Ingenieria del Software II

Segundo Cuatrimestre de 2020

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Taller 3

Random Testing

Integrantes	LU	Correo electrónico
Tripodi, Guido	843/10	guido.tripodi@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Random Testing

Es una **técnica de prueba de software de caja negra** donde los programas se prueban generando entradas aleatorias e independientes. Los resultados de la salida se comparan con las especificaciones del software para verificar que la salida de la prueba se aprueba o falla. Esta estrategia a menudo genera un número significativo de casos de prueba en un corto período de tiempo, pero puede tener dificultades en cubrir partes de un programa de difícil acceso porque por ejemplo, para cubrirlo, se requieran valores específicos.

2. Conceptos

2.1. Code Coverage

Es una métrica relacionada a *Unit Testing*. La idea es, medir el porcentaje de líneas y caminos de ejecución en el código, que son cubiertas por al menos un caso de test.

2.2. Test Coverage

Es una técnica que determina si nuestros casos de test están **cubriendo** nuestro código y en qué medida lo están haciendo. En otras palabras, qué todo lo que tenga que ser testeado, esté siendo testeado.

2.3. Mutation Testing

Es una técnica que genera **automáticamente**, **mutaciones** o fallas de las funciones de tu código, para luego ejecutar tus tests. Si alguno de tus test falla, entonces el mutante es matado, si no, vive. La calidad de tu **TestSuite** dependerá del porcentaje de mutantes matados.

3. Herramientas Utilizadas

3.1. Randoop

Herramienta de generación de **casos de prueba** aleatorios, que ha demostrado tener una efectividad similar a la de las pruebas unitarias manuales en que respecta a la detección de errores. Muchas veces, los casos de prueba generados, detectan distintos errores que los encontrados por las pruebas unitarias manuales, demostrando que ambos enfoques para la generación de software confiable, son complementarios. Randoop puede encontrar una gran cantidad de defectos en cuestión de minutos.

3.2. JaCoCo

Es una herramienta que analiza la cobertura de código que alcanzan las pruebas que se tienen. Nos genera un reporte utilizando diferentes métricas, a partir del cual podemos analizar qué tan efectivo es nuestro **TestSuite** actual.

3.3. PiTest

PiTest es una herramienta que ejecuta tus **Unit Tests** sobre diferentes versiones de tu código, modificadas automáticamente (**mutation analysis**). Cuando nuestro código cambia, éste debe producir diferentes resultados y causará que los **Unit Tests** fallen. Si un test no falla, puede ser que haya un problema con tu **TestSuite**.

4. Explicación de los Resultados

5. Ejercicio 1

5.1. ¿Cuantas test cases produjo Randoop? ¿Hay failing test cases?

Randoop generó 386 tests. No se reportaron **failing test cases**, por lo tanto, todos los tests generados serán catalogados como **Regression Tests**.. El código parece cumplir el contrato de *equals()* y *hashCode()*, y como todavía no implementamos ningún otro contrato, **Randoop** no encuentra ningún error.

6. Ejercicio 2

6.1. ¿Cuántas líneas cubiertas reporta JaCoCo? ¿Cuántos branches cubiertos reporta JaCoCo?

JaCoCo reporta 46 de 53 lıneas cubiertas y 25 de 26 branches cubiertos. Puedo notar que principalmente las líneas no cubiertas ocurren en el método *hashCode()*. La explicación por la que **Randoop** no está generando tests de ese método es que el contrato de Java de *hashCode()* no lo 'obliga' a devolver un número en particular para una representación, sino que solamente debe cumplir que no produce un error y que dos objetos iguales deben tener el mismo *hashCode()*.

Como **Randoop** solo hace chequeos de igualdad, permitirle utilizar *hashCode()* haría que se produjeran tests que corroboren que el resultado de *hashCode()* es un número en particular, llegando a una sobreespecificación.

7. Ejercicio 3

Para implementar el **repOk** definí un método el cual chequea si *elems* es distinto de null y a su vez si *readIndex* es mayor o igual a -1 y menor estricto que la longitud de *elems*, en base a esto, evalúa uno a uno cada elemento de *elems* chequeando si el mismo es null.

Una vez finalizada la evaluación correspondiente devolverá **True** o **False** en base a la evaluación de cada elemento mencionada.

Finalmente ejecuto de nuevo Randoop pero ahora con un **presupuesto de 1 minuto**, obteniendo **1621 Regression Tests** y **188 Error-Revealing o Failing tests**.

Este resultado me pareció muy interesante, ya que indica que al realizar el método **repOK** y aumentar el tiempo de ejecución, **Randoop** pudo generar más entradas para testear y de mejor calidad al restringir la generación de entradas, concentrándose solamente en entradas válidas de **StackAr**, Por ultimo, **Randoop** generó test cases que fallan, lo que revela que el programa no funciona como debería.

8. Ejercicio 4

Luego de estudiar cual podría ser la causa de falla, pude concluir que la misma se producía cuando se eliminaba un elemento del **stack**, el mismo no era suplantado por un null, rompiendo la condición de que todos los elementos de índice mayor a *readIndex* debían ser null. Por lo tanto, hacer que *pop()* ponga un null en la posición del elemento que acaba de sacar arregló los tests que fallaron anteriormente. Se corrió nuevamente **Randoop** por un minuto, y no se encontró ningún test que falle.

9. Ejercicio 5

9.1. ¿Cuántas mutantes construye PiTest? ¿Cual es el mutation score (mutantes vivos / mutantes totales) que reporta PiTest?

PiTest construye 50 mutaciones y obtiene un score del %80.

Encontramos varios mutantes que sobrevivieron los cuales podemos distinguirlos de la siguiente manera:

- En el método *pop()* sobrevivió uno que cambió el valor de retorno por null.
- En el método *hashCode()* sobrevivieron varios mutantes que cambiaban las operaciones matemáticas.
- En el método *equals()* sobrevivieron algunos mutantes que cambiaban los valores devueltos en algunas líneas.
- En el método *repOk()* los cuales cambiaban los valores de retorno y algunas operaciones, entre otras cosas.

9.2. Extender manualmente el test suite para obtener el mejor mutation score posible con PiTest. ¿Cual es el mejor mutation score que pudo obtener? ¿Que mutantes equivalentes encontró?

El mejor score que se pudo lograr mediante el agregado de tests custom fue del %94.

Esto se logró refactorizando en una función estática a la función *repOk*: como los mutantes se encontraban en ella, se necesitaba poder hacer testing sobre la misma: es decir testear la función que decidıa si mi estructura pasaba o no todos los tests. Para ello se generó una clase paralela a *StackAr*, llamada *ExtendedStack* la cual utiliza como *repOk* a la función estática de *StackAr* y presenta errores para poder testear a la función *repOk*.

Las mutaciones que no pudieron ser evitadas se efectuaron por código aportado por la cátedra (hashCode, equals) fueron:

- **equals**: el mutante se encuentra en la evaluación de readIndex al final de todas las clausulas, que como se respeta la representación de la estructura, una desigualdad por este atributo nunca se da (se generan mutantes en código muerto, con lo cual dan equivalencias).
- **hashCode**: parte de las mutaciones se pudieron eliminar haciendo tests sobre la implementación del método en si: se especificó en el test el uso del primo 31 como número mágico, otra parte de las mutaciones no pudieron ser eliminadas ya que se abordaban temas de overflow de enteros para la generación del hash.

Para eliminar las mutaciones sobre el equals basta cambiar el orden de evaluación del predicado sobre *readIndex* hasta antes de que se evalúe la igualdad de los arrays elems. Por parte del *hashCode*, para poder eliminar completamente todos los mutantes, se deberıa poder tener información sobre el arreglo *elems* (el cual es privado y no se puede tener acceso externo a él)