.

3.5.2. Prueba de valores límite

Este tipo de pruebas son complementarias a las pruebas de particiones. El objetivo es generar valores que puedan probar si la interfaz y el programa funcionan correctamente. Imagínese que se accede a la página web de un banco para testearla y la interfaz, cuando va a transferirse una cantidad, comunica: "La cifra máxima que usted puede transferir hoy es de 10 000 euros".

Si quiere probarse dicha interfaz, el *tester* probaría, por ejemplo, valores fuera de rango como −100 o 20 000; también valores en los límites como 0, 1, 9999, 10 000 y 10 001, o valores típicos e intermedios como 9000 o 2500.

El objeto de esta prueba se halla en que, muchas veces, los programadores se equivocan al establecer los límites en la frontera (se equivocan y ponen $<$ en vez de $\leq$, por ejemplo).

3.5.3. Prueba de interfaces

Una interfaz de usuario o GUI *(graphical user interface)*, generalmente, se testea con una técnica que se denomina *prueba de interfaces.*

Generalmente, una interfaz es una serie de objetos con una serie de propiedades. Toda esta serie de objetos con sus propiedades en su conjunto formarán la interfaz.

Esos objetos van tomando valores durante la ejecución del programa. En esa ejecución, el usuario va introduciendo valores en la interfaz y haciendo clic sobre algunos objetos.

Dependiendo de las entradas, la interfaz proporcionará una salida determinada. Esa salida debería ser la esperada. Muchas veces, cuando se testea un programa, hay que conocer su funcionalidad.

A) Cómo testear una interfaz

Una primera prueba puede consistir en seguir el manual de usuario. El *tester* deberá introducir datos (mejor datos reales que inventados) como si se tratase del propio usuario y comprobar que las salidas proporcionadas son las esperadas.

Si el software pasa esta prueba, entonces, podrá pasar a sufrir un testeo más serio utilizando casos de prueba.

B) Testear la usabilidad

Tiene por objeto evaluar si el producto generado va a resultar lo esperado por el usuario. Hay que ver y trabajar con la interfaz desde el punto de vista del usuario. Además, en su testeo, deberían utilizarse datos reales.

Solamente, al observar cómo el usuario interactúa con el software y escuchando su *feedback*, pueden detectarse aquellas características de este que lo hacen difícil y tedioso de utilizar. Una vez detectadas esas disfunciones, se realizarán los cambios pertinentes, de tal manera que el software sea fácil de usar y eficiente. Es importante escuchar la opinión del cliente porque, a la postre, es la persona que va a trabajar de forma sistemática con el software.

Los test de usabilidad deben estar integrados en el ciclo de vida de desarrollo del software. La usabilidad ha de tenerse en cuenta no solamente en el momento de realizar las pruebas, sino también en el momento de diseñar la interfaz. A la hora de diseñar la interfaz, habría que hacerse las siguientes preguntas:

- ¿Los usuarios comprenderán cómo funciona la interfaz de una manera sencilla?
- ¿Es la interfaz lo suficientemente rápido y eficiente para el usuario?

Téngase en cuenta que, muchas veces, en las interfaces, se echan de menos teclas rápidas o combinaciones de teclas, valores por defecto, autocompletar, etc.

Además, si solamente va a comprarse un nuevo software que va a ser utilizado por 300 personas, por ejemplo, habrá que tener muy en cuenta la usabilidad. El objetivo es que los usuarios hagan el trabajo mejor y más rápido, por lo tanto, la usabilidad en una compra de software tiene mucho que decir.

C) Testear la accesibilidad

Mucha gente no sabe que la accesibilidad no solamente es que el software esté diseñado para usuarios con discapacidad, sino que también sea accesible por *frameworks* de test automatizados.

Un software es accesible cuando el programa o aplicación se adecua a los usuarios con discapacidad, pueden hacer su trabajo de forma efectiva y la satisfacción con él es buena. Además, hay que tener en cuenta que, en ocasiones, hay estándares y requerimientos preestablecidos de accesibilidad que el software ha de cumplir.

Algunos software son testeados por expertos en accesibilidad que realizan auditorías de este, de tal manera que determinan si ese software supera los requerimientos exigibles.

Selenium/Katalon

Investiga

¿Qué es una auditoría informática y cuál es el perfil de un auditor informático?

3.6. Herramientas de depuración de código

Va a tomarse como ejemplo JUnit, un *framework* que permite realizar test repetibles (pruebas de regresión), es decir, que puede diseñarse un test para un programa o clase concreta y ejecutarlo tantas veces como sea necesario. La ventaja es que puede (o mejor, debe) ejecutarse el test cada vez que se modifique o cambie algo del código y verificar si el programa sigue funcionando correctamente tras los cambios.

Crear una primera clase de test

Deshabilitar un test

Trabajar con excepciones

Crear un conjunto de test o *suite de test*

Actividades propuestas

Indica si las siguientes afirmaciones son verdaderas o falsas y razona tus respuestas:

V F 3.7. La orden @Test(timeout=10) hace que el test falle si tarda más de 10 segundos.

V F 3.8. El método de inicialización @BeforeClass se ejecutará antes de cada test.

Verificar

3.7. Planificación de pruebas

La planificación de las pruebas es un punto importante en la toma de decisiones de un proyecto. Qué tipo de pruebas y cuándo van a realizarse son preguntas que hay que tener en cuenta desde el principio.

Si los proyectos son grandes (tienen envergadura), lo más normal es que se realicen las pruebas que se comentarán a continuación. En caso contrario (proyectos pequeños con bajo presupuesto), serán los responsables técnicos del desarrollo los que tendrán que establecer la estrategia adecuada.

A continuación, se resumen las pruebas y el momento en el que se realizan.

3.7.1. Pruebas unitarias

Suelen realizarse durante las primeras fases de diseño y desarrollo. Obviamente, no hay que demorar mucho la realización de dichas pruebas, puesto que luego hay que integrar todo el software (las distintas unidades) y los fallos van acumulándose y la localización y diagnóstico se complican.

En el caso de la POO, las pruebas unitarias deberían realizarse a nivel de objeto y, luego, a nivel de paquete o librería. No tiene sentido probar un paquete por separado si no se han realizado pruebas de los objetos a nivel individual.

3.7.2. Pruebas de integración

Una vez que los componentes individuales se han probado, es momento de ir integrando módulos. Las pruebas de integración tendrán que hacerse al final de la fase de diseño (se realizarán pruebas para corroborar que el diseño es factible y eficiente) y también al final de la fase de codificación (una vez realizadas todas las pruebas individuales, se integran componentes y se prueban en conjunto verificando que funcionan correctamente de forma conjunta).

Existen *pruebas de integración ascendentes y descendentes*. Pueden probarse los módulos más generales y, luego, ir a los más específicos o al contrario.

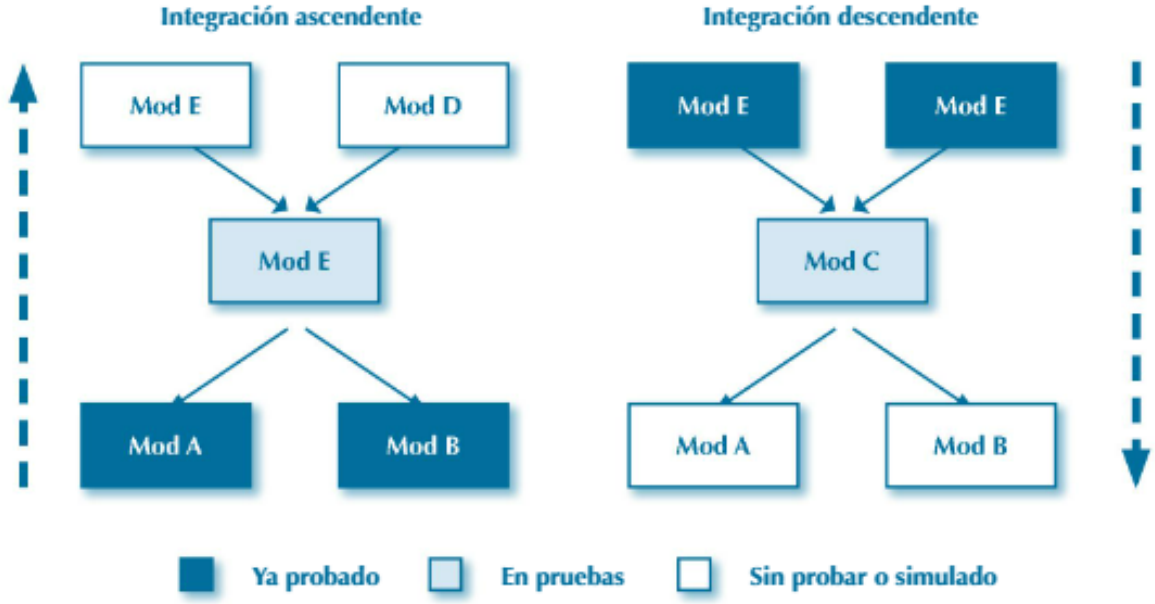

Figura 3.3
Pruebas de integración ascendente y descendente.

Como puede verse en la figura 3.3, la filosofía de las pruebas de integración ascendentes es contraria a las descendentes. En las descendentes, generalmente, hay que hacer módulos de pruebas o programas de pruebas para probar unitariamente los módulos individuales, mientras que, en las descendentes, muchas veces, hay que crear módulos, objetos y clases ficticias para probar partes más generales del programa.

Aunque son filosofías diferentes, el objetivo es siempre el mismo: probar que el sistema en conjunto funciona correctamente.

3.7.3. Pruebas de aceptación o validación

Este tipo de pruebas tratan de probar el sistema completo. Además de probar que los requisitos del programa se cumplen uno por uno, el equipo de pruebas mirará también si técnicamente el programa es estable y no tiene ningún fallo.

Además, habrá que probar el rendimiento del sistema modificando la carga y observando su evolución. También se harán pruebas de estrés para cerciorarse de que el sistema va a responder eficientemente ante cualquier eventualidad.

Existen en este estadio pruebas *alfa,* las cuales se realizan en un entorno controlado y bajo unas especificaciones concretas, y pruebas *beta,* en las que los usuarios prueban el sistema en un entorno no controlado por los desarrolladores.

3.7.4. Automatización de pruebas

Muchas veces, es necesario automatizar las pruebas o repetir las mismas pruebas tras realizar mantenimientos, modificaciones o correcciones del software.

Es siempre bueno conservar los datos, set de pruebas, programas y módulos de prueba, puesto que no se sabe si van a ser necesarios en un futuro.

En el caso de que se hayan utilizado herramientas u otro sistema, se documentará y se almacenarán en un repositorio para su ejecución automática posterior.

Investiga

¿En qué consistió el efecto 2000? ¿Tuvo mucha repercusión en las empresas?

3.8. Calidad del software

La calidad es un tema que, desde hace años, tiene una importancia en el mundo de la comercialización de productos. El mercado actual es muy competitivo y la calidad es uno de los aspectos diferenciales que hace que un producto triunfe o fracase. Basta citar a Blackberry, empresa que, en medio de la vorágine de un mercado tan competitivo como el de la telefonía móvil, cometió varios errores tanto estratégicos como técnicos que la relegaron a un plano poco significativo.

Investiga

Elabora una lista con las razones por las que se originó la crisis del software. ¿Cómo piensas que podría haberse solucionado este problema?

Dadas sus características, garantizar la calidad del software es un proceso mucho más difícil que el de otro producto, dado que un proceso industrial es más fácil de testear que el proceso de desarrollo de software.

El software tiene que estar libre de defectos y de errores y también tiene que adecuarse a los parámetros con los cuales se ha diseñado y desarrollado. Las aplicaciones informáticas están presentes en multitud de ámbitos. Existe tanto software empotrado como aplicaciones en dispositivos que nadie puede imaginarse (lavadoras, televisiones, aires acondicionados, etc.).

En los años noventa, se vivió una crisis del software. Fueron en esos años en los que la calidad y el proceso de desarrollo no tenían mucha importancia en los que se vivieron las consecuencias de desarrollar un software con poca profesionalidad en muchos casos. Algunas características de esta crisis fueron:

a) *Calidad insuficiente del producto final*. Muchos de los errores tenían su base en un análisis pobre con una poca comunicación con el cliente.
b) *Estimaciones de duración de proyectos y asignación de recursos inexactas.* Con el problema que ello conlleva.
c) *Escasez de personal cualificado en un mercado laboral de alta demanda*. Algunos programas no estaban desarrollados bajo el paradigma de la programación estructurada. No tenían una estructura racional ni lógica, con lo cual los errores se multiplicaban y el mantenimiento era un suplicio.
d) *Tendencia al crecimiento del volumen y complejidad de los productos.* En algunos casos, dichos desarrollos complejos estaban poco probados, pobremente documentados, etc.

Investiga

¿Qué son las cárnicas del software y por qué han contribuido a que la calidad de los programas informáticos haya descendido?

Con el tiempo, se ha constatado que la calidad no se mide solamente por unos parámetros de funcionamiento, sino que hay otros aspectos que son importantes, como el soporte, es decir, el respaldo organizacional que tiene un producto como la formación, la asistencia a problemas inesperados y el mantenimiento permanente y efectivo.

Para valorar dicha calidad, se lleva a cabo la evaluación y el rendimiento de las aplicaciones.

Las mediciones de rendimiento de un software pueden estar orientadas hacia el usuario (tiempos de respuesta) u orientadas hacia el sistema (uso de la CPU). Son medidas típicas del rendimiento diferentes variables de tiempo (tiempo de retorno, tiempo de respuesta y tiempo de reacción), la capacidad de ejecución, la carga de trabajo, la utilización, etc.

TOMA NOTA

¿Qué es una prueba de rendimiento?

Se realizan este tipo de pruebas para comprobar cómo el software realiza una tarea determinada en un sistema con unas condiciones de trabajo concretas. Supuestamente, con una carga de trabajo normal (habitual).

Estas pruebas de rendimiento pueden servir para evaluar otros parámetros como el uso de los recursos, la fiabilidad del software, la escalabilidad del sistema, etc.

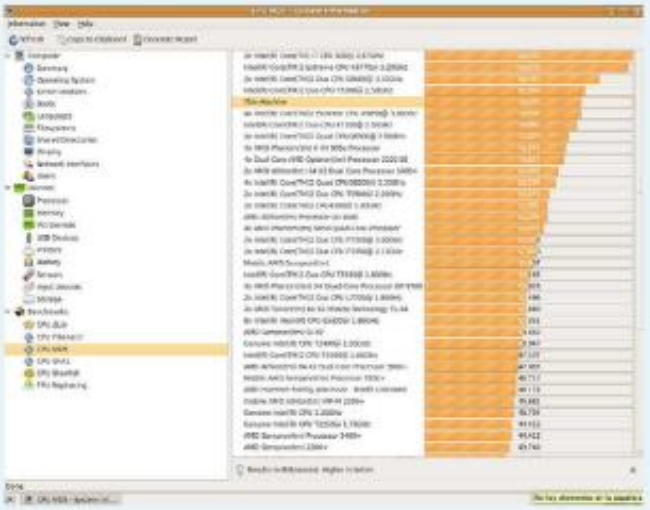

Figura 3.4
Benchmark o prueba de rendimiento de un sistema.

Para evaluar el software, es necesario contar con criterios adecuados que permitan analizar el software desde diferentes puntos de vista.

Las *pruebas de carga* se realizan sobre el sistema simulando una serie de peticiones esperadas o un número de usuarios esperado trabajando de forma concurrente, realizando un número de transacciones determinado. En estas pruebas, se evalúan los tiempos de respuesta de las transacciones. Generalmente, se realizan varios tipos de carga (baja, media y alta) para evaluar el impacto y poder graficar el rendimiento del sistema.

Investiga

Busca herramientas para la realización de pruebas de carga y rendimiento y cita al menos cuatro de ellas.

Otro tipo de pruebas bastante útiles son las *pruebas de estrés* en las que la carga va elevándose más y más para ver cómo de sólida es la aplicación y cómo se maneja ante un número de usuarios y transacciones extremos.

También existen otros tipos de pruebas como las *pruebas de estabilidad* donde se somete de forma continuada al sistema a una carga determinada o bien *pruebas de picos* donde el volumen de carga va cambiando.

PARA SABER MÁS

Un *benchmark* es una aplicación o conjunto de aplicaciones cuya finalidad es evaluar el rendimiento de un sistema.

Existen cuatro categorías generales de pruebas de comparación:

- ✓ *Pruebas de aplicaciones base*: que se encargan de ejecutar y cronometrar los tiempos de estas.
- ✓ *Pruebas* playback: que usan llamadas al sistema durante actividades específicas de una aplicación como uso de disco o llamadas a rutinas de gráficos, ejecutándolas aisladamente.
- ✓ *Pruebas sintéticas:* que enlazan actividades de la aplicación en subsistemas específicos.
- ✓ *Pruebas de inspección:* que no intentan imitar la actividad, sino que las ejecuta directamente en su entorno productivo.

Existen múltiples programas para llevar a cabo este tipo de pruebas: Hardinfo, Winstone o Winbench de ZDNet, etc.

3.8.1. Medidas o métricas de calidad del software

Se definen los criterios de calidad (o factores de calidad) de un software al principio de un proyecto y dichos criterios siguen teniéndose en cuenta durante toda su vida.

No puede existir ningún criterio o factor de calidad que no pueda medirse. Algunos criterios de calidad pueden ser los siguientes:

- Número de errores por un número determinado de líneas de código.
- Número medio de revisiones realizadas a una función o módulo de programa.

Generalmente, para evaluar los criterios de calidad, se realizan RTF o revisiones técnicas formales.

TOMA NOTA

¿Qué es una revisión técnica formal o RTF?

El objetivo de una RTF es descubrir errores. Tanto en la lógica como en la funcionalidad, implementación, diseño, etc.

Se evalúa el software para comprobar que sigue los estándares y protocolos establecidos.

Estas RTF las realiza el responsable de calidad (responsable de SQA). Como es una revisión formal, se cita a las partes implicadas y se informa de cómo va a llevarse a cabo y las responsabilidades.

No se revisa todo el software, sino una parte de él.

Una vez que se termina la reunión, el responsable de SQA deberá emitir un informe con los errores o desviaciones detectadas y las acciones correctivas que tienen que llevarse a cabo. Dicho responsable también tendrá que revisar que las correcciones se realizan.

Los criterios o factores de calidad, como no podía ser de otra forma, se establecen mediante métricas o medidas. Véanse algunas de las métricas de calidad más utilizadas:

1. *Tolerancia a errores.* Mide los efectos que tiene un error sobre el software en conjunto. El objetivo es que no haya errores, pero, si los hay, que sus efectos sean limitados.
2. *Facilidad de expansión.* Mide la facilidad con la que pueden añadírsele nuevas funcionalidades a un software concreto. Cuanto más fácil sea de ampliar, mejor.
3. *Independencia de plataforma del hardware.* Es sabido que un programa en Java es de los más independientes que existen. Cuanto mayor sea el número de plataformas donde pueda ejecutarse un software, mejor.
4. *Modularidad.* Número de componentes independientes de un programa.
5. *Estandarización de los datos.* Se evalúa si se utilizan estructuras de datos estándar a lo largo de un programa.

¿Qué has aprendido?

- ✓ Una vez vistos todos los conceptos de este capítulo, podrás comprender que las pruebas a un software son algo fundamental y muy importante.
- ✓ Un buen desarrollador nunca liberará una *release* a un cliente o al mercado sin haberle hecho las pruebas oportunas.
- ✓ Ten en cuenta que el concepto de pruebas abarca prácticamente todo el ciclo de vida del proyecto. Hay que verificar y revisar todo, desde documentos de análisis al software mismo.
- ✓ Y, sobre todo, ten presente que un software a la larga es más caro si no se prueba y no tiene calidad que si un equipo independiente hace un buen trabajo de revisión. Los fallos o errores o falta de calidad en un software son como una bola de nieve: cuanto más rueda por la ladera, más grande se hace y llega un momento en el que pararla es prácticamente imposible. Si se hubiese hecho al principio, sería todo más fácil.

Resumen

- Los procedimientos de prueba son la definición del objetivo que desea conseguirse con las pruebas y qué es lo que va a probarse y cómo.
- Por lo tanto, la ausencia de errores en las pruebas nunca significa que el software supere la prueba.
- Los programas deberían probarse por personal ajeno al proyecto.
- Un caso de prueba es un documento que se crea con el objetivo de encontrar fallos. Probar es ejecutar casos de prueba uno a uno, pero el que un software pase todos los casos de prueba no quiere decir que el programa esté exento de fallos.

- El *beta testing* es la prueba de un producto software antes de que salga una versión definitiva al mercado.
- Las pruebas funcionales, como su nombre indica, buscan que los componentes software diseñados cumplan con la función con la que fueron diseñados y desarrollados.
- En las pruebas estructurales, se comprueba la aplicación de forma más técnica y pueden llegar a ser pruebas de caja blanca.
- Las pruebas de regresión o las pruebas repetidas son aquellas que se hacen al software para comprobar, tras un cambio, que todo funciona correctamente.
- Entre las pruebas de caja blanca, están las siguientes:

  — Pruebas de cubrimiento.
  — Pruebas de condiciones.
  — Pruebas de bucles.

- Entre las pruebas de caja negra, están las siguientes:

  — Pruebas de cubrimiento.
  — Pruebas de clases de equivalencia de datos.
  — Pruebas de valores límite.

- Es importante testear la usabilidad y la accesibilidad de un software.
- JUnit es un *framework* que permite realizar test repetibles (pruebas de regresión), eso quiere decir que puede diseñarse un test para un programa o clase concreta y ejecutarlo tantas veces como sea necesario.
- La planificación de las pruebas es un punto importante en la toma de decisiones de un proyecto porque hay que dilucidar qué tipo de pruebas van a hacerse y cuándo van a realizarse.
- Las pruebas unitarias suelen realizarse durante las primeras fases de diseño y desarrollo.
- Las pruebas de integración se realizan una vez que los componentes individuales se han probado. Existen pruebas de integración ascendentes y descendentes.
- Las pruebas de aceptación o validación tratan de probar el sistema completo.
- Muchas veces, es necesario automatizar las pruebas o repetir las mismas pruebas tras realizar mantenimientos, modificaciones o correcciones del software.
- El software tiene que estar libre de defectos y de errores y también ha de adecuarse a los parámetros con los cuales se ha diseñado y desarrollado. La calidad sirve para garantizar esto.
- Las métricas de calidad sirven para medir los criterios de calidad de un software. Algunas métricas son:

  — Tolerancia a errores.
  — Facilidad de expansión.
  — Independencia de plataforma del hardware.
  — Modularidad. Número de componentes independientes de un programa.
  — Estandarización de los datos.

Ejercicios prácticos

1. **¿Qué aspectos crees que debería tener en cuenta un plan de pruebas?**
2. **Si se tiene la siguiente condición:**

```
if ( Valencia && Barcelona && Madrid ){}
```

   En una prueba de condiciones, qué es lo que tendrá que compararse.

3. **Imagina que quiere realizarse un test de JUnit y desea conectarse con una base de datos para recuperar datos de prueba. ¿Qué etiqueta debería utilizarse?**
4. **¿Compensa a una compañía enviar una versión beta a un gran número de *gamers*?**

   Razona tu respuesta.

5. **Realiza un test en Katalon.**

   Realiza un test que cargue la página myfpschool.com.
   Una vez cargada, ha de ejecutarse el *link* "Entornos de desarrollo" y, cuando cargue la página, se verificará que existe el texto "mandamientos".

6. **Crea una clase de test en JUnit.**

   Dimas quiere realizar una clase que convierta grados Fahrenheit a Celsius, y viceversa. Conoce la fórmula y quiere implementar dos métodos en la clase fahrenheittocelsius() y celsiustofahrenheit() que conviertan grados de una unidad a otra, y viceversa.
   Además, quiere testear que convierten los métodos correctamente los valores –5, 0, 15 y 32.
   ¿Puedes ayudarlo a crear la clase y el test con JUnit 4?

7. **Crea un conjunto de test o suite de test en JUnit.**

   Dimas se ha animado y, dentro del mismo proyecto, quiere realizar una clase que convierta números romanos a decimales, y viceversa (métodos roman2dec y dec2roman). Quiere realizar un par de test con JUnit para que testee los números XXI y 2016.
   Además, cree que puede ser útil una segunda clase que convierta dólares a euros, y viceversa (métodos dollar2euro y euro2dollar), también con sus test en JUnit que testeen 10,5 dólares y 20,30 euros.
   Por último, para no realizar los test por separado, lo que se propone es crear una *suite* de pruebas que testee las tres clases.

AUTOEVALUACIÓN

1. Una RTF es:

   - ○ *a)* La revisión del software realizada por un ingeniero de pruebas.
   - ○ *b)* La revisión del software realizada por el equipo de desarrollo.
   - ○ *c)* La revisión del software realizada por el auditor y el desarrollador de software.

2. Las pruebas deberían realizarse siempre:

   - ○ *a)* Por programadores.
   - ○ *b)* Por personal ajeno al proyecto.
   - ○ *c)* Por programadores y personal ajeno al proyecto.

3. El *beta testing*:

   - ○ *a)* Intentará descubrir el mayor número de errores posibles.
   - ○ *b)* Es realizado por muchos ingenieros de pruebas.
   - ○ *c)* Se realiza en las instalaciones de desarrollo.

4. Las pruebas funcionales:

   - ○ *a)* Son pruebas de caja negra.
   - ○ *b)* Se llaman también *pruebas de regresión*.
   - ○ *c)* Son pruebas de caja blanca.

5. JUnit permite realizar pruebas:

   - ○ *a)* De caja blanca.
   - ○ *b)* Funcionales.
   - ○ *c)* De regresión.

6. Son pruebas *clear box testing*:

   - ○ *a)* Pruebas de cubrimiento.
   - ○ *b)* Pruebas funcionales.
   - ○ *c)* Pruebas de regresión.

7. JUnit es:

- a) Una librería.
- b) Un *framework*.
- c) Un *plugin* de NetBeans.

8. Las pruebas de integración son:

- a) Descendentes.
- b) Ascendentes.
- c) Ascendentes y descendentes.

9. ¿Cuál de las siguientes secuencias es la correcta?

- a) Versión alfa, beta y *release*.
- b) Versión beta, alfa y *release*.
- c) Versión *release*, alfa y beta.

10. ¿Cuál de las siguientes no es métrica de calidad de un software?

- a) La tolerancia a errores.
- b) La facilidad de expansión.
- c) El tiempo dedicado a las pruebas.

SOLUCIONES

1.	a	b	**c**	6.	**a**	b	c
2.	a	b	**c**	7.	a	**b**	c
3.	**a**	b	c	8.	a	b	**c**
4.	**a**	b	c	9.	**a**	b	c
5.	a	b	**c**	10.	a	b	**c**