Verificación (1)

Alejandro Teruel

2 de mayo 2013

Contenido: Error, defecto y falla. Cómo generar casos de prueba unitaria para la programación dirigida por casos de prueba.

Definiciones claves

Error (*error*)
Es una equivocación cometida por un desarrollador. Algunos ejemplos de errores son: un error de tipeo, una malinterpretación de un requerimiento o de la funcionalidad de un método. El estándar 829 de la IEEE coincide con la definición de diccionario de error como "una idea falsa o equivocada". Por ende un programa no puede tener o estar en un error, ya que los programas no tienen ideas; las ideas las tienen la gente.

Defecto (*fault, defect*)
Un error puede conducir a uno o más defectos. Un defecto se encuentra en un artefacto y puede definirse como una diferencia entre la versión correcta del artefacto y una versión incorrecta. De nuevo coincide con la definición de diccionario, "imperfección". Por ejemplo, un defecto es haber utilizado el operador "<" en vez de "<=".

Falla (*failure*)
En terminología IEEE, una falla es la discrepancia visible que se produce al ejecutar un programa con un defecto, respecto a la ejecución del programa correcto. Es decir, una falla es el *síntoma* de uno o más defectos.

Un ejemplo puede ayudar a aclarar la distinción entre estos términos[1]. Al conducir mi automóvil puedo darme cuenta que tiende a apagarse al detenerlo en los semáforos. Es decir, el automóvil me presenta una *falla*, la tendencia a apagarse a bajas revoluciones. Lo llevo al mecánico, quién busca el *defecto* que origina la falla; lo encuentra al darse cuenta que el carburador está mal entonado (el mínimo está muy bajo). El mecánico se disculpa, alegando que el *error* fue entonar el carburador de acuerdo a las características de un motor que no es el de mi automóvil, pensando que compartían el mismo punto de entonación.

Pruebas

> *Someter a prueba un producto, un servicio o un sistema consiste en revisarlo con miras a determinar bajo qué condiciones falla, caracterizar cómo se manifiesta y propaga la falla e identificar posibles defectos incluyendo como tales aquellos elementos que puedan propender a la comisión de errores.*

1 Le agradezco el ejemplo a la Profesora Amelia Soriano.

Someter a prueba un cronómetro de precisión puede involucrar diferentes pruebas para averiguar si está funcionando correctamente, dentro de qué rango de temperatura, presión, humedad y estado de su bateria funciona correctamente, qué ocurre cuando empieza a fallar (¿adelanta, retrasa, se para, titila el *display*?), cómo se propagan las fallas y si algunos de sus controles se presta a ser confundido o mal usado. Someter el cronómetro a prueba también puede incluir actividades de revisión o inspección del mismo, en funcionamiento o apagado, o de su diseño, con miras a encontrar defectos.

Análogamente se puede se pueden diseñar y ejecutar casos de prueba para averiguar si un software funciona correctamente, dentro de qué ambientes de ejecución (sistemas de operación, navegadores web, plataformas) y con qué carga (número de sesiones simultáneas, volumen de datos) funciona correctamente, qué ocurre cuando se presenta una falla (se cayó la red, se corrompieron los datos), como se propagan esas fallas y si algunas de las características del software tienden a confundir o a ser mal usados por los usuarios; también podemos *inspeccionar* el diseño o el código de un software y monitorizar la ejecución del software con miras a encontrar defectos.

La programación dirigida por casos de prueba: ¿Cómo generar los casos de prueba?

Comencemos por la construcción de los componentes más elementales que conformarán nuestro programa orientado por objetos: métodos y clases.

Obviemos momentáneamente los casos en que podemos aprovechar clases o métodos ya existentes y supongamos que nos hemos convencido de tener que programar una nueva clase y sus métodos.

Comenzaríamos por construir un caso de prueba que consiste en crear un objeto válido de la nueva clase. Como no hemos definido la clase y su constructor, la ejecución del caso falla -si no falla significa que ya existe en el alcance una clase con ese nombre.

Procedemos a programar el encabezado de la clase y su constructor. Corremos nuevamente el caso de prueba que debería pasar exitosamente.

Si ya tenemos una idea de cuál debe ser el invariante de la clase, el próximo caso de prueba a construir sería intentar construir un objeto que incumple con el invariante -paso necesario para cumplir con las exigencias de una programación segura. Corremos este nuevo caso y resulta que,erróneamente, se crea un objeto de la clase.

Agregamos alternativamente código, casos de prueba y refactorizaciones hasta que tenemos un constructor que sólo construye objetos que cumplan con la invariante de la clase y que lance una excepción en caso contrario. Tome en cuenta que en el estilo de *programación segura*, la clase no puede confiar en que el entorno se responsabilice de crear objetos que cumplan con el invariante.

El próximo paso es empezar a programar los demás métodos de la clase:

Casos de prueba para métodos

1. El primer caso de prueba puede ser tan simple como invocar el método a programar.

   El primer caso debe fallar, pues aún no se ha declarado el método. El caso prueba que, en el alcance de la invocación no exista un método predefinido con el mismo nombre. Codificamos lo suficiente para que se pase la prueba (el método con el cuerpo vacío).

2. Especificamos varios casos para el método y seguimos desarrollando el método aplicando la programación dirigida por casos de prueba.

¿Hay alguna recomendación que podamos seguir para generar los casos de prueba del paso 2?

Procesar valores extremos

Incluya casos extremos en los parámetros de entrada y en las variables de instanciación, p. ej. el mayor número entero representable y el menor número entero representable (¿uno es el negativo del otro?) (¿Qué valores debe considerar como extremos para los tipos punto flotante? ¿En cadenas de caracteres?)

Particionar los datos

Particione los datos de entrada en conjuntos tales que sea razonable que el procesamiento de un elemento del conjunto sea representativo del resto. Por ejemplo, un parámetro entero puede particionarse en el conjunto de números enteros negativos, el conjunto de números enteros positivos y el conjunto unitario que contenga cero.

La partición puede hacerse de acuerdo con la estructura inherente al tipo del parámetro, como hicimos con el parámetro entero o de acuerdo con la estructura de la precondición, postcondición e invariante de la clase... Por ejemplo una precondición tipo:

{pre: *P(x,y)* and *Q(x,y)*}

puede dar origen a cuatro particiones de x,y (recurde que x,y pueden ser atributos del objeto o parámetros de la llamada):

1. Se cumple *P(x,y)* y se cumple *Q(x,y)*
2. Se cumple *P(x,y)* y no se cumple *Q(x,y)*
3. No se cumple *P(x,y)* y se cumple *Q(x,y)*
4. No se cumple *P(x,y)* ni se cumple *Q(x,y)*

Si está siguiendo el estilo de programación segura, el código debe lanzar una excepción si no se cumple la precondición (casos 2,3,4)

En realidad para que el método se ejecute debe cumplir la pre-condición y la invariante de la clase a la entrada al cuerpo del método. Supongamos que la pre-condición es *P(x, a)* y la invariante de clase es *Q(a),* donde x es un parámetro y a es una variable de instanciación. En tal caso, dado que ya hemos programado de forma de garantizas que se cumpla la invariante al invocarse el constructor y que programaremos para cerciorarnos que se cumpla a la salida de las demás invocaciones el método, los únicos casos que vale la pena construir son:

- Se cumple *P(x,a)* y se cumple *Q(a)*
- No se cumple *P(x,a)* y se cumple *Q(a)*

Generar casos a partir de la partición de la salida es más difícil, pues debemos buscar datos para los parámetros de entrada y los atributos tales que la salida caiga en particiones distintas de la salida. Esencialmente esto puede ocurrir para una postcondición expresada en forma de disjunción pero no debería poder ocurrir para una postcondición expresada en forma de conjunción. Recuerde siempre

incluir pruebas para cerciorarse que la invariante de la clase se cumple a la salida del método.

¿Qué debe hacer si la complejidad del método es tal que usted como programador no está convencido que la invariante de clase se cumple al culminarse la ejecución del cuerpo del método?

Se le puede presentar la tentación de incluir un chequeo de la invariante a la salida del cuerpo del método que levante una excepción de incumplirse. Sin embargo, un programador ético no puede, ni debe, conformarse con dejar un método programado de manera que no esté seguro si cumple con su postcondición y la invariante de clase. Tiene que cerciorarse que lo haga, y si para lograrlo tiene que pedir ayuda, crear y aplicar más casos de prueba, rediseñar el método, la clase o el software completo, pues le corresponde hacerlo. Los proponentes más extremos de programación segura insistirían en hacer el chequeo de la postcondición y del invariante de clase, pero no para exonerar la responsabilidad del programador de entender su código, sino como salvaguarda por la posibilidad de una programación errada.

Fronteras

La práctica muestra que uno de los errores más frecuentes que comete el programador es "pelarse por uno": colocar un operador de "menor o igual" en vez de menor (o viceversa), olvidarse de sumar uno (o sumar uno de más). Estos errores son representativo de lo que podríamos considerar como errores en fijar los límites o fronteras entre una clase de equivalencia y otra. Por ende para detectar los defectos causados por este tipo de errores, es importante construir casos de prueba que prueben que la frontera esté bien puesta, que el valor que marca uno y otro lado de la frontera sean probados. Así si una frontera corresponde a $x<10$ (para x entero), es conveniente crear un caso de prueba que fuerce x a tener el valor 10 y otro que lo fuerce a tener el valor 9[2].

Note que en realidad, los casos extremos corresponden también a fronteras.

Combinaciones

Podemos diseñar los casos de prueba tal que cada uno de ellos pone a prueba un parámetro o una variable de instanciación de la clase a la vez. Por ejemplo podríamos crear un caso de prueba donde sólo uno de los parámetros esté en un valor extremo o de frontera. También podríamos crear casos de prueba que coloquen a varios parámetros simultáneamente en sus valores extremos.

La ventaja de tener una sola variable en un caso extremo es que se reduce el número de casos de prueba y que puede ubicar el defecto que conduce a cualquier falla más fácilmente. La deventaja es que, con esos casos, no detectaremos fallas que se deben a interrelaciones que se dan cuando varios elementos están en sus valores extremos.

Nuevamente un acoplamiento más débil facilita las pruebas: a menos parámetros más manejable es el diseño de casos de prueba que revisen combinaciones.

A veces puede ayudar a simplificar la situación la filosofía de detección de errores. Por ejemplo, algunos compiladores tratan de reportar todos los defectos que tiene el código que se pretende compilar, mientras que otros detienen el proceso de compilación al detectar el primer defecto. Desarrollar un compilador, y particularmente elaborar casos de prueba bajo la primera filosofía ("tratar de reportar el mayor número posible de defectos del código fuente") es mucho más complejo que

2 Una de las refactorizaciones recomendadas es la eliminación de *constantes mágicas.* Al refactorizar este código, el entero 10 debería ser sustituido por una constante que permita entender de dónde sale ese 10....

desarrollar un compilador y sus casos de prueba bajo la segunda filosofía.

Algunos desarrolladores advierten que es importante que un buen caso de prueba para un desarrollo dirigido por casos falle sólo -no es conveniente que fallen dos casos por la misma razón. Por ende sólo deben probarse dos parámetros en sus valores extremos si hay razones para sospechar que tal caso podría fracasar a pesar que se aprueben dos casos más simples, uno para cada parámetro en su caso extremo. Debemos aclarar que los criterios de construcción para los casos de prueba desarrollados específicamente para asegurar la calidad del software pueden ser diferentes que para programar el software[3].

Malicia

No es una técnica, sino una actitud relacionada con la habilidad de "pensar fuera de la caja". Pensar en particiones de datos, en sus fronteras puede reforzar ciertas suposiciones que tiene el programador sobre cómo se resuelve su tarea de programación, en las clases de equivalencia de los datos; ni la partición ni el pensar en fronteras ayuda a encontrar un ejemplo que muestre que una clase de equivalencia identificada por el programador son, a efectos de la programación, dos clases diferentes. No está de más dejarse guiar por su intuición, una sospecha o la malicia cuando esté haciendo casos de prueba. Lo que tienen que estar pendiente es de no caer en una actitud "paranoica" que lo lleve a proliferar, innecesariamente los casos de pruebas. Si ha caído en modo "paranoia", a lo mejor el problema es otro: en el fondo no comparte o no le resultan claros los requerimientos, no confía en el diseño o en sus habilidades como programador. La solución en este caso va más allá que desquitarse con más casos arbitrarios de prueba.

Casos de prueba para clases

Tenemos una clase y sus métodos han sido probados. ¿Terminamos?

Es recomendable que usted trate de identificar algunas secuencias de llamadas de métodos con propiedades especiales.... Por ejemplo, si está programando una clase Pila (*stack),* sería conveniente agregar casos de prueba para chequear si:

1. Inversas son inversas p. ej. *pop(push(s,x))* de como resulta *s*.
2. Cómo y cuando se presenta la falla cuando las secuencias de operaciones constructoras llegan a su límite (*push**, es decir empilar repetidamente hasta causar una excepción por falta de memoria).

Inspirándonos en la idea de *workflows* que menciona Scott Bain, podemos referirnos a este tipo de pruebas como pruebas de correctitud para el flujo de acciones.

¿Siempre conviene programar completamente clase por clase?

En desarrollos ágiles partimos de *casos de uso*, requerimientos dados por el cliente o usuario, no por el diseñador o el programador. El desarrollador diseña las clases que cree requerir para cumplir con esos casos de uso y procede a programarlos. Salvo para casos triviales, es improbable que una clase sea capaz de satisfacer un caso de uso. Es mucho más frecuente que el caso de uso lo resuelva una interacción entre objetos de ciertas clases. Si esto es así, ¿por qué no programar las clases sólo en la

3 Esta observación la hace Scott Bain en *Lean-Agile Engineering Practices*.www.netobjectives.com 2007. Si adoptamos esta idea, no deberíamos construir un caso de prueba previo a la programación para un valor extremo a menos que sepamos o sospechemos, con fundamento, que causará una falla.

medida requerida para satisfacer cada caso de uso?

La estrategia alterna planteada es válida también, pero puede resultar más estresante para el programador. La razón de ello, es que cada clase representa un "contexto" de alta cohesión, al saltar de la programación de (una porción) de una clase a (una porción) de otra, estamos cambiando ("suichando") de contexto. Y los cambios de contextos son cognitivamente demandantes, sobre todo si tal programación debe mantener la alta cohesión de contextos relativamente complejos. Por un lado, trabajar parcialmente sobre varias clases a la vez conlleva como riesgo perder de vista el foco, la cohesión que debe encapsular cada clase. Por otro lado, la ventaja de trabajar parcialmente cada clase es que se mantiene un importante foco de atención sobre el caso de uso.

La ventaja de trabajar completamente cada clase es que el foco se mantiene sobre la cohesión de la clase -el riesgo que se corre es perder de vista el requerimiento del caso de uso.

Si el número de clases que interviene en un caso de uso es bajo probablemente sea menos estresante la estrategia de desarrollo parcial que si se trata de un elevado número de clases.

Considere el caso de un maestro ajedrecista. Un maestro ajedrecista puede jugar una partida a la vez o puede jugar en tableros simultáneos, pero jamás se le podrá exigir el mismo nivel de juego en tableros simultáneos enfrentando múltiples oponentes que cuando enfrenta un sólo oponente...

Al final el asunto puede ser cosa de gustos y de circunstancias, como el uso del *zapping* para ver televisión. Si estamos buscando un programa que nos llame la atención, el *zapping* puede ser una buena estrategia, pero si estamos tratando de seguir varios programas en detalle con el *zapping*, inevitablemente algo dejaremos de ver o entender. Regresando al software, si hay una gran necesidad de completar un caso de uso para mostrárselo al cliente, programe parcialmente las clases, a sabiendas que probablemente tendrá que incurrir en un mayor esfuerzo de refactorización posteriormente; si las clases que están a un nivel de detalle muy distante del dominio del caso de uso o la cadena de interacción entre las distintas clases es muy larga, probablemente sea mejor cerrar más el desarrollo de cada clase.

Casos de prueba más allá de clases...

Los casos de prueba que hemos desarrollado hasta ahora son casos de *prueba unitaria ,* dado que consideremos a la clase y sus métodos la unidad o elemento de la programación orientada a objetos. Estas unidades las debemos ir integrando para formar agrupaciones de clases que formen componentes o subsistemas. En capítulos posteriores estudiaremos cómo ensamblar y someter a prueba tales integraciones.

Pero, una clase no es un sistema y un caso de prueba de un método o una clase no es un caso de uso

Nos hemos centrado en cómo se elaboran los casos de pruebas para programar un método o una clase en la programación dirigida por casos de prueba. A estas alturas, el lector más atento se estará pregunta ¿de dónde salieron las clases que tenemos que programar? La respuesta la encontraremos al revisar con más cuidado a las actividades de diseño, en un capítulo posterior.

¿Y si quiero aprovechar clase pre-existentes, debo someterlas a prueba?

En realidad lo primero que deberíamos hacer cuando enfrantamos una tarea de programación de una

clase es averiguar si existe una clase pre-existente que puedo aprovechar. En este momento no entraremos a explicar cómo hacer tal búsqueda. Podríamos encontrar una clase que podemos aprovechar tal cual está (por ejemplo una clase de una librería estándar de Java), podríamos encontrar una clase que requiere de ciertas modificaciones o podríamos no encontrar una clase aprovechable lo cual nos obliga a escribir una nueva clase.

Caso 1

Suponiendo que encontramos una clase que creemos que nos pueda servir tal cual está, podríamos decidir incluir algunos casos de prueba para cerciorarnos que entendemos bien cómo usarla. Podemos recurrir a más pruebas si no tenemos mucha confianza en la fuente donde obtuvimos la clase.

Caso 2

Si encontramos una clase que creemos que requiere de ciertas modificaciones, después de hacer los casos de prueba pertinentes, podríamos considerar crear una clase que herede de la encontrada. Otra opción (más compleja) es extraer una superclase de la conocida para "soslayar" lo que no aplica de la que encontramos -¡cuidado! y crear una subclase de ella (creando una clase "hermana" de la que encontramos.

En ambos casos es conveniente ejecutar casos de prueba (nuestros o pre-existentes) para mejorar nuestra comprensión de lo que pretendemos aprovechar.

Si confiamos en lo que estamos incorporando, conviene no hacer pruebas unitarias exhaustivas (aunque esto siempre tiene un riesgo asociado, también hay que recordar que ¡el tiempo corre!).

Un caso interesante se produce cuando te toca hacerle mantenimiento a un código existente que no fue desarrollado bjo la filosofía de programación dirigida por casos de prueba y peor, para el cuál no se desarrollaron, están incompletos o no se encuentran los casos de pruebas utilizados para su verificación. La primera reacción puede ser de parar cualquier programación hasta no someter exhaustivamente todo el código a prueba. Kent Beck (página 126, *Extreme Programming Explained*, Addison-Wesley, 2000) recomienda resistir tal tentación y más bien elaborar los casos de prueba por demanda, es decir, a medida que sean estrictamente necesarios y sólo para las partes del código que estés afectando :

- Cuando requieras agregar una funcionaldad a código no probado, escribe primero los casos de prueba para su funcionalidad actual;
- Cuando requieras corregir una falla o corregir un defecto, escribe un caso de prueba primero.
- Cuando necesites refactoriar, escribe casos de prueba primero.

Lecturas y enlaces recomendadas

[Falta actualizar esta sección]

- Roger Pressman: *Software Engineering: A Practitioner's Approach.* 7° Edición. McGraw-Hill, 2010. Ver capítulos 14, 17.
- Tim Riley, Adam Goucher: *Beautiful Testing: Leading professionals reveal how they improve software.* O´Reilly, 2009. Ver mi reseña del libro.
- William Perry: *Effective Methods for Software Testing.* Segunda edición, Wiley 2000. Revisar

los capítulos más generales.

[*Faltan revisar y actualizar estas* referencias adicionales *que datan del 2001]*