ENTORNOS DE DESARROLLO

UT3: Diseño y realización de pruebas II

CFGS Desarrollo de Aplicaciones Multiplataforma
Curso 2024-25

Índice de contenidos

1. Pruebas de caja blanca....2
   1.1 Pruebas de cubrimiento....2
   1.2 Prueba de condiciones....2
   1.3 Prueba de bucles....3
2. Pruebas de caja negra....3
   2.1 Pruebas de clases de equivalencia de datos....3
   2.2 Prueba de valores límite....4
   2.3 Prueba de interfaces....4
3. Dobles de prueba....4
   3.1 Tipos de dobles de pruebas....5
4. Herramientas de depuración de código....6
   4.1 Funcionalidades principales....6
   4.2 Funcionamiento....7

1. Pruebas de caja blanca

Las pruebas de caja blanca son técnicas que evalúan el funcionamiento interno del software, centrándose en su estructura y lógica. Veamos tres tipos específicos de pruebas de caja blanca:

1.1 Pruebas de cubrimiento

Definición: Las pruebas de cubrimiento (o cobertura de código) evalúan qué partes del código fuente han sido ejecutadas durante las pruebas. Su objetivo es asegurar que todas las líneas de código, declaraciones y ramas se prueben adecuadamente.

Tipos de cubrimiento:

- **Cubrimiento de línea (o declaración)**: Mide el porcentaje de líneas de código que se ejecutan durante las pruebas.
- **Cubrimiento de ramas**: Evalúa que todas las ramas (condiciones if, else, etc.) se hayan probado, asegurando que cada camino lógico del código se haya evaluado.

Importancia: Ayuda a identificar partes del código que no han sido probadas y que podrían contener errores ocultos.

1.2 Prueba de condiciones

Definición: La prueba de condiciones se centra en evaluar las condiciones lógicas dentro de las estructuras de control (como las sentencias if, switch, etc.) para asegurar que todos los posibles resultados de esas condiciones se verifiquen.

Objetivo: Validar que cada condición dentro de las expresiones lógicas se evalúe tanto como verdadero (true) como falso (false) durante las pruebas.

Métodos:

- **Pruebas de condiciones simples**: Se evalúa una única condición.
- **Pruebas de condiciones compuestas**: Se evalúan múltiples condiciones combinadas (por ejemplo, con operadores lógicos como AND y OR).

Importancia: Garantiza que las decisiones lógicas del código se ejecuten correctamente, ayudando a prevenir errores en el flujo de control del programa.

1.3 Prueba de bucles

Definición: La prueba de bucles se enfoca en evaluar el comportamiento de los bucles (como for, while, y do-while) en el código para asegurarse de que se ejecuten correctamente bajo diversas condiciones.

Objetivo: Verificar que los bucles se ejecuten el número correcto de veces y que manejen adecuadamente los casos de bucles vacíos, de una sola iteración y de múltiples iteraciones.

Tipos:

- **Pruebas de bucle infinito**: Asegurarse de que no se produzcan bucles que no terminen.
- **Pruebas de límite**: Probar los límites de las condiciones que controlan el bucle para garantizar que se manejan correctamente.

Importancia: Ayuda a evitar errores como bucles que no terminan o que se ejecutan menos veces de lo esperado, lo que puede llevar a comportamientos inesperados en el software.

2. Pruebas de caja negra

Las pruebas de caja negra son técnicas de prueba que se centran en evaluar el comportamiento del software a partir de sus especificaciones, sin tener en cuenta la estructura interna del código. Veamos tres tipos específicos de pruebas de caja negra:

2.1 Pruebas de clases de equivalencia de datos

Definición: Esta técnica implica dividir el conjunto de datos de entrada en clases o categorías que se espera que produzcan resultados similares. La idea es que si un caso de prueba de una clase funciona correctamente, los otros casos de la misma clase también lo harán.

Objetivo: Reducir el número de pruebas necesarias al identificar conjuntos de datos que se comportan de manera equivalente. Esto permite probar menos casos al mismo tiempo que se asegura una buena cobertura.

Ejemplo: Si un campo de entrada acepta valores numéricos entre 1 y 100:

- Clases válidas:
    - Un valor dentro del rango (por ejemplo, 50).
- Clases inválidas:
    - Un valor por debajo del rango (por ejemplo, 0).
    - Un valor por encima del rango (por ejemplo, 150).

Se pueden realizar pruebas usando un valor de cada clase, en lugar de probar cada número en el rango.

2.2 Prueba de valores límite

Definición: Esta técnica se basa en probar los valores en los límites de las clases de equivalencia, así como justo por encima y por debajo de esos límites. Se considera que los errores son más probables en los bordes de las entradas válidas.

Objetivo: Asegurar que el software maneje correctamente los límites establecidos en las especificaciones. Esta técnica se centra en los puntos críticos donde los errores suelen ocurrir.

Ejemplo: Siguiendo el ejemplo anterior del rango de 1 a 100:

- Valores a probar:
    - Justo por debajo del límite inferior: 0 (inválido).
    - En el límite inferior: 1 (válido).
    - En el límite superior: 100 (válido).
    - Justo por encima del límite superior: 101 (inválido).

2.3 Prueba de interfaces

Definición: Esta técnica evalúa la interacción entre diferentes componentes del software o entre el software y otros sistemas externos. Se centra en cómo los datos fluyen a través de las interfaces y cómo se manejan las entradas y salidas.

Objetivo: Garantizar que las interfaces funcionen correctamente, permitiendo que los componentes se comuniquen sin errores y que los datos se transmitan y reciban de manera adecuada.

Ejemplo: Si hay una interfaz de usuario que se comunica con una API:

- Probar que al enviar datos válidos desde la interfaz, la API devuelve la respuesta esperada.
- Verificar cómo la interfaz maneja respuestas erróneas de la API, como errores de conexión o datos incorrectos.

3. Dobles de prueba

Los **dobles de prueba** son objetos o componentes que imitan el comportamiento de partes del sistema que se están probando. Se utilizan en el contexto de pruebas de software para facilitar la verificación y validación del código, especialmente cuando ciertas dependencias o componentes del sistema son difíciles de manejar, costosos de configurar o lentos en ejecución.

Los dobles de prueba permiten a los desarrolladores y probadores simular el comportamiento de componentes externos, de modo que se pueda centrar la prueba en la unidad de código que se está evaluando.

3.1 Tipos de dobles de pruebas

Existen varios tipos de dobles de prueba, cada uno con propósitos específicos. Los más comunes son:

1. **Mocks**:
    - **Definición**: Son objetos simulados que se utilizan para verificar interacciones específicas entre el código bajo prueba y sus dependencias. Los mocks permiten definir expectativas sobre cómo se deben llamar a los métodos de las dependencias, incluyendo los parámetros que se deben pasar y la cantidad de veces que se deben invocar.
    - **Uso**: Se utilizan en pruebas donde es importante asegurarse de que el código bajo prueba se comunique correctamente con otros componentes. Si las expectativas no se cumplen durante la prueba, se produce un fallo.
2. **Stubs**:
    - **Definición**: Son implementaciones simplificadas de un componente que devuelven datos predefinidos. No verifican interacciones ni cuentan con expectativas; simplemente devuelven valores fijos o comportamientos sencillos.
    - **Uso**: Se utilizan cuando se necesita simular el comportamiento de una dependencia, pero no se requiere verificar cómo se ha llamado a esa dependencia.
3. **Fakes**:
    - **Definición**: Son implementaciones que tienen una lógica más compleja que los stubs, pero que todavía son simplificadas en comparación con las implementaciones reales. Los fakes tienen un comportamiento real, pero son más ligeros y fáciles de manejar que la implementación original.
    - **Uso**: Se utilizan para probar el código en un entorno controlado, donde la lógica de la dependencia es necesaria, pero no se requiere la complejidad de la implementación completa.
4. **Spies**:
    - **Definición**: Son similares a los mocks, pero se centran en registrar información sobre cómo se ha llamado a los métodos en lugar de definir expectativas. Los spies permiten observar interacciones sin establecer requisitos previos.
    - **Uso**: Se utilizan para comprobar si ciertos métodos han sido llamados y con qué argumentos, lo que es útil en pruebas de interacción.

5. **Dummies**:
    - **Definición**: Son objetos que se pasan como argumentos a métodos o funciones, pero que no se utilizan realmente. Son útiles para cumplir con las cabeceras de métodos cuando no se necesita un comportamiento real.
    - **Uso**: Se utilizan principalmente para completar el conjunto de parámetros requeridos en las pruebas sin afectar la lógica del código.

4. Herramientas de depuración de código

Advertido un problema durante la fase de pruebas del software, a menudo, resulta difícil identificar el error que lo causa. Las **herramientas de depuración de código** son aplicaciones o utilidades que ayudan a los desarrolladores a **identificar, diagnosticar y corregir errores** en el código fuente de un programa. Estas herramientas permiten analizar el comportamiento del software en tiempo de ejecución y son esenciales para entender cómo se ejecuta el código línea por línea, lo que facilita la detección de fallos lógicos y problemas de rendimiento o seguridad.

4.1 Funcionalidades principales

1. **Ejecución paso a paso**: Permite ejecutar el código línea por línea para observar en detalle el flujo del programa y el cambio de estado en cada paso.
2. **Puntos de interrupción (breakpoints)**: Permite pausar la ejecución del programa en puntos específicos. Esto ayuda a observar el estado del programa en momentos clave y a revisar el valor de las variables en un contexto particular.
3. **Inspección de variables y estructuras de datos**: Permite visualizar el valor de variables, objetos, arreglos y otras estructuras de datos en tiempo real para verificar si contienen los valores esperados.
4. **Llamadas a funciones (stack trace)**: Muestra el historial de llamadas de funciones o métodos, ayudando a entender el flujo de ejecución y a identificar dónde ocurrió un error específico.
5. **Evaluación de expresiones**: Permite evaluar expresiones y ejecutar comandos específicos durante la depuración para probar cómo cambiarían los valores o el flujo del programa.
6. **Perfilado de rendimiento**: Algunas herramientas ofrecen métricas sobre el tiempo de ejecución y consumo de recursos, lo que permite detectar cuellos de botella.

4.2 Funcionamiento

Las herramientas de depuración funcionan a través de un **control preciso del flujo de ejecución** del programa. Mediante el uso de breakpoints y la ejecución paso a paso, estas herramientas intervienen en el programa para pausarlo y acceder a información detallada en tiempo real. Por lo general, las herramientas de depuración están integradas en los entornos de desarrollo (IDE) o se pueden utilizar como software independiente, y se comunican con el programa mediante el uso de una interfaz de depuración que interactúa con el sistema de ejecución o el compilador para manipular el flujo y extraer datos.