Prueba del software: más que una fase en el ciclo de vida

Software Testing: More than a Stage in the Life Cycle

Edgar Serna M.[(1)], **Fernando Arango I.**[(2)]
[(1)] M.Sc. Universidad de San Buenaventura Medellín, edgar.serna@usbmed.edu.co
[(2)] PhD. Universidad Nacional de Colombia sede Medellín, farango@unal.edu.co

Recibido 31 de agosto de 2010, modificado 18 de enero de 2012, aprobado 18 de enero de 2012.

Palabras claves
Caminos de ejecución, capacidad de prueba, escenario de prueba, flujo de datos, prueba del software, verificación.

Keywords
Data flow, execution paths, software testing, testability, test scenario, verification.

Resumen
La prueba de software es probablemente la parte menos comprendida del ciclo de vida del desarrollo de software. En este trabajo, mediante una propuesta metodológica de cuatro fases, se muestra por qué es difícil detectar y eliminar errores, por qué es complejo el proceso de realizar pruebas y por qué es necesario prestarle más atención.

Abstract
Software testing probably is the least understood part of the software testing life cycle. In this work, by means of a methodological proposal of four stages, is showed why is complex the process of carrying out the testing software, why is necessary to pay it more attention and why is so difficult to detect and delete the mistakes.

Introducción

Toda empresa que desarrolla software prueba sus productos pero, aun así, antes de entregarlos siempre contienen anomalías residuales de diversa gravedad. A veces, es difícil imaginar cómo es que los probadores no detectan algunos errores evidentes. En muchas empresas, los probadores están mal preparados para ejecutar la difícil tarea de ensayar los productos software, cada vez más complejos. Los resultados en muchas encuestas informales, hechas a los asistentes a seminarios, sugieren que algunos de los que realizan pruebas, como profesión o como un complemento al desarrollo, tienen un adecuado entrenamiento en pruebas o tienen acceso a buenos libros de pruebas de software.

James Whittaker [1] brinda algunas luces acerca de por qué el proceso de probar el software actual es tan retador e identifica varios enfoques concretos que todos los probadores deberían ser capaces de aplicar fácilmente: el probador eficiente tiene a su disposición un amplio conjunto de técnicas de prueba, entiende cómo se utilizará el producto en su entorno operativo, tiene buen olfato para encontrar errores sutiles y tiene a la mano una bolsa de trucos que sabe utilizar. Los métodos que se describen en este trabajo pueden ayudar a los probadores para dar una respuesta sensata a la cuestión de lo que realmente quieren expresar cuando dicen: “estamos ejecutando pruebas a un sistema software”.

Los desarrolladores conocen la frustración cuando se reciben reportes de errores de parte de los usuarios. Cuando esto sucede inevitablemente se preguntan: ¿cómo escaparon esos errores a las pruebas? Sin duda que invirtieron incontables horas en el examen cuidadoso de cientos o miles de variables y sentencias de código, así que ¿cómo puede un error eludir esta vigilancia? La respuesta requiere, en primer lugar, una mirada más atenta a las pruebas de software en el contexto de desarrollo y, en segundo lugar, es necesario comprender el papel que juegan los probadores y los desarrolladores en dicho contexto dos funciones parecidas pero muy diferentes.

Suponiendo que las anomalías que reportan los usuarios realmente son errores, la respuesta a la anterior pregunta podría ser cualquiera de las siguientes:

- 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.
- El orden en que se ejecutan 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.
- 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 demasiado 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.
- El entorno operativo del usuario nunca se probó. Es posible que los probadores tengan conocimiento de dicho entorno, pero 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 de red creen redes de miles de nodos en su laboratorio de pruebas, los usuarios sí lo pueden hacer y, de hecho, lo hacen en sus entornos reales.

Desde una visión general del problema y del proceso de las pruebas del software, este artículo investiga y describe los problemas que enfrentan los probadores e identifica las cuestiones técnicas que cualquier solución de prueba debe abordar. Además, estudia las clases de soluciones que actualmente se utilizan en la práctica.

Probadores y procesos de prueba

Los probadores de software deben considerar lo siguiente al planificar y ejecutar las pruebas: 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 sólo deben tener buenas habilidades de desarrollo —a menudo las pruebas requieren una gran cantidad de código— sino también conocimientos en lenguajes formales, teoría de grafos, lógica computacional y algoritmia. De hecho, los probadores creativos aplican muchas disciplinas, relacionadas con la informática, al problema de las pruebas, a menudo con resultados impresionantes.

Incluso el software más simple presenta obstáculos por lo que, para tener una visión más clara acerca de algunas de las dificultades inherentes a las pruebas de software, 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 prueba.

Estas fases le ofrecen a los probadores una estructura en la que pueden agrupar los problemas relacionados y que deben resolver antes de pasar a la siguiente fase.

Fase 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, y enumerar las entradas que pueden circular por cada una de ellas. Éste podría ser el asunto más importante que enfrentan y, teniendo en cuenta los diversos formatos de archivo, los protocolos de comunicación y las terceras partes disponibles —interfaces de programación de las aplicaciones— puede ser muy complicado. Las interfaces más comunes son:
- Las interfaces humanas, incluyen todos los métodos comunes con los que las personas se comunican con el software. La más destacada es la interfaz gráfica de usuario —*Graphical User Interface* GUI—, pero todavía se utilizan antiguos diseños como la interfaz de línea de comandos y la basada en menús. Los posibles mecanismos de entrada que se deben considerar son los clics del ratón, pulsaciones de teclado y entradas desde otros dispositivos. Los probadores deciden entonces cómo organizar estos datos para comprender cómo ensamblarlos en una prueba efectiva.
- Las interfaces de software, llamadas APIs —*Application Programming Interfaces*—, indican cómo utiliza el software al sistema operativo, la base de datos o la librería, en tiempo de ejecución. Los servicios que estas aplicaciones ofrecen se modelan como entradas de prueba, pero el desafío para los probadores es comprobar, no sólo las probables, sino también las inesperadas. Por ejemplo, todos los desarrolladores esperan que el sistema operativo guarde los archivos por ellos, pero olvidan que el sistema operativo les puede informar que el medio de almacenamiento está lleno por lo que, incluso, los mensajes de error deben probarse.
- Las interfaces del sistema de archivos, existen siempre que el software lea o escriba datos en archivos externos. Los desarrolladores deben escribir líneas de código de comprobación de errores para determinar si el archivo contiene datos y formato adecuados. Por lo tanto, los probadores deben construir o generar archivos con contenido, que a la vez sea legal e ilegal, y archivos que contengan texto y formatos variados.
- Las interfaces de comunicación, permiten el acceso directo a los dispositivos físicos como los controladores de dispositivos y otros sistemas embebidos y requieren un protocolo de comunicación específico. Para examinarlas, los probadores deben ser capaces de generar protocolos válidos e inválidos; además de poder ensamblar muchas y diferentes combinaciones de comandos y datos, para aplicarlos a la interfaz bajo prueba, en el formato del paquete apropiado.

Luego, los probadores deben comprender las interacciones de usuario que están fuera del control del software bajo prueba, ya que las consecuencias pueden ser graves si el software no está preparado. Ejemplos de 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?

Cada entorno único de aplicación puede resultar en un nú-

mero significativo de interacciones de usuario que se debe probar.

Para tener en cuenta

Cuando una interfaz presenta problemas de tamaño o de complejidad infinitos, los probadores se enfrentan a dos dificultades: 1) seleccionar cuidadosamente los valores para cualquier variable de entrada, y 2) decidir cuál será la secuencia de las entradas. En la selección de valores, deben determinar el de las variables individuales y asignar las combinaciones adecuadas cuando el programa acepta múltiples variables como entrada.

Frecuentemente utilizan la técnica *Boundary Value Partitioning* [2] para seleccionar valores individuales para las variables, en o alrededor de sus fronteras. Por ejemplo, probar los valores máximo, mínimo 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 ó 16.000 no hace ninguna diferencia para el software bajo prueba.

Una cuestión más compleja es elegir los valores para múltiples variables procesadas simultáneamente, que potencialmente podrían afectar a otras, y para lo que debe considerarse el producto que resulta de la combinación de valores completo. Por ejemplo, para dos números enteros: considerar ambos positivos, ambos negativos, uno positivo y uno cero, y así sucesivamente [3]. 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 y 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. El modelo más común es un grafo o diagrama de estados, aunque existen muchas variaciones: otros modelos populares incluyen expresiones regulares y gramaticales, herramientas de la teoría de lenguajes; menos utilizados son los modelos estocásticos de procesos y los algoritmos genéticos. Pero, en general, el modelo es una representación que describe cómo se combinan las entradas, y los símbolos de los eventos, para formar palabras y oraciones sintácticamente válidas.

Esas oraciones son secuencias de entrada que se pueden aplicar al software bajo prueba. Un ejemplo de esto es, considerar la entrada *Filemenu.Open*, que involucra una caja de dialogo para seleccionar archivos; *Filename*, que representa la selección —tal vez con clics del mouse— de un archivo existente, y *ClickOpen* y *ClickCancel*, que representan el botón accionado. La secuencia *Filemenu.Open Filename ClickOpen* es correcta, como muchas otras, pero la secuencia *ClickCancel Filemenu.Ope*n es incorrecta, ya que el botón de cancelación no se puede presionar hasta que la caja de diálogo se haya invocado. Un modelo en lenguaje formal puede hacer una distinción entre las secuencias a aplicar.

Podemos representar el uso correcto de la caja de diálogo para seleccionar archivos, por ejemplo, en un editor de texto, con la expresión: *Filemenu.Open filename* (ClickOpen|ClickCancel*, en la que el asterisco representa el operador de clausura de Kleene [4] y demuestra que la acción *filename* puede ocurrir cero o más veces. Esta expresión indica que la primera entrada recibida es *Filemenu.Open*, seguida de cero o más selecciones de un *filename* —una combinación de clics del ratón y entradas de teclado— y que, a continuación, se presiona el botón *Open* o *Cancel*. Este sencillo modelo representa todas las combinaciones de entrada que pueden suceder y si tienen sentido o no. Para completarlo, tendríamos que representar secuencias para las interfaces de usuario y del sistema operativo. Además, necesitaríamos una descripción de los archivos legales y corruptos para investigar a fondo la interacción del sistema de archivos, tarea que requiere usar ampliamente la lógica, la descomposición y la abstracción.

Fase 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. Sólo un subconjunto de ellos se puede aplicar en cualquier programa de desarrollo de software realista, así que ¿cómo hace un probador inteligente para seleccionar ese subconjunto? ¿17 es mejor valor de entrada que 34? ¿cuántas veces se debe seleccionar un *filename* antes de pulsar el botón *Open*? Estas cuestiones, que tienen muchas respuestas, actualmente se investigan activamente. Los probadores, sin embargo, 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—. Estos son los criterios mínimos que utilizan para juzgar la completitud de su trabajo, por lo tanto, el conjunto de casos de prueba que muchos eligen es el que cumpla con sus metas de cobertura.

Pero si el código y la cobertura de entrada son suficientes, los productos entregados deberían tener muy pocos errores. 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. En cuanto al dominio de entrada, no les interesan las individuales, sino las secuencias de entrada que, en su conjunto, representan escenarios a los que el software debe responder, pero también existe un número infinito de ellas.

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. “Mejor” y “adecuadamente” son subjetivos: los probadores típicamente buscan el conjunto que garantizará encontrar la mayoría

de los errores. Muchos usuarios y profesionales de aseguramiento de la calidad del software están interesados en que los probadores evalúen los escenarios de uso típicos —cosas que ocurren con mayor frecuencia en el uso del producto—. Probar esos escenarios puede asegurar que el software funciona de acuerdo con lo especificado, y que se han detectado los errores más frecuentes.

Para citar un caso, consideremos nuevamente el ejemplo del editor de texto: para probar el uso típico, nos centraremos en la edición y el formato, puesto que es lo que la mayoría de usuarios reales hace; no obstante, para encontrar errores, un lugar con mayor probabilidad son las características más difíciles de código, como el dibujo de figuras y la edición de tablas.

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.

Sin embargo, el flujo de control es sólo un aspecto del código fuente. Actualmente, ¿Qué software mueve datos de un lugar a otro? La familia de flujo de datos del criterio adecuado para datos de prueba describe la cobertura de estos datos [5], como el seleccionar un conjunto de casos de prueba que garantice que cada estructura de datos se inicialice y que posteriormente se utilice.

Por último, es interesante la siembra de errores [2], aunque tiene más atención de los investigadores que de los probadores. En este método, los errores se añaden intencionadamente en el código fuente, y se diseñan escenarios de prueba para encontrarlos. Lo ideal sería que al encontrar éstos, también se encontraran errores reales. Por lo tanto, es posible un criterio como el siguiente: seleccionar un conjunto de casos de prueba que exponga cada uno de los errores sembrados.

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 cada interfaz de control —ventanas, menús, botones, etc.— se recorra.

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.

Para resumir, los investigadores de pruebas estudian algoritmos para seleccionar conjuntos de prueba mínimos que cumplan los criterios para caminos de ejecución y dominios de entrada. La mayoría de ellos está de acuerdo en que es prudente utilizar varios criterios cuando se toman decisiones importantes para cada versión del producto. Los experimentos para comparar criterios adecuados para datos de prueba son necesarios, así como los nuevos criterios. No obstante, por ahora, los probadores deben estar conscientes de qué criterios integrar en su metodología y comprender las limitaciones inherentes de esos criterios cuando reportan resultados.

Fase 3: ejecutar y evaluar los escenarios

Una vez identificado el conjunto de casos de prueba adecuado, los probadores los convierten a formatos ejecutables, a menudo código, de modo que los escenarios de prueba resultantes simulen la acción de un usuario típico. 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 existen 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. De esta forma, ayuda a identificar anomalías y a aislar errores. Estos ganchos se eliminan cuando el software se entrega.

La evaluación de escenarios, la segunda parte de esta fase, es fácil de fijar pero difícil de ejecutar —es mucho 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. En la práctica, esta comparación es difícil de lograr. 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 faltas ortográficas? 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 cuidadosamente analiza los datos que aparecen.

Dos enfoques para evaluar las pruebas

Al tratar con el problema de la evaluación de la prueba los investigadores aplican dos enfoques: la formalización y el código de prueba embebido.

- La formalización consiste en "*formalizar*" el proceso de escritura de las especificaciones, y la forma como, desde ellas, se derivan el diseño y el código [6]. 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. La industria generalmente le ha rehuido a los métodos formales; no obstante, una buena especificación, aunque informal, sigue siendo de gran ayuda. Sin una especificación, posiblemente los probadores pueden encontrar sólo los errores más obvios. Por otra parte, la ausencia de una especificación redundaría en una pérdida significativa de tiempo cuando se reporten anomalías, no especificadas, como errores.
- Esencialmente existen 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, por ejemplo, de una prueba al API o a un depurador. 2) Un tipo más complejo de código embebido tiene características de programa de auto-prueba [7]: 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 del software resultante debe ser equivalente a su estado pre-operacional. En esta situación el oráculo no es perfecto: podría haber un error, en ambas operaciones, en el que cada error enmascara a otro.

Pruebas de regresión

Después 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*?

Cualquier selección puede: a) arreglar sólo el problema que fue reportado, b) fallar al arreglar el problema reportado, c) arreglar el problema reportado pero interrumpir un proceso que antes trabajaba, o d) fallar al arreglar 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 [8]. Por otra parte, las nuevas versiones del software a menudo vienen con características y funcionalidades nuevas, además de los errores corregidos, así que las pruebas de regresión le quitarían tiempo a las pruebas del código nuevo. Para ahorrar recursos, los probadores deben trabajar en estrecha colaboración con los desarrolladores para establecer prioridades, y reducir al mínimo las pruebas de regresión.

Otro inconveniente de estas pruebas es que pueden, temporalmente, modificar el criterio adecuado para datos de prueba seleccionado en la fase anterior. Cuando se realizan pruebas de regresión, los probadores sólo pretenden demostrar la ausencia de errores y forzar su aplicación a que exhiba un comportamiento específico. El resultado es que el criterio adecuado para datos de prueba, que hasta ahora guía la selección de los casos de prueba, es ignorado. Por lo que, en su lugar, los probadores deben asegurarse de que el código se haya corregido adecuadamente.

Asuntos relacionados

Los desarrolladores deberían, idealmente, escribir código teniendo en su mente las pruebas: si el código va a ser difícil de verificar y validar, entonces se debería reescribir, de tal forma que pueda verificarse y validarse adecuadamente. Del mismo modo, una metodología de prueba debería juzgarse de acuerdo con su contribución a la automatización y al oráculo de la solución de los problemas. Muchas metodologías propuestas ofrecen poca orientación en cualquier de estas áreas. Otra preocupación para los probadores, mientras ejecutan pruebas de verificación o validación, es cómo coordinar con los desarrolladores las actividades de depuración. Dado que los errores los identifican los probadores, pero los diagnostican los desarrolladores, puede suceder: 1) que su reproducción fracase y 2) que el escenario de prueba se ejecute nuevamente.

Que fracase la reproducción no es tan simple como parece, por lo que la respuesta obvia sería, por supuesto, volver a ejecutar la prueba fracasada y observar nuevamente el comportamiento de los resultados, aunque 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 —tal es el caso de las aplicaciones cliente-servidor que requieren la reproducción de las condiciones del entorno tanto en el cliente como en el servidor—. Además, 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 a la aplicación bajo prueba. No es de extrañar que una de las frases que comúnmente se escucha en los laboratorios de prueba es: "Bueno, se comporta de forma diferente antes de..."

Fase 4: medir el progreso de las pruebas

Supongamos que cualquier día, el jefe de un probador viene y le pregunta: "¿Cuál es el estado de sus pruebas?" Los probadores escuchan a menudo esta pregunta, pero no están bien preparados para responderla. La razón es que 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? La respuesta podría ser: un alto número de errores, que significa que la prueba se ejecutó completamente y que persisten muy pocos errores; o que simplemente que el software tiene un montón de errores y que, a pesar de que muchos fueron encontrados, muchos otros permanecen ocultos.

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 estas preguntas:

- ¿He probado para errores de programación común? [9]
- ¿He ejercitado todo el código fuente? [2]
- ¿He forzado a todos los datos internos a ser inicializados y utilizados? [5]
- ¿He encontrado todos los errores sembrados? [2]

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? [9]
- ¿He aplicado todas las posibles entradas? [2]
- ¿Tengo el dominio de los estados del software completamente explorado? [1]
- ¿He ejecutado todos los escenarios que espero que un usuario ejecute? [10]

Estas preguntas —fundamentalmente el criterio adecuado para datos de prueba— son útiles para los probadores, empero, determinar cuándo detener las pruebas o cuándo está listo un producto para su liberación, es más complejo. Se necesitan medidas cuantitativas de la cantidad de errores que queden en el software, y de la probabilidad de que cualquiera de ellos sea descubierto por el usuario. Si los probadores pudieran lograr esta medida, sabrían cuando parar las pruebas y sería posible que se acercaran cuantitativamente al problema de forma estructural y funcional.

Capacidad de prueba

Desde un punto de vista estructural, Jeffrey Voas [11] propuso la capacidad de prueba como una manera para determinar la complejidad de la aplicación de una 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 las probabilidades de que se queden otros sin descubrir, disminuyen. 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. La capacidad de prueba es un concepto convincente pero que apenas comienza a popularizarse, y todavía no se han publicado suficientes datos acerca de su capacidad predictiva.

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 [10] —modelos matemáticos de escenarios de prueba y datos de errores— están bien establecidos. Estos modelos pretenden predecir cómo se comportará el software en su entorno funcional, con base en cómo se comportó durante las pruebas. Para lograr esto, la mayoría de ellos requieren la especificación de un perfil operativo: una descripción de cómo se espera que los usuarios apliquen las entradas.

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 puedan ensamblar adecuadamente estos perfiles. Por otra parte, las hipótesis hechas por los modelos de fiabilidad aún no verifican teórica o experimentalmente, salvo en dominios de aplicación específicos. Sin embargo, estudios de caso exitosos demuestran que estos modelos pueden ser creíbles.

Conclusiones y trabajo futuro

- Las compañías de software se enfrentan con serios desafíos en la prueba de sus productos, cada vez más grandes, debido a que el software es cada vez más complejo.
- 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 que nunca se haya cometido.
- Los investigadores en pruebas se enfrentan igualmente a estos desafíos. Las compañías de software están ansiosas por financiar buenas ideas de investigación, pero la demanda fuerte es por más práctica experimental y menos trabajo

académico. El tiempo para concatenar la investigación académica con los productos industriales es ahora.

- Las cuatro fases que estructuran la metodología propuesta 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.

Referencias bibliográficas

[1] J. A. Whittaker & M. G. Thomason. "A Markov Chain Model for Statistical Software Testing". *IEEE Transactions on Software Engineering*, IEEE Trans. Software Eng., Vol. 20, No. 10, Oct. 1994, pp. 812-824.

[2] G. J. Myers. *The Art of Software Testing*. New York: John Wiley & Sons, 1976, pp. 59-65.

[3] T. J. Ostrand & M. J. Balcer. "The Category-Partition Technique for Specifying and Generating Functional Tests". *Communications of the ACM*. Comm. ACM, Vol. 31, No. 6, June 1988, pp. 676-686.

[4] Kozen, D. "On Hoare Logic, Kleene Algebra and Types". *Studies in Epistemology, Logic, Methodology and Philosophy of Science*, Vol. 315, Aug. 2002, pp. 119-133.

[5] S. Rapps & E. J. Weyuker. "Selecting Software Test Data Using Dataflow Information". *IEEE Transactions on Software Engineering*. IEEE Trans. Software Eng., Vol. 11, No. 4, Apr. 1985, pp. 367-375.

[6] D. K. Peters & D. L. Parnas. "Using Test Oracles Generated from Program Documentation". *IEEE Transactions on Software Engineering*. IEEE Trans. Software Eng., Vol. 24, No. 3, Mar. 1998, pp. 161-173.

[7] D. Knuth. "Literate Programming". *The Computer Journal*. The Comp. Jour., Vol. 27, No. 2, May 1984, pp. 97-111.

[8] G. Rothermel & M. J. Harrold. "A Safe, Efficient Algorithm for Regression Test Selection". *In Proceedings of the Conference on Software Maintenance ICSM '93*. Montreal, Canada, Sept. 1993, pp. 358-367.

[9] J. B. Goodenough & S. L. Gerhart. "Toward a Theory of Test Data Selection". *IEEE Transactions on Software Engineering*. IEEE Trans. Software Eng., Vol. 2, No. 2, June 1975, pp. 156-173.

[10] J. D. Musa. "Software Reliability Engineered Testing". *Computer*, Vol. 29, No. 11, Nov. 1996, pp. 61-68.

[11] J. M. Voas. "PIE: A Dynamic Failure-Based Technique". *IEEE Transactions on Software Engineering*. IEEE Trans. Software Eng., Vol. 18, No. 8, Aug. 1992, pp. 717-727.