Técnicas de pruebas

Temas de la clase: análisis del caso de estudio 1 (planteado en la clase anterior). Técnicas de pruebas. Estrategias de pruebas. Caja blanca. Caja negra.

Análisis y solución del caso de estudio 1

Para esto se dedicarán las primeras dos horas de esta clase

Ver la solución propuesta en el documento “soluc_caso_estudio1”.

Prueba del Software

Una estrategia de prueba del software integra los métodos de diseño de casos de pruebas en una serie de pasos bien planeada que desembocará en la eficaz construcción de la aplicación.

Cualquier estrategia de prueba debe incorporar la planeación de pruebas, el diseño de casos de pruebas, la ejecución de pruebas y la recolección y evaluación de los datos resultantes.

La prueba es un conjunto de actividades que se planean con anticipación y se realizan de manera sistemática. Por lo tanto, se debe definir un conjunto de pasos en que se puedan incluir técnicas y métodos específicos del diseño de casos de prueba.

Técnicas de pruebas (testing)

No debemos confundir pruebas (testing) con depuración (debugging).

Para simplificar, podríamos decir que la depuración, es lo que hacemos cuando sabemos que un programa produjo un error (“dio un error”).

Podríamos decir que la prueba es un intento sistemático de “romper” un programa que pensamos que está funcionando bien.

Un buen comienzo sería pensar en problemas potenciales mientras escribimos nuestro código. Tanto las pruebas sencillas como las más elaboradas, ayudan a asegurar que los programas nacen funcionando correctamente y permanecen así a medida que van creciendo.

El esfuerzo de probar un programa mientras lo vamos haciendo es muy redituable. Pensar en las pruebas mientras escribimos un programa llevará a tener un mejor código, porque ese es el momento en que mejor sabemos lo que el programa debería hacer.

Si en lugar de hacer esto esperamos a que aparezca alguna falla ("que algo se rompiera en la ejecución"), en ese momento ya podríamos habernos olvidado de cómo funcionaba el programa. Cuando trabajamos bajo presión (y nos dicen: "el problema del programa debería estar corregido ya…"), necesitaremos comprenderlo todo de nuevo, lo que lleva tiempo, y las modificaciones ("fixes") van a ser menos pensadas y más frágiles porque nuestra nueva comprensión del programa podría ser incompleta.

Es importante probar un programa sistemáticamente. De ese modo podemos saber en cada paso qué estamos probando y qué resultados esperamos. Deberíamos ser ordenados para no pasar por alto nada importante.

El proceso de prueba debería ir mano a mano con la construcción del programa. La idea de escribir un programa completo y luego probarlo todo de una vez es más difícil y lleva más tiempo que ir probando incrementalmente. Se debería escribir una parte y probarla, luego agregar más código y probar nuevamente y así sucesivamente.

Si tenemos más de un componente, deberíamos probarlos por separado, pero luego en conjunto para ver que así también funcionen.

El concepto de prueba incremental también se aplica a cómo probamos características del programa. La prueba debería enfocarse primero a las partes más simples y más frecuentemente utilizadas. Solo una vez que estas funcionen correctamente, deberíamos pasar a probar otros componentes. De este modo, en cada nivel hay más para probar y tenemos más confianza en que los mecanismos básicos están funcionando bien.

Si bien cada falla ("bug") es más difícil de ser encontrada que su antecesora, no necesariamente es más difícil de corregir.

Debemos saber el resultado que se espera del programa. Para cada prueba necesitamos saber cuál es la respuesta correcta. Si no lo supiéramos estaríamos perdiendo nuestro tiempo. Esto puede parecer obvio, ya que para la mayoría de los programas podemos decir si el programa está funcionando o no. Por ejemplo un programa que copia un archivo, lo copió o no. El resultado que muestra un informe, aparece como se deseaba o no.

Pero algunos programas son más difíciles de caracterizar. Un algoritmo numérico (¿la salida tiene tolerancia de error?), un cálculo complejo (¿funciona para valores extremos?), la generación de gráficos (¿están los pixeles en el lugar correcto?), un encriptador (¿se podrá desencriptar lo que habíamos encriptado?).

De todos modos, podemos decir que la regla más importante de las pruebas es: **HACERLAS**.

Algunas consideraciones

Probar mientras se escribe el código: cuanto más temprano se encuentra un problema es mejor. Si pensamos sistemáticamente en el código que estamos escribiendo mientras lo hacemos, podemos verificar características del programa mientras el mismo está siendo construido. Así, nuestro código ya "habría tenido una prueba" aún antes de ser compilado por primera vez.

Programar "a la defensiva": una técnica útil es agregar código para manejar las situaciones "que no pueden ocurrir". Son situaciones que no son lógicamente posibles de ocurrir, pero podrían ocurrir igual debido a una falla en otro lado (otro componente cuyo mal funcionamiento perjudique al que estamos haciendo). En muchas ocasiones los programadores suelen decir "...para qué preveer esto si nunca va a pasar…".

El componente que nosotros desarrollamos, puede recibir como entrada lo que devuelve otro componente, desarrollado por otra gente, que entrega el resultado de una manera que no esperábamos.

En esto, la programación de un sistema mediano, se parece un poco a la conducción de un vehículo. Podemos conducir (programar) bien, pero otros que transitan con nosotros manejan (programan) mal. Entonces debemos tomar previsiones.

Probar el código en sus límites: la mayoría de los errores ocurren en los límites (de un loop, de un array completo, etc.)

Verificar los errores retornados: se deben controlar los códigos de error devueltos por llamadas a funciones, operaciones de entrada/salida, etc. Una consulta y una actualización pueden fallar y hacer que las operaciones siguientes cancelen la ejecución.

Características de las pruebas

- Las pruebas solo pueden mostrar la presencia de defectos, no su ausencia

- Las pruebas serán exitosas si detectan errores

- Las pruebas siempre deben realizarse contra un resultado esperado y los resultados obtenidos se deben revisar minuciosamente

- La prueba finalizará cuando el proceso establece confianza en que el componente o sistema hace lo que se supone tiene que hacer, ya que nunca se podrá demostrar que un componente o sistema no tiene defectos.

Nunca dejarán de encontrarse nuevos defectos. Pero llegará un momento en que el costo de encontrar nuevos defectos es mayor que el beneficio de corregirlos. Ese es el punto en el que debe detenerse la prueba.

Los distintos tipos de pruebas pueden ser efectuadas por los desarrolladores, por los "testers" y por los usuarios.

Un "tester" es la persona cuyo rol es verificar especificaciones y documentos respecto a cumplimiento de estándares, completitud y corrección, y /o validar productos respecto al cumplimiento de los requerimientos funcionales y no funcionales.

El tester está activamente involucrado en el proceso de desarrollo. Revisa especificaciones, prepara los planes de prueba, y administra el proceso de seguimiento de defectos.

Tipos y estrategias de prueba

Fase	Tarea	Tipo de prueba	Estrategia de Prueba	Ejecutor
Diseño y desarrollo	Prueba unitaria	Unitaria	Caja negra Caja blanca	Desarrollador
Diseño y desarrollo	Prueba periódica	Funcional Regresión Recupero Seguridad	Normalmente caja negra	Tester
Diseño y desarrollo	Prueba de integración	Funcional Stress Volumen Performance	Caja negra	Tester

		Regresión Recupero Seguridad Interfases con otros sistemas Conversión de datos		
Aceptación	Prueba de aceptación	Aceptación del usuario Ambiente e instalación Stress Volumen Performance Recupero Seguridad Interfases con otros sistemas Conversión de datos	Caja negra	Usuario Operaciones Auditoría Seguridad

Estrategia de caja negra

Las pruebas de caja negra se desentienden completamente del comportamiento y estructura interna del programa. También son llamadas pruebas conducidas por los datos o pruebas conducidas por las entradas/salidas.

Las condiciones y casos de prueba tendrán variaciones considerando, según corresponda:

- Partición de equivalencias: debido a que nunca se podrán probar todos los valores posibles de un atributo o campo, se seleccionan conjuntos de valores típicos, tales que cubran un extenso abanico de casos de prueba. Estas pruebas son llamadas por "partición de equivalencias" o "subdominios". La prueba se realiza tomando, por ejemplo, un valor válido y dos inválidos del mismo tipo que la clase válida para cada subconjunto.

- Condiciones de borde: se ingresan valores para los casos extremos, y casos inválidos para los valores siguientes a los extremos.

- Otros tipos de datos: se prueban ingresos de otros tipos de datos (números en lugar de caracteres alfabéticos, fechas en lugar de números, etc.). Muchas veces dichas validaciones son resueltas por la herramienta utilizada para el desarrollo.

- Condiciones cruzadas: por ejemplo, no se pueden cargar los datos del cónyuge si la persona no está casada. La estructura del modelo de datos define también estas reglas cruzadas que deben ser probadas.

- Conjetura de errores (sospechas): se crean casos de prueba con mayor cantidad de variaciones, para componentes considerados de riesgo, debido a complejidad, circunstancias del desarrollo, programas modificados por varias personas, programa armado pegando pedazos de otros, etc.

- Transición-estado, o causa-efecto: se crean casos de prueba que ejerciten todas las combinaciones de acciones / cambios de estado causados, para los diagramas de transición-estado que correspondan a los ciclos de vida de las entidades de interés.

Estrategia de caja blanca

Los casos de prueba se definen a partir del conocimiento de la estructura interna del programa. También son llamadas pruebas estructurales.

Para analizar la estructura del componente a probar, se representa el flujo de control del mismo a través de un diagrama de flujo. Se determina el conjunto básico de caminos independientes, y se preparan los casos de pruebas que forzarán la ejecución de cada camino.

Debería usarse esta estrategia para componentes críticos y/o sospechosos.

Las condiciones y casos de prueba tendrán variaciones considerando:

- Cobertura de sentencias: cada línea de código debe ser ejecutada al menos una vez.

- Cobertura de decisión: para cada decisión debe proveerse, al menos, un resultado verdadero y uno falso.

- Cobertura de condición: para cada condición en una decisión deben proveerse valores que ejerciten todos los casos posibles, al menos una vez.

- Cobertura de decisión / condición: combinación de las anteriores, todos los caminos posibles e independientes, según salidas de condiciones y decisiones, eliminando los casos redundantes o no posibles.

Prueba Unitaria

Se realiza sobre una unidad de código claramente definida, es llevada a cabo por los desarrolladores. Requiere un trabajo complejo y laborioso.

Prueba Funcional

Prueba que verifica los requerimientos funcionales del sistema. Comprueba que las partes de un sistema que funcionan bien aisladamente en la prueba unitaria, también lo hacen en conjunto.

Prueba de Stress

Prueba del sistema excediendo los límites de su capacidad de procesamiento y de almacenamiento, teniendo en cuenta situaciones no previstas originalmente. Se utiliza cuando hay poco control sobre los usuarios u otros sistemas que interactúan con el nuestro, o para componentes críticos en cuanto al nivel de servicio.

Las entradas de datos generados automáticamente exigen a un programa de modo diferente que los datos ingresados por un usuario. Un gran volumen por si mismo, tiende a romper las cosas porque se rebalsan los buffers, los arreglos, los contadores, y son efectivos encontrando almacenamientos de tamaño fijo no probados en un programa.

Los programadores tienden a evitar casos imposibles como entradas vacías, o entradas fuera de rango, y tampoco crean nombres muy largos o valores numéricos muy altos. En contraste, un programa generador de datos de prueba no tiene idea de qué debería evitar.

Pruebas de Volumen y Performance

Tienen como objetivo verificar que el sistema soporta los volúmenes máximos definidos en la cuantificación de requerimientos para la capacidad de procesamiento y la capacidad de almacenamiento. Se debe analizar si es o no necesario realizar estas pruebas, ya que suelen ser costosas (por el consumo de recursos y la generación de datos de prueba) y no siempre son relevantes.

Pruebas de Regresión

Pruebas orientadas a verificar que, luego de introducido un cambio en el código, la funcionalidad original no ha sido alterada. Se debe probar que el defecto reportado no persista y que el resto del sistema continúe funcionando correctamente. Para realizar estas pruebas se requiere de condiciones y casos de prueba reusables.

Cuando corregimos un problema, hay una tendencia natural a probar solo que los arreglos funcionen. Es fácil descuidar la posibilidad de que una corrección altere otra funcionalidad que no tenía problemas (se arregla algo y el arreglo desarregla otra cosa).

Prueba de Recupero

Pruebas orientadas a verificar que el sistema se recupera correctamente ante situaciones que afectan o suspenden su ejecución (estos casos fueron tratados en la clase anterior cuando se habló de “reenganche de procesos”).

Pruebas de Aceptación del Usuario

Pruebas realizadas por los usuarios, para verificar que el sistema se ajusta a sus requerimientos y es fácil de usar. Estas pruebas, que podrían ser consideradas como “no fundamentales”, pueden causar el éxito o el fracaso de una implementación.

Aunque hayamos hecho un buen relevamiento para ver cómo debería funcionar la aplicación, la misma puede ser eficaz en cuanto a que cumple con los procesos que debía realizar, pero su uso podría resultar incómodo o complicado para los usuarios.

Pruebas de Ambiente e Instalación

El sector Operaciones deberá probar los requerimientos no funcionales y la instalación del sistema para las distintas configuraciones.

Pruebas de Seguridad

Los sectores Seguridad y Auditoría deben verificar que se cumplan los criterios de seguridad establecidos, tanto en el ámbito de la aplicación como en el ámbito de los datos.

Inspecciones de Código

Consisten en revisar un programa o porciones selectas del mismo, con el objetivo de encontrar defectos. Las inspecciones actúan de manera estática, no se ejecuta el código.

Dichas inspecciones deben realizarse antes de las pruebas y permiten encontrar defectos que posiblemente no sean detectados con las pruebas, debido a un enfoque mucho más eficiente.

Los resultados de las inspecciones forman la base para la prevención de futuros defectos. La práctica indica que, dependiendo de varios factores, se identifican entre 7 y 20 defectos por cada 1000 líneas no comentadas de código. Dichas inspecciones son solo para beneficio de los programadores, y no para ser usadas como evaluación de los mismos.

Condiciones de prueba

Son descripciones de situaciones que reflejan qué es lo que se quiere probar del sistema. La elaboración de condiciones de prueba es un proceso creativo llevado a cabo por el tester.

- Condiciones de prueba de caja negra: para cada uno de los componentes que requieren pruebas de caja negra, definir las condiciones que deben ser probadas en función de las entradas que reciben, salidas esperadas, y funciones que realizan, modificando condiciones existentes o agregando nuevas condiciones.

- Condiciones de prueba de caja blanca: para aquellos componentes en los que se vayan a realizar pruebas de caja blanca, identificar los caminos independientes de las secciones del código que van a ser probadas con esta técnica.

- Condiciones de prueba que verifiquen la funcionalidad: a partir de los casos de uso y sus alternativas, establecer aquellas situaciones que van a ser probadas para verificar que el sistema cumple con los requerimientos planteados por los usuarios.

- Condiciones de prueba a partir del modelo de datos: la cardinalidad de las relaciones define las reglas del negocio que deben ser probadas.

- Condiciones de prueba para Stress, Volumen, Performance y Recupero: definir qué pruebas se van a realizar para verificar que el sistema cumple con los requerimientos de capacidad de procesamiento y almacenamiento (requerimientos no funcionales). Definir qué pruebas se realizarán para ver la forma en la que el sistema se comporta ante situaciones no previstas originalmente, y qué pruebas se van a hacer para verificar que los mecanismos de recuperación definidos funcionan correctamente ante situaciones de falla.

Casos de prueba, datos de prueba y resultados esperados

Para cada una de las condiciones de prueba definidas, se establece un conjunto de variaciones de interés para la prueba, y se definen los datos que harán que se cumpla con esa condición / variación. Se define el resultado esperado de cada prueba en forma explícita y completa (resultado de cálculos, mensajes a ser emitidos, navegación, etc.)

La generación de los datos de prueba puede requerir que sean pobladas tablas, archivos, bases de datos enteras, y/o que se utilicen herramientas o aplicaciones existentes, o bien que deban construirse nuevas herramientas especialmente para la prueba en cuestión.

Automatización de las pruebas (test automation)

Probar manualmente un programa es muy tedioso y a veces poco redituable. Un testing apropiado involucra cantidad de pruebas, cantidades de lotes de datos de entrada y muchas comparaciones de resultados.

Es valioso dedicar un tiempo a generar un script (guión a seguir) o un programa simple que comprenda una prueba.

La forma más básica de automatización de las pruebas son los tests de regresión.

Los mismos ejecutan una secuencia de pruebas que comparan la nueva versión de un programa con la anterior. La intención es comprobar que el comportamiento no ha cambiado excepto en las formas esperadas.