UNIVERSDAD NACIONAL MAYOR DE SAN MARCOS
FACULTAD DE INGENIERÍA DE SISTEMAS E INFORMÁTICA

CURSO:
Métodos Formales de Pruebas - Testing de Software

Tema:
Fundamentos de Pruebas de Software
Semana 3

Docente: Mg. Wilder Inga

Testing Software

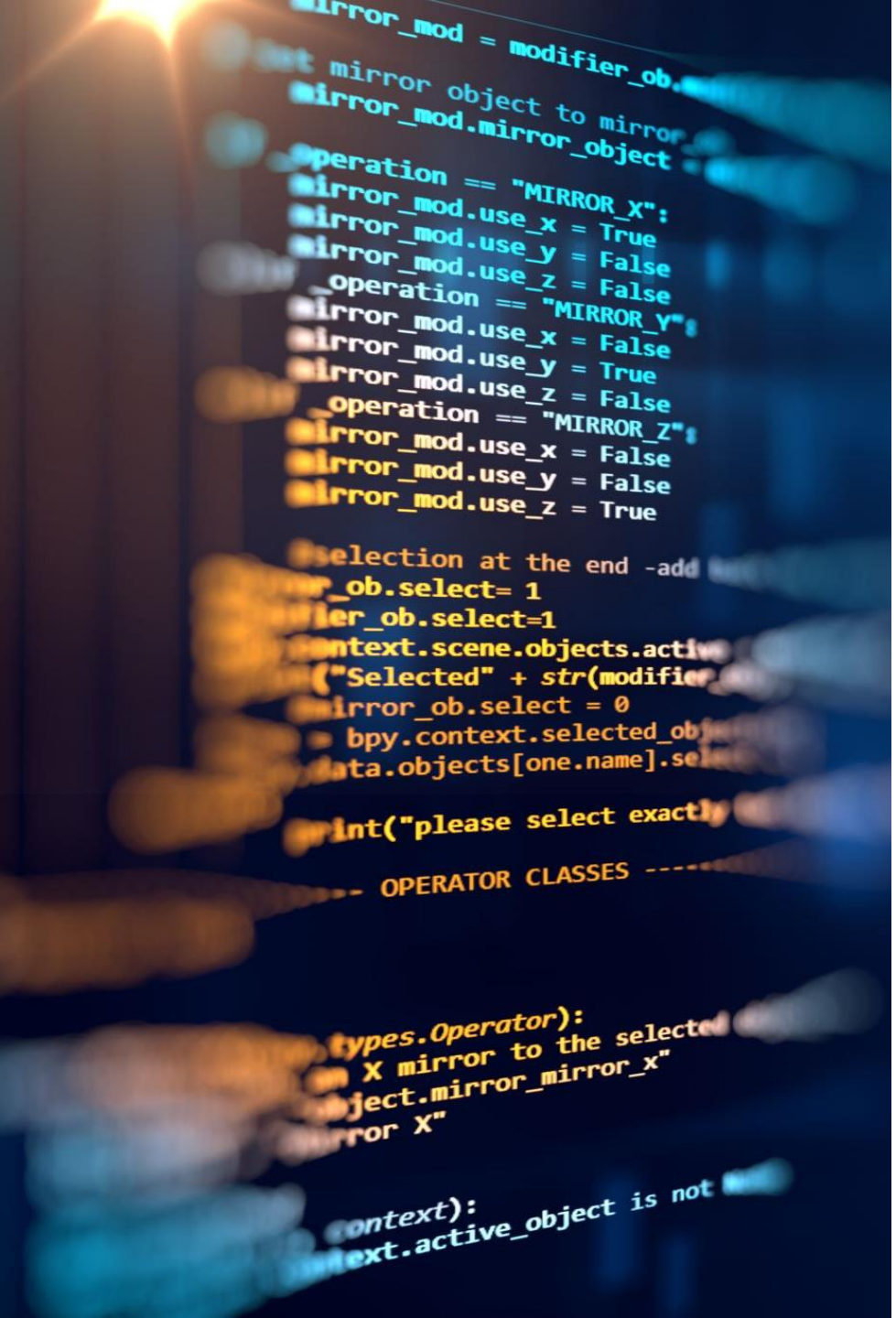
Las compañías de software se enfrentan con serios desafíos cada vez más grandes en la prueba de sus productos debido a que cada día **se incrementa la complejidad del software.**

Lo primero y más importante que hay que hacer es reconocer la naturaleza compleja de las pruebas y tomarlas en serio: contratar a las personas más inteligentes que se pueda encontrar, ayudarlas a conseguir y/o brindarles las herramientas y el entrenamiento que requieran para aprender su oficio, y escucharlos cuando hablan acerca de la **calidad del software**. Ignorarlos podría ser el error más costoso para la empresa y el producto.

Testing Software

Los investigadores en pruebas se enfrentan igualmente a estos desafíos. Las compañías están ansiosas por financiar buenas ideas de investigación, pero la demanda fuerte es por más práctica experimental y menos trabajo académico.

Las cuatro fases que estructuran la metodología propuesta en este capítulo no se deben considerar como definitivas y únicas, porque este es un campo en constante desarrollo e investigación en el que están involucrados muchos investigadores y empresas.



EL PROCESOS DE PRUEBA

- Al planificar y ejecutar las pruebas los probadores de software deben considerar: el software y su función de cálculo, las entradas y cómo se pueden combinar, y el entorno en el que el software eventualmente funcionará.
- Este difícil proceso requiere tiempo, sofisticación técnica y una adecuada planificación. Los probadores no solo deben tener buenas habilidades de desarrollo (a menudo las pruebas requieren una amplia cantidad de código), sino también conocimientos en lenguajes formales, teoría de grafos, lógica computacional y algoritmia.

EL PROCESOS DE PRUEBA

Para tener una visión más clara acerca de las dificultades inherentes a las pruebas, es necesario acercarse a ellas a través de la aplicación de cuatro fases:

1. Modelar el entorno del software
2. Seleccionar escenarios de prueba
3. Ejecutar y evaluar los escenarios
4. Medir el progreso de las pruebas

1: Modelar el entorno del software

La tarea del probador es simular la interacción entre el software y su entorno para lo que debe identificar y simular las interfaces que utiliza el sistema. Las interfaces más comunes son:

- Humanas: Interfaz gráfica de usuario GUI, clics, pulsaciones de teclado y entradas desde otros dispositivos.
- De software: APIs, WS, sistema operativo, la base de datos o las librerías, en tiempo de ejecución. Comprobar, no solo las probables, sino también las inesperadas. Por ejemplo, todos los desarrolladores esperan que el sistema operativo guarde los archivos por ellos, pero olvidan que les puede informar que el medio de almacenamiento está lleno, por lo que, incluso, los mensajes de error deben probarse.
- Del sistema de archivos, datos, formato, contenido.
- Las interfaces de comunicación, acceso directo a dispositivos físicos, controladores (IoT), protocolos válidos e inválidos

1: Modelar el entorno del software

Cada entorno único de aplicación puede resultar en un número significativo de interacciones de usuario que se debe probar.

Situaciones que los probadores deben abordar son:

- Usando el sistema operativo un usuario elimina un archivo que otro usuario tenía abierto, ¿qué pasará la próxima vez que el software intente accesar ese archivo?
- Un dispositivo se reinicia en medio de un proceso de comunicación, ¿podrá el software darse cuenta de esto y reaccionar adecuadamente, o simplemente lo dejará pasar?
- Dos sistemas compiten por duplicar servicios desde la API, ¿podrá la API atender correctamente ambos servicios?

1: Modelar el entorno del software

Cuando una interfaz presenta problemas de tamaño o de complejidad infinitos:

- Se suele usar la técnica Boundary Value Partitioning, para seleccionar valores individuales para las variables, en o alrededor de sus fronteras. Por ejemplo, probar los valores máximos, mínimos y cero para un entero con signo es una prueba común, lo mismo que los valores que rodean cada una de estas particiones, por ejemplo, 1 y -1 que rodean la frontera cero. Los valores entre las fronteras se tratan como el mismo número: utilizar 16 o 16.000 no hace ninguna diferencia para el software bajo prueba.

elegir los valores para múltiples variables procesadas simultáneamente y que potencialmente podrían afectar a otras

- Al decidir cómo será la secuencia de entrada, los probadores tienen un problema de generación de secuencia, por lo que deben tratar cada entrada física y cada evento abstracto como símbolos en el alfabeto de un lenguaje formal, definir un modelo de ese lenguaje que les permita visualizar el posible conjunto de pruebas, e indagar cómo se ajusta cada una a la prueba general.

2: Seleccionar escenarios de prueba

Muchos modelos de dominio y particiones de variables representan un número infinito de escenarios de prueba, cada uno de los cuales cuesta tiempo y dinero

Solo un sub-conjunto de ellos se puede aplicar en cualquier programa de desarrollo de software realista, así que ¿cómo hace un probador inteligente para seleccionar ese sub-conjunto? ¿17 es mejor valor de entrada que 34? ¿cuántas veces se debe seleccionar un filename antes de pulsar el botón Open?

los probadores prefieren una respuesta que se refiera a la cobertura de código fuente, o a su dominio de entrada, y se orientan por la cobertura de las declaraciones de código (ejecutar cada línea de código fuente por lo menos una vez), o la cobertura de entradas (aplicar cada evento generado externamente).

En cuanto al código, no son las declaraciones individuales las que interesan a los probadores, sino los caminos de ejecución: secuencias de declaraciones de código que representan un camino de ejecución del software, pero, desafortunadamente, existe un número infinito de caminos.

2: Seleccionar escenarios de prueba

Las pruebas se organizan desde dichos conjuntos infinitos hasta lograr, lo mejor posible, los criterios adecuados de datos de prueba; que se utilizan adecuada y económicamente para representar cualquiera de esos conjuntos

Por ejemplo las pruebas se organizan desde dichos conjuntos infinitos hasta lograr, lo mejor posible, los criterios adecuados de datos de prueba; que se utilizan adecuada y económicamente para representar cualquiera de esos conjuntos

2: Seleccionar escenarios de prueba

Criterios de prueba de los caminos de ejecución

Los criterios adecuados para datos de prueba se concentran en la cobertura de caminos de ejecución o en la cobertura de secuencias de entrada, pero rara vez en ambos. El criterio de selección de caminos de ejecución más común es el de aquellos que cubran las estructuras de control. Por ejemplo: 1) seleccionar un conjunto de casos de prueba que garantice que cada sentencia se ejecute al menos una vez, y 2) seleccionar un conjunto de casos de prueba que garantice que cada estructura de control If, Case, While, … se evalúe en cada uno de sus posibles caminos de ejecución.

2: Seleccionar escenarios de prueba

Criterios de prueba del dominio de entrada

El criterio para el rango de cobertura del dominio abarca desde la cobertura de una interfaz sencilla hasta la medición estadística más compleja:

- Elegir un conjunto de casos de prueba que contenga cada entrada física.
- Seleccionar un conjunto de casos de prueba que garantice que se recorra cada interfaz de control.

2: Seleccionar escenarios de prueba

Criterios de prueba del dominio de entrada

El criterio de discriminación requiere una selección aleatoria de secuencias de entrada hasta que, estadísticamente, representen todo el dominio infinito de las entradas:

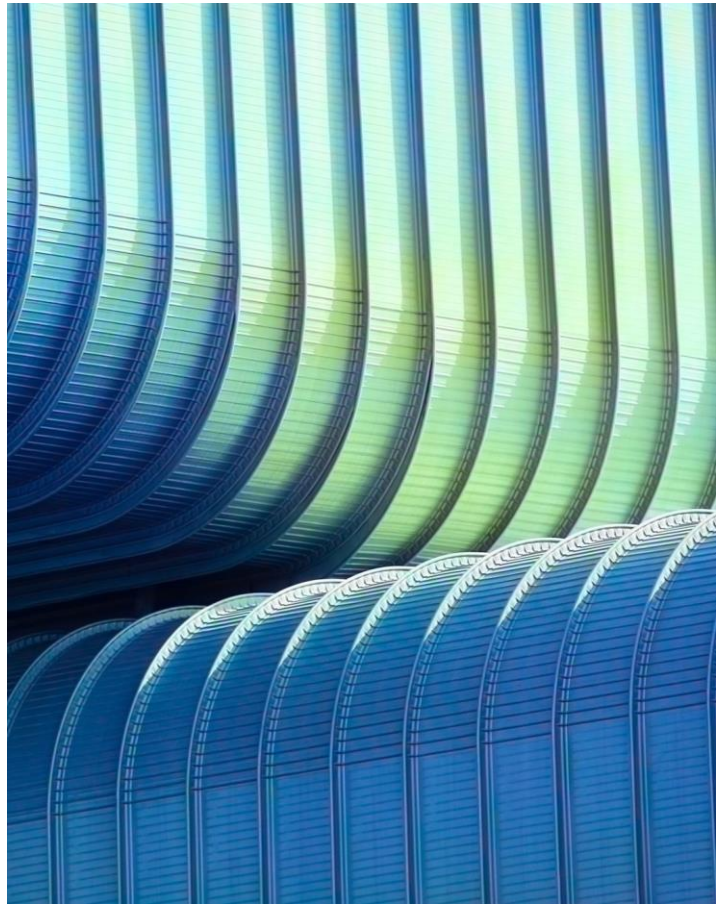
- Seleccionar un conjunto de casos de prueba que tenga las mismas propiedades estadísticas que el dominio de entrada completo.
- Escoger un conjunto de rutas que puedan ser ejecutadas por un usuario típico.

3: Ejecutar y evaluar los escenarios

Debido a que los escenarios de prueba se ejecutan manualmente constituye un trabajo intensivo y, por tanto, propenso a errores, por lo que los probadores deben tratar de automatizarlos tanto como sea posible.

En muchos entornos es posible aplicar automáticamente las entradas a través del código que simula la acción de los usuarios, y existe herramientas que ayudan a este objetivo. Pero la automatización completa requiere la simulación de cada fuente de entrada, y del destino de la salida de todo el entorno operacional.

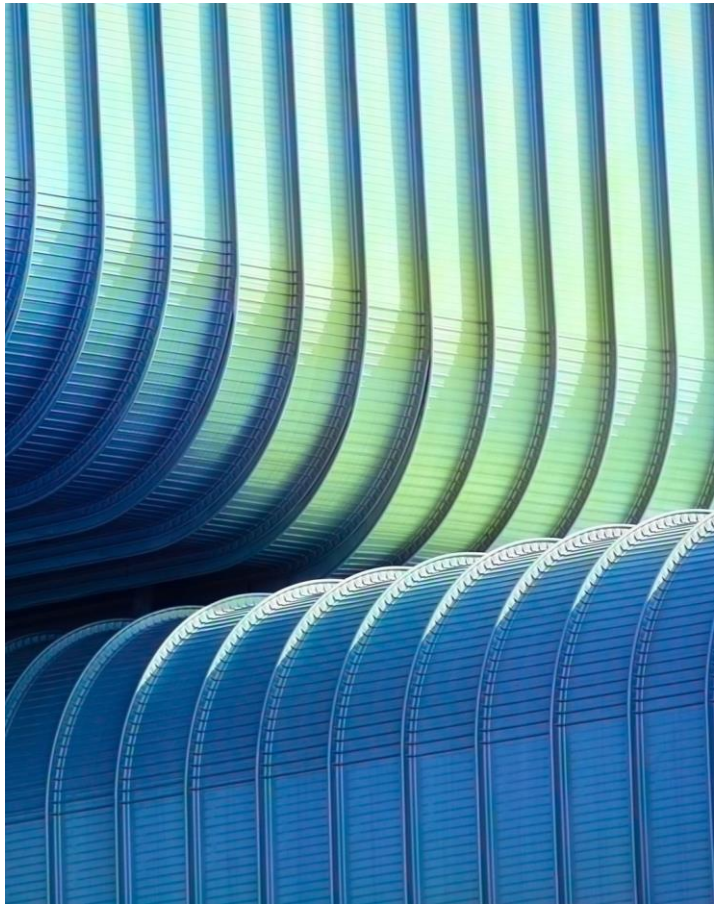
A menudo, los probadores incluyen código para recoger datos en el entorno simulado, como ganchos o seguros de prueba, con los que recogen información acerca de las variables internas, las propiedades del objeto, y otros. Esto ayuda a identificar anomalías y a aislar errores. Estos ganchos se eliminan cuando el software se entrega.



3: Ejecutar y evaluar los escenarios

La evaluación de escenarios, la segunda parte de esta fase, es fácil de fijar, pero difícil de ejecutar porque es menos automatizada. La evaluación implica la comparación de las salidas reales del software, resultado de la ejecución de los escenarios de prueba, con las salidas esperadas, tal y como están documentadas en la especificación, que se supone correcta, ya que las desviaciones son errores.

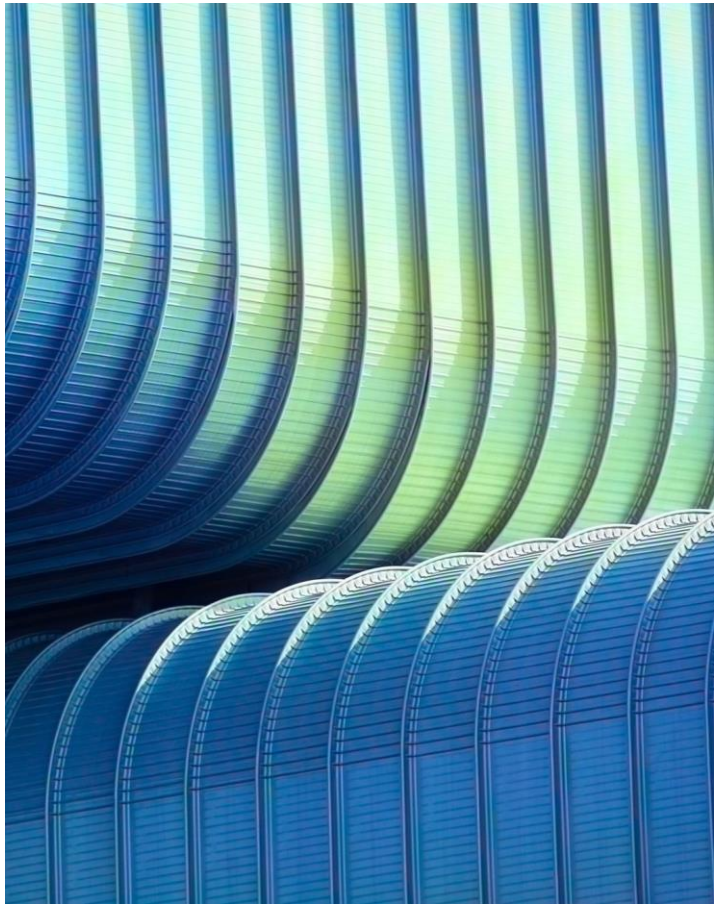
Teóricamente, la comparación (para determinar la equivalencia) de dos funciones arbitrarias computables es irresoluble. Volviendo al ejemplo del editor de texto: si la salida se supone que es resaltar una palabra mal escrita, ¿cómo se puede determinar que se ha detectado cada instancia de errores ortográficos? Tal dificultad es la razón por la que la comparación de la salida real versus la esperada se realiza generalmente por un oráculo humano: un probador que monitorea visualmente la pantalla de salida y analiza cuidadosamente los datos que aparecen.



3: Ejecutar y evaluar los escenarios

La evaluación de escenarios, la segunda parte de esta fase, es fácil de fijar, pero difícil de ejecutar porque es menos automatizada. La evaluación implica la comparación de las salidas reales del software, resultado de la ejecución de los escenarios de prueba, con las salidas esperadas, tal y como están documentadas en la especificación, que se supone correcta, ya que las desviaciones son errores.

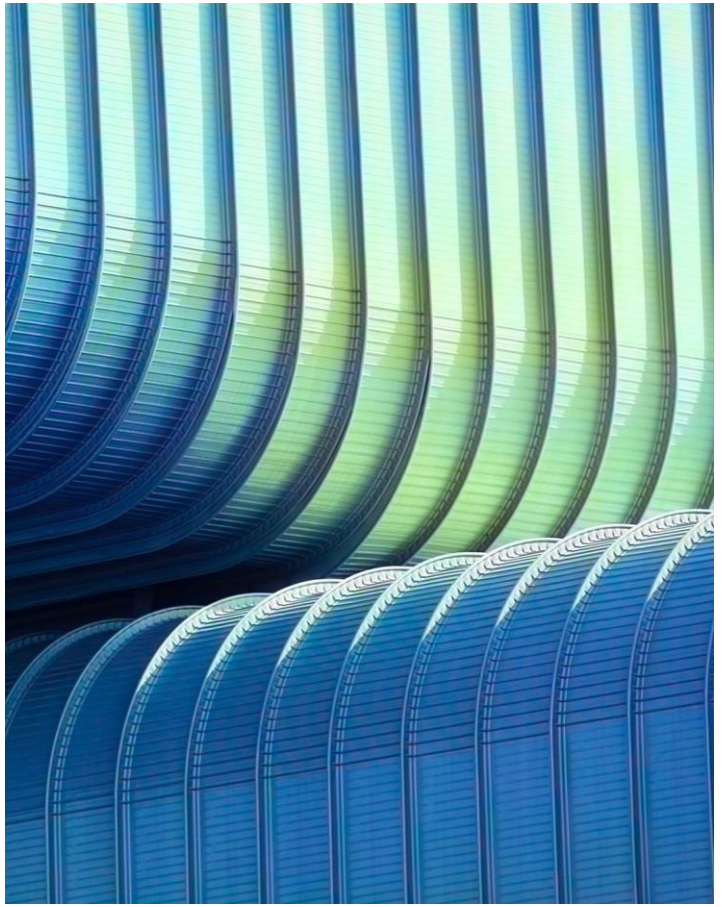
Teóricamente, la comparación (para determinar la equivalencia) de dos funciones arbitrarias computables es irresoluble. Volviendo al ejemplo del editor de texto: si la salida se supone que es resaltar una palabra mal escrita, ¿cómo se puede determinar que se ha detectado cada instancia de errores ortográficos? Tal dificultad es la razón por la que la comparación de la salida real versus la esperada se realiza generalmente por un oráculo humano: un probador que monitorea visualmente la pantalla de salida y analiza cuidadosamente los datos que aparecen.



3: Ejecutar y evaluar los escenarios

La evaluación de escenarios, la segunda parte de esta fase, es fácil de fijar, pero difícil de ejecutar porque es menos automatizada. La evaluación implica la comparación de las salidas reales del software, resultado de la ejecución de los escenarios de prueba, con las salidas esperadas, tal y como están documentadas en la especificación, que se supone correcta, ya que las desviaciones son errores.

Teóricamente, la comparación (para determinar la equivalencia) de dos funciones arbitrarias computables es irresoluble. Volviendo al ejemplo del editor de texto: si la salida se supone que es resaltar una palabra mal escrita, ¿cómo se puede determinar que se ha detectado cada instancia de errores ortográficos? Tal dificultad es la razón por la que la comparación de la salida real versus la esperada se realiza generalmente por un oráculo humano: un probador que monitorea visualmente la pantalla de salida y analiza cuidadosamente los datos que aparecen.



3: Ejecutar y evaluar los escenarios

Enfoques para evaluar las pruebas

- La formalización consiste en formalizar el proceso de escritura de las especificaciones y la forma cómo, desde ellas, se derivan el diseño y el código. Tanto el desarrollo orientado por objetos, como el estructurado, tienen mecanismos para expresar formalmente las especificaciones, de forma que se simplifique la tarea de comparar el comportamiento real y el esperado.
- Esencialmente, existe dos tipos de código de prueba embebido:
    1. El más simple es el código de prueba que expone algunos objetos de datos internos, o estados, de tal forma que un oráculo externo pueda juzgar su correctitud más fácilmente. Al implementarse, dicha funcionalidad es invisible para los usuarios. Los probadores pueden tener acceso a resultados del código de prueba a través de, por ejemplo, una prueba al API o a un depurador.
    2. Un tipo más complejo de código embebido tiene características de programa de auto-prueba: a veces se trata de soluciones de codificación múltiple para el problema, para chequear o escribir rutinas inversas que deshacen cada operación. Si se realiza una operación y luego se deshace, el estado resultante debe ser equivalente al estado pre-operacional.

3: Ejecutar y evaluar los escenarios

Pruebas de regresión

Después de que los probadores presentan con éxito los errores encontrados, generalmente los desarrolladores crean una nueva versión del software, en la que, supuestamente esos errores se han eliminado. La prueba progresa a través de versiones posteriores del software hasta una que se selecciona para entregar. La pregunta es: ¿cuántas re-pruebas, llamadas pruebas de regresión, se necesitan en la versión n, cuando se re-utilizan las pruebas ejecutadas sobre la versión n-1?

3: Ejecutar y evaluar los escenarios

Pruebas de regresión

Cualquier selección puede:

1) corregir solo el problema que fue reportado,
2) fallar al corregir el problema reportado,
3) corregir el problema reportado, pero interrumpir un proceso que antes trabajaba, o
4) fallar al corregir el problema reportado e interrumpir un proceso funcional.

Teniendo en cuenta estas posibilidades sería prudente volver a ejecutar todas las pruebas de la versión n-1 en la versión n, antes de probar nuevamente, aunque esta práctica generalmente tiene un costo elevado.

Las nuevas versiones del software a menudo vienen con características y funcionalidades nuevas, además de los errores corregidos, así que las pruebas de regresión les quitarían tiempo a las pruebas del código nuevo.

3: Ejecutar y evaluar los escenarios

Adicionalmente:

Se debería codificar de tal forma que pueda verificarse y validarse adecuadamente.

Dado que los errores los identifican los probadores, pero los diagnostican los desarrolladores, puede suceder:

1) Que su reproducción fracase, o
2) Que el escenario de prueba se ejecute nuevamente.

Volver a efectuar una prueba no garantiza que se reproduzcan las mismas condiciones originales. Re-ejecutar un escenario requiere conocer con exactitud el estado del sistema operativo y cualquier software que lo acompañe, condiciones del entorno, etc.

Hay que conocer el estado de automatización de la prueba, los dispositivos periféricos y cualquiera otra aplicación de segundo plano, que se ejecute localmente o través de la red y que podría afectar la aplicación bajo prueba.

4: Medir el progreso de las pruebas

¿cuál es el estado de sus pruebas?

- en la práctica, medir las pruebas consiste en contar cosas: el número de entradas aplicadas, el porcentaje de código cubierto, el número de veces que se ha invocado la aplicación, el número de veces que se ha terminado la aplicación con éxito, el número de errores encontrados, y así sucesivamente.
- La interpretación de estas cuentas es difícil: ¿encontrar un montón de errores es buena o mala noticia? ¿un alto número de errores significa que la prueba se ejecutó completamente y que persisten muy pocos errores?; o ¿el software tiene un montón de errores y que, a pesar de que muchos fueron encontrados, otros permanecen ocultos?.

4: Medir el progreso de las pruebas

Los valores de estos conteos pueden dar muy pocas luces acerca de los avances de las pruebas, y muchos probadores alteran estos datos para dar respuesta a las preguntas, con lo que determinan la completitud estructural y funcional de lo que han hecho. Por ejemplo, para comprobar la completitud estructural los probadores pueden hacerse preguntas como:

- ¿He probado para errores de programación común?
- ¿He ejercitado todo el código fuente?
- ¿He forzado a todos los datos internos a ser inicializados y utilizados?
- ¿He encontrado todos los errores sembrados?

Y para probar la completitud funcional:

- ¿He tenido en cuenta todas las formas en las que el software puede fallar y he seleccionado casos de prueba que las muestren y casos que no lo hagan?
- ¿He aplicado todas las posibles entradas?
- ¿Tengo completamente explorado el dominio de los estados del software?
- ¿He ejecutado todos los escenarios que espero que un usuario ejecute?

4: Medir el progreso de las pruebas

Capacidad de prueba

la idea de que el número de líneas de código determine la dificultad de la prueba es obsoleta, la cuestión es mucho más complicada y es donde entra en juego la capacidad de prueba. Si un producto software tiene alta capacidad de prueba:

1) Será más fácil de probar y, por consiguiente, más fácil de encontrar sus errores, y
2) Será posible monitorear las pruebas, por lo que los errores y disminuyen las probabilidades de que se queden otros sin descubrir.

Una baja capacidad de prueba requerirá muchas más pruebas para llegar a estas mismas conclusiones, y es de esperar que sea más difícil encontrar errores.

4: Medir el progreso de las pruebas

Modelos de fiabilidad

¿Cuánto durará el software en ejecución antes de que falle? ¿Cuánto costará el mantenimiento del software? Sin duda es mejor encontrar respuestas a estas preguntas mientras todavía se tenga el software en el laboratorio de pruebas. Desde un punto de vista funcional los modelos de fiabilidad (modelos matemáticos de escenarios de prueba y datos de errores) están bien establecidos. Con base en cómo se comportó durante las pruebas, estos modelos pretenden predecir cómo se comportará el software en su entorno funcional. Para lograrlo la mayoría de ellos requieren la especificación de un perfil operativo: una descripción de cómo se espera que los usuarios apliquen las entradas.

4: Medir el progreso de las pruebas

Modelos de fiabilidad

Para calcular la probabilidad de una falla estos modelos hacen algunas suposiciones acerca de la distribución de probabilidades subyacentes que regulan las ocurrencias de las anomalías. Tanto los investigadores como los probadores expresan escepticismo acerca de que se pueda ensamblar adecuadamente estos perfiles.

Errores que llegan al usuario

Los desarrolladores conocen la frustración cuando reciben reportes de errores de parte de los usuarios. Cuando esto sucede inevitablemente se preguntan: ¿cómo escaparon esos errores a las pruebas?

Se invirtieron incontables horas en el examen cuidadoso de cientos o miles de variables y sentencias de código, ¿cómo puede un error eludir esta vigilancia?

1. **El usuario ejecuta un código no probado**. Por falta de tiempo, no es raro que los desarrolladores liberen código sin probar, en el que los usuarios pueden encontrar anomalías.
2. **El orden en que se ejecuta las declaraciones en el ambiente de uso difiere del que se utilizó durante la prueba**. Este orden puede determinar si el software funciona bien o no.

Errores que llegan al usuario

3. **El usuario aplica una combinación de valores de entrada no probados**. Las posibles combinaciones de valores de entrada, que miles de usuarios pueden hacer a través de una interfaz de software, simplemente son numerosas para que los probadores las apliquen todas y, como deben tomar decisiones difíciles acerca de qué valores de entrada probar, a veces toman las equivocadas.
4. **El entorno operativo de usuario nunca se probó**. Es posible que los probadores tengan conocimiento de dicho entorno, pero que no cuenten con el tiempo suficiente para probarlo. Tal vez no replicaron la combinación de hardware, periféricos, sistemas operativos y aplicaciones del entorno del usuario en el laboratorio de pruebas. Por ejemplo, aunque es poco probable que las empresas que escriben software creen redes de miles de nodos en su laboratorio de pruebas, los usuarios sí lo hacen y, de hecho, lo hacen en sus entornos reales.

Actividad