Ingeniería del Software II

Taller #1 – Mutation Analysis

LEER EL ENUNCIADO COMPLETO ANTES DE ARRANCAR.

Fecha de entrega: 26 de Marzo de 2025 **(16:59hs)**
Fecha de re-entrega: 9 de Abril de 2025 **(16:59hs) (no hay extensiones)**

Conceptos básicos

La técnica de *Mutation Analysis* nos permite juzgar la efectividad de un *test suite* midiendo cuán bien puede encontrar defectos "artificiales". En esta técnica, un **mutante** es una versión levemente modificada del programa original que está bajo *test*. Notar que una versión idéntica del programa **no es un 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 %.

Setup y contenido del taller

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. Además contiene un archivo *respuestas.md* donde deberán contestar las preguntas. Además, 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.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`.

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
EQ (==)	Igualdad
NE (! =)	Desigualdad
LT (<)	Menor estricto
GT (>)	Mayor estricto
LE (<=)	Menor o igual
GE (>=)	Mayor o igual

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

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

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

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 ej: 'a op b'
`spoon.reflect.code.CtIf`	una condición ej: '$a > b$'
`spoon.reflect.code.CtLiteral`	un literal ej: 'i'
`spoon.reflect.code.CtReturn`	una sentencia ej: '`return`'
`spoon.reflect.code.CtUnaryOperator`	un operador unario ej: 'op a'

También tengan en cuenta la función CtTypeInformation::isPrimitive para comprobar si un valor de retorno es efectivamente un objeto.

Una vez implementados los operadores, utilizarlos para generar los mutantes del programa original `StackAr`, corriendo la clase *Main*. Pueden notar que se muestran por consola la cantidad y tipos de mutantes generados, ésta información también se guarda en un archivo llamado *mutants_info.txt*.

Luego 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 de `StackTests3`?

b ¿Cuántos mutantes vivos y muertos encontraron?

c Comente cuáles son todos los mutantes vivos que quedaron y porqué son equivalentes al programa original (si no lo fueran, todavía es posible mejorar el *mutation score*).

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?

Formato de Entrega

El taller debe ser entregado en el campus de la materia. La entrega debe ser un archivo `entrega.zip` con el **proyecto completo** *StackAr* que descargaron del campus, con su código modificado.

Esto incluye:

- El código de los operadores de mutación que completaron.
- Los mutantes generados.
- El *test suite* que escribieron para mejorar el *mutation score*.
- El reporte de *JaCoCo* para el *test suite* final que hayan armado.
- El archivo `respuestas.md` con las respuestas a las preguntas de los ejercicios.