Ingeniería del Software II

Taller #4 – Mutation Analysis + Random Testing

LEER EL ENUNCIADO COMPLETO ANTES DE ARRANCAR.

Fecha de entrega: 7 de Octubre de 2024 **(16:55hs)**
Fecha de re-entrega: 28 de Noviembre de 2024 (23:55hs) **(no hay extensiones)**

Descargar el proyecto *StackAr* del campus, e importarlo en la IDE IntelliJ IDEA. Este proyecto contiene una implementación de un *stack* (pila) en Java, junto con varios *test suites* que lo prueban.

Parte 1 (23/9/2024): Mutation Analysis

Setup y contenido

Además del *stack* (pila) en Java y los *test suites*, el archivo descargado contiene clases para generar mutantes del programa original y para calcular el **mutation score** de un *test suite*. En detalle:

- El paquete `org.autotest.mutants` contendrá los mutantes del programa original (arranca vacío).
- El paquete `org.autotest.mutantGenerator.operators` contiene los operadores de mutación que se pueden aplicar.
- Las clases `Mutant` y `MutantsGenerator` se encargan de generar los mutantes del programa original.
- La clase `Main` es la que se debe ejecutar para generar los mutantes del programa original.
- La clase `StackAr` es la implementación original del stack.
- La interfaz `Stack` es la interfaz que implementa `StackAr`, y todos los mutantes que generemos.
- La clase abstracta `MutationAnalysisRunner` es la clase base que debemos extender para escribir *test suites* y poder calcularles el **mutation score**.

Para generar los mutantes del programa original debemos correr la clase `Main` del proyecto. Esto se puede hacer desde la IDE haciendo click derecho sobre la clase y seleccionando *Run Main.main()*. Los mutantes generados se encuentran en la carpeta `src/main/java/org/autotest/mutants` del proyecto. Cada mutante contará con un comentario *JavaDoc* al principio que indica el operador de mutación que se aplicó.

Para correr los tests, hacer click derecho sobre el módulo test en la IDE y seleccionar *Run Tests*. El reporte de *coverage* generado por `JaCoCo` al finalizar los tests se encuentra en el archivo `build/reports/jacoco/test/html/index.html`.

Conceptos básicos

La técnica de *Mutation Analysis* nos permite juzgar la efectividad de un *test suite* midiendo cuán bien puede detectar defectos "artificiales". En esta técnica, un **mutante** es una versión levemente modificada del programa original que está bajo *test*. Un test detecta un mutante si el test pasa en el programa original y falla en el programa mutante. Decimos que un mutante está **vivo** si ningún test encuentra el defecto que introduce. Por otro lado, un mutante está **muerto** si al menos un test falla (i.e., detecta el defecto que introduce). Luego, para calcular el **mutation score** de un *test suite* tenemos que:

- Generar los mutantes del programa original.
- Para cada mutante, ejecutar todos los tests del *test suite* sobre el mutante.
- El **mutation score** es la cantidad de mutantes muertos dividido la cantidad total de mutantes. Por ejemplo, si tenemos 100 mutantes y 80 están muertos, el **mutation score** es 80 %.

Framework Spoon

El taller utiliza el framework *Spoon*[1] para generar los mutantes del programa original. Puede encontrar más documentación sobre *Spoon* en los siguientes links:

- Las clases de *Spoon* que representan partes estructurales del programa como declaraciones de clases y métodos: `https://spoon.gforge.inria.fr/structural_elements.html`
- Las clases de *Spoon* que representan código ejecutable Java como bloques *if*: `https://spoon.gforge.inria.fr/code_elements.html`
- Las clases de *Spoon* que se pueden utilizar para crear fragmentos de código Java: `https://spoon.gforge.inria.fr/factories.html`

Ejercicio 1

Completar todos los operadores de mutación en los paquetes `org.autotest.mutantGenerator.operators.*` de la carpeta `src/main`. Los lugares a completar se indican con el comentario `// COMPLETAR`. Los operadores de mutación propuestos en el taller se basan en algunos de los que están definidos en la herramienta PiTest[2]. El operador implementado debe *modificar* el código original (esto quiere decir que, nunca puede generar como mutante el mismo programa original, sino que debe existir algún cambio sintáctico en el mismo).

Para completar la implementación de cada operador, se debe:

- Completar la implementación del método `isToBeProcessed(CtElement candidate)`, que devuelve `true` si el operador se puede aplicar al nodo `candidate` del AST, y `false` en caso contrario.
- Completar la implementación del método `process(CtElement candidate)` que aplica el operador al nodo `candidate` del AST y/o las funciones auxiliares necesarias.

A modo de ejemplo, se proveen las implementaciones de los siguientes operadores:

- `org.autotest.mutantGenerator.operators.binary.ConditionalsBoundaryMutator`,
- `org.autotest.mutantGenerator.operators.constants.MinusOneConstantMutator`, y
- `org.autotest.mutantGenerator.operators.returns.EmptyReturnsMutator`.

Para esta implementación se deberán considerar las siguientes operaciones binarias provistas en `BinaryOperatorKind`:

Operador	Descripción
PLUS (+)	Suma
MINUS (-)	Resta
MUL (*)	Multiplicación
DIV (/)	División
MOD (%)	Módulo/Resto
BITAND (&)	AND binario
BITOR (|)	OR binario
BITXOR ($\hat{}$)	XOR binario
SL ($\ll$)	Signed Shift Left
SR ($\gg$)	Signed Shift Right
USR ($\gg>$)	Unsigned Shift Right
EQ (==)	Igualdad
NE (! =)	Desigualdad
LT (<)	Menor estricto
GT (>)	Mayor estricto
LE (<=)	Menor o igual
GE (>=)	Mayor o igual

[1] `https://spoon.gforge.inria.fr`

[2] `https://pitest.org/quickstart/mutators/`

así como las siguientes operaciones unarias provistas en `UnaryOperatorKind`:

Operador	Descripción
PREINC (++)	Suma y luego retorna el valor (++a)
PREDEC (–)	Resta y luego retorna el valor $(--a)$
POSTINC (++)	Retorna el valor y luego suma (a++)
POSTDEC (–)	Retorna el valor y luego resta $(a--)$

Algunas clases que pueden ser de utilidad (referencia completa en `https://spoon.gforge.inria.fr/code_elements.html`)

Clase	Elemento del Código
`spoon.reflect.code.CtBinaryOperator`	un operador binario a op b
`spoon.reflect.code.CtIf`	una condición $a > b$
`spoon.reflect.code.CtLiteral`	un literal i
`spoon.reflect.code.CtReturn`	una sentencia `return`
`spoon.reflect.code.CtUnaryOperator`	un operador unario op a

Una vez implementados los operadores, utilizarlos para generar los mutantes del programa original `StackAr`, y responder:

a ¿Cuántos mutantes se generaron en total?

b ¿Qué operador de mutación generó más mutantes? ¿Cuántos y por qué?

c ¿Qué operador de mutación generó menos mutantes? ¿Cuántos y por qué?

Ejercicio 2

Utilizando los mutantes generados, evaluar los *test suites* provistos por la cátedra en el proyecto (`StackTests1` y `StackTests2`). Responder:

a ¿Cuántos mutantes vivos y muertos encontraron cada uno de los *test suites*?

b ¿Cuál es el *mutation score* de cada *test suite*?

Ejercicio 3

Extender el *test suite* `StackTests2` para obtener el mejor **mutation score** posible. En el archivo llamado `StackTests3` puede encontrar una copia de los tests lista para ser extendida. Cada uno de los métodos que se agreguen deben comenzar con el prefijo `test`. A su vez, deben utilizar los métodos `createStack()` y `createStack(int capacity)` para construir *stacks* con capacidad *default* y con capacidad fijada manualmente, respectivamente.

Responder:

a ¿Cuál es el *mutation score* logrado para los tests del `StackTests2` mejorado (i.e. `StackTests3`)?

b ¿Cuántos mutantes vivos y muertos encontraron?

c Comente cuáles son todos los mutantes vivos que quedaron y justifique por qué son mutantes equivalentes al programa original[3].

[3] si no fueran mutantes equivalentes, entonces debe crear un test case que pueda detectarlo y así mejorar el *mutation score* obtenido

d ¿Cuál es el *instruction coverage* promedio que lograron para las clases mutadas? Puede encontrar este valor al final del reporte de *JaCoCo* para el paquete `org.autotest.mutants` (la última fila da el *Total*).

e ¿Cuál es el peor *instruction coverage* que lograron para una clase mutada? ¿Por qué creen que sucede esto?

Parte 2 (30/9/2024): Random Testing

Setup y Contenido:

Vamos a utilizar `Randoop` para generar tests unitarios, para ello contamos con una *Gradle task* llamada **randoop** que limpia, genera y modifica los tests. Esta tarea se puede ejecutar desde la IDE abriendo el panel *Gradle* y haciendo doble-click sobre la tarea **randoop** (en la sección de *verification*). Los archivos generados por `Randoop` se guardan en la carpeta `src/test/java/org/autotest/generated` del proyecto, con el nombre `RegressionTest#.java`. Tenga en cuenta que cada archivo puede contener más de un test. Para inspeccionar la configuración de la herramienta `Randoop`, puede revisar los argumentos de la *Gradle task* **randoop** en el archivo `build.gradle`.

Una vez generados, puede correr los tests unitarios desde la IDE haciendo click derecho sobre el módulo *generated* y seleccionando *Run Tests*. El reporte de *coverage* generado por `JaCoCo` al finalizar los tests se encuentra en el archivo `build/reports/jacoco/test/html/index.html`.

Tener en cuenta que los reportes de JaCoCo se sobreescriben por lo tanto deben mover a otro path los que deseen guardar.

Herramienta Randoop

`Randoop` es una herramienta de generación automática de casos de test unitarios para `Java` (en formato `JUnit`). Esta herramienta implementa una técnica de generación de casos de tests aleatorios, guíados por retroalimentación (*feedback-directed random testing*[4]). Cada caso de test consiste en una secuencia de llamadas a métodos de la clase bajo prueba, seguida de una aserción que captura el comportamiento esperado de la clase bajo prueba. `Randoop` puede ser utilizado con dos propósitos: encontrar errores en un programa y crear tests de regresión para detectar si cambia el comportamiento de su programa en el futuro.

En este taller, utilizaremos `Randoop` para generar automáticamente casos de test unitarios para la clase `StackAr`. Los siguientes links proveen más información al respecto de esta herramienta:

- Website de `Randoop`: `https://randoop.github.io/randoop/`
- Manual de uso de `Randoop`: `https://randoop.github.io/randoop/manual/index.html`

Ejercicio 4

En este ejercicio vamos a ejecutar `Randoop` sobre la clase `StackAr` para que genere todos los tests aleatorios posibles durante 15 segundos. Para ello, debe ejecutar la *Gradle task* llamada **randoop** que limpia y genera tests escribiéndolos en el paquete `org.autotest.generated` de la carpeta `src/test/java` Una vez generados, responder las siguientes preguntas:

a ¿Cuántos casos de test produjo `Randoop`?

[4]Feedback-directed random test generation by Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. In ICSE '07: Proceedings of the 29th International Conference on Software Engineering.

b ¿Alguno de los tests generados falló?

c ¿Cuál es el *instruction coverage* alcanzado por los tests generados para la clase `StackAr`?

Ejercicio 5

En este ejercicio vamos a utilizar `Randoop` para detectar fallas en la clase `StackAr`. Para ello, debe completar el método `repoOK()` de la clase `StackAr` para que retorne **true** solamente si la estructura del `StackAr` es válida. Tener en cuenta que el código de este método no debe tirar excepciones. Una instancia de `StackAr` es válida sii:

- $elems \neq null$
- $readIndex \geq -1$ y $readIndex < elems.length$
- $\forall\ i > readIndex,\ elems_i = null$

Una vez completado el método `repoOK()`, ejecutar `Randoop` por 1 min en lugar de 15 segundos sobre `StackAr` y correr los tests generados. Si algún test generado falla, reparar el defecto que produjo la falla y repetir el proceso hasta no encontrar mas errores. Luego, responder:

a ¿Qué fallas fueron detectadas por los tests producidos por `Randoop`?

b ¿Cuál es el *instruction coverage* alcanzado por el último test suite generado (aquel que no encontró nuevas fallas en `StackAr`)?

Ejercicio 6

En este ejercicio vamos a combinar generación de casos de tests con un análisis de mutaciones. Para ello, vamos a ejecutar nuevamente `Randoop` para generar tests unitarios sobre la clase `StackAr` reparada. Esta vez, usaremos la *Gradle task* llamada **randoopWithMutationAnalysis** que no solo genera casos de tests con `Randoop`, sino que también los modifica para poder ejecutar el mutation analysis que fue desarrollado anteriormente en la Parte #1 del Taller.

Ejecutar el nuevo archivo `RegressionTest.java` (sin número) generado y reportar cuál es este mutation score obtenido por los tests generados sobre al clase `StackAr` reparada.

Formato de Entrega

El taller debe ser entregado en el campus de la materia. La entrega debe incluir un archivo `entrega.zip` con el código completo del proyecto *StackAr* modificado por ustedes. El cuál debe contener:

- El código de los operadores de mutación que completaron.
- Los mutantes generados.
- El *test suite* (`StackTests3`) que escribieron para mejorar el **mutation score** (ejercicio #3).
- El archivo `StackAr` con sus modificaciones correspondientes.
- Los tests generados por Randoop (con los generados la última vez alcanza).

Además no se olviden de:

- Los reportes de *JaCoCo* para: `StackTests3` y el final de Randoop (ejercicios #3 y #5).
- Un archivo `RESPUESTAS` con las respuestas a las preguntas de los ejercicios.