Taller #3 - Random Testing

Ingeniería de Software II

Alumno: Lucas Raposeiras
LU: 34/15

Ejercicio 1

Randoop es una herramienta para generar casos de test aleatoriamente.

Para compilar el archivo `StackAr.java` corremos el comando `mvn compile` en el directorio `stackar` utilizando Java 8. Este comando nos genera un archivo `StackAr.class` en el directorio `stackar/target/classes/org/autotest/`.

Luego ejecutamos Randoop con el siguiente comando:

```
java -ea -classpath lib/randoop-all-4.2.3.jar:target/classes randoop.main.Main gentests --testclass=org.autotest.StackAr
--time-limit=15 --testsperfile=500 --junit-output-dir=src/test/java
```

La parte resaltada en rojo se encarga de ejecutar el *entry point* de Randoop, mientras que el resto del comando contiene parámetros de Randoop.

Parámetros de Randoop:

- `gentests` es el comando de Randoop que genera tests unitarios para un conjunto de clases.
- `--testclass` especifica la clase para la cual queremos generar tests. En este caso pondremos `org.autotest.StackAr`, que sería la clase `StackAr` del paquete `org.autotest`.
- `--time-limit` especifica el número máximo de segundos para dedicar a generar pruebas. Cero significa que no hay límite. Si no es cero, Randoop no es determinístico **(puede generar diferentes conjuntos de pruebas en diferentes ejecuciones)**. En este caso, el ejercicio nos pide ejecutar Randoop por 15 segundos, así que pondremos `15`.
- `--testsperfile` especifica el número máximo de pruebas para escribir en cada archivo JUnit. En este caso pondremos `500` para que los archivos no sean tan grandes.
- `--junit-output-dir` especifica el nombre del directorio en el que se deben escribir los archivos JUnit. En este caso pondremos `src/test/java` ya que ese es el directorio donde `maven` buscará los archivos de tests para ejecutarlos.

Luego de ejecutar el comando y esperar 15 segundos a que termine, Randoop nos crea los archivos `RegresionTest.java` y `RegressionTest0.java` en el directorio `stackar/src/test/java/`. En la salida del comando notamos que Randoop reporta que **se generaron 359 *regression tests***. También podemos ver que este número coincide con la cantidad de métodos que se generaron en el la clase `RegressionTest0.java`. La cantidad de tests que se generan dependen de la capacidad de procesamiento del equipo en el cual se ejecutó Randoop. En el caso de mi equipo *(MacBook Pro 2017, 2.3 GHz Dual-Core Intel Core i5, 8 GB 2133 MHz LPDDR3)*, con 15 segundos se generaron 359 tests.

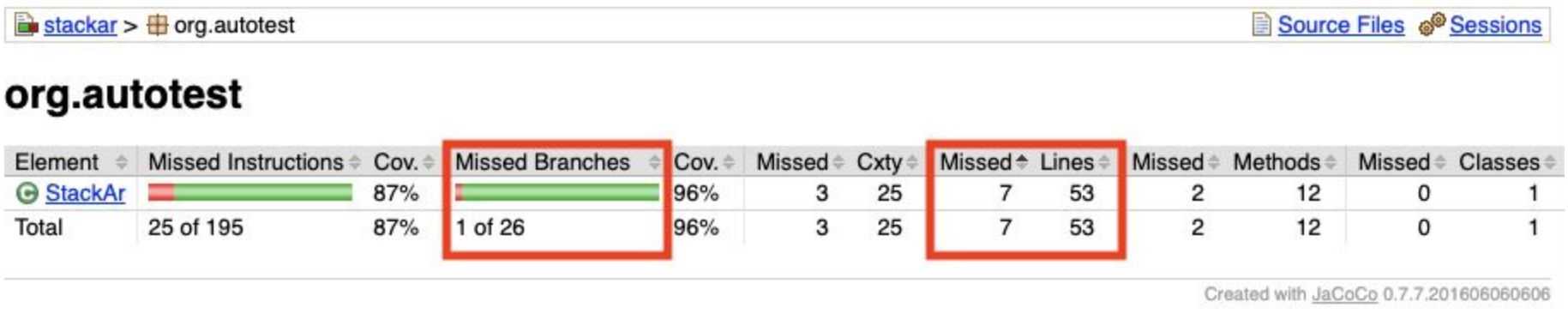
Ningún test case generado por Randoop es un failing test case. Esto lo sabemos debido que no se generó ningún archivo del tipo `ErrorTest.java` luego de la ejecución de Randoop. Esto significa que, cuando ejecutemos los tests en el ejercicio 2, **ninguno debería fallar.**

Ejercicio 2

JaCoCo es una herramienta que genera reportes de cuánto porcentaje de código cubre un test suite (*coverage*).

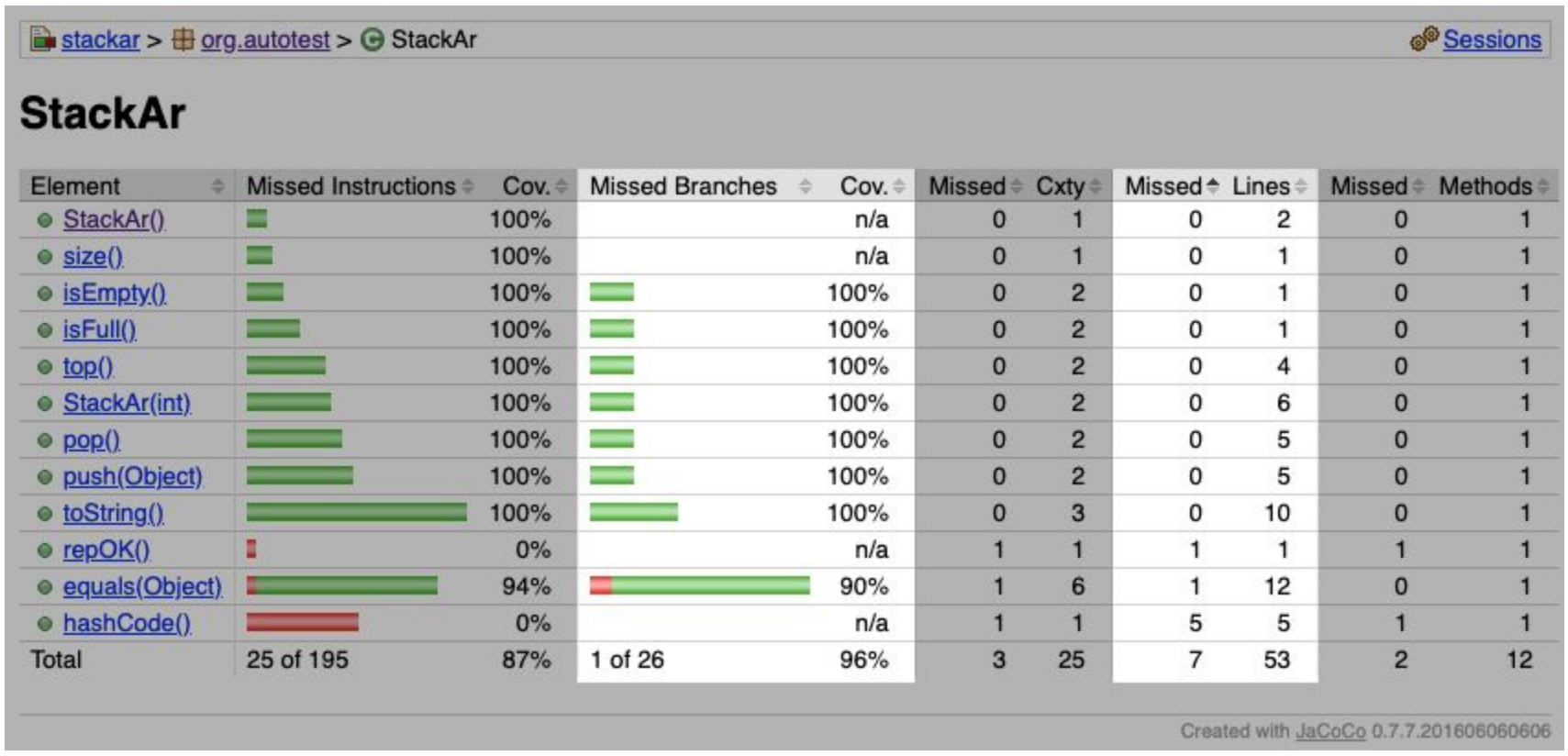
Para ejecutar JaCoCo podemos utilizar el comando `mvn test`, o bien `mvn clean install`.

Luego de ejecutar el comando y esperar a que termine, JaCoCo nos crea el directorio `stackar/target/site/jacoco/` con los archivos del *coverage*. Si abrimos el archivo `index.html` nos encontramos lo siguiente:



Allí podemos ver que se reporta 1 branch de 26 sin cubrir, y 7 líneas no cubiertas de 53. En otras palabras, se reporta que **se cubrieron 25 branches de 26 y 46 líneas de 53**.

Si entramos en detalle a ver la el reporte de la clase `StackAr` nos encontramos lo siguiente:



Allí vemos que la branch que nos falta cubrir se encuentra en el método `equals`, mientras que las líneas que nos falta cubrir se encuentran en los métodos `repOK`, `equals` y `hashCode`. Si bien es verdad que Randoop genera test aleatorios, los cuales pueden no cubrir el 100% del código, vamos a explicar a continuación que estos resultados tienen una explicación más concreta y que está relacionada a los métodos recién mencionados.

Veamos en detalle método `equals`:

```
1.     @Override
2.     public boolean equals(Object obj) {
3.         if (this == obj)
4.             return true;
5.         if (obj == null)
6.             return false;
7.         if (getClass() != obj.getClass())
8.             return false;
9.         StackAr other = (StackAr) obj;
10.        if (!Arrays.equals(elems, other.elems))
11.            return false;
12.        if (readIndex != other.readIndex)
13.            return false;
14.        return true;
15.    }
16.
```

Allí vemos, en rojo, que la rama true del if de la línea 3 no fue cubierto. Esto se debe a que ningún test generado por Randoop hizo un *assert* de `equals` entre una misma instancia de `StackAr`. Esta rama no cubierta del if también se corresponde con una de las 7 líneas no cubiertas de toda la clase.

Veamos en detalle el método `repOK`:

```
1.     @CheckRep
2.     public boolean repOK() {
3.         return true;
4.     }
5.
```

Allí vemos, en rojo, que la línea 3 no fue cubierta. Esto se debe a que ningún test generado por Randoop ejecuta explícitamente el método repOK, sino que éste es usado internamente por Randoop como invariante de representación. Randoop utiliza el invariante de representación para ver cuáles tests son failing tests cases. De hecho, ahora que vemos la implementación del método, nos damos cuenta que en el ejercicio 1 no había ningún failing test case ya que el invariante de representación devuelve siempre `true` (es decir, cualquier instancia de `StackAr` en cualquier estado es válida).

Veamos en detalle el método `hashCode`:

```
1.     @Override
2.     public int hashCode() {
3.         final int prime = 31;
4.         int result = 1;
5.         result = prime * result + Arrays.hashCode(elems);
6.         result = prime * result + readIndex;
7.         return result;
8.     }
```

Allí vemos, en rojo, que las líneas 3 a 7 no fueron cubiertas. Randoop no genera tests que ejecutan hashCode ya que hashCode es un método difícil de testear porque devuelve un resultado no determinístico.

Ejercicio 3

Completamos el método `repOK` con la siguiente implementación:

```
    @CheckRep
    public boolean repOK() {
1.        for (int i = this.readIndex + 1; i < this.elems.length; i++) {
2.            if (this.elems[i] != null) {
3.                return false;
4.            }
5.        }

6.        return (this.elems != null && this.readIndex >= -1 && this.readIndex < elems.length);
    }
```

Veamos que esta implementación retorna **`true`** sólo si la instancia de `StackAr` cumple el invariante de representación y **`false`** en caso contrario.

Una instancia de `StackAr` es válida sii:

- $elems \neq null$
- $readIndex \geq -1 \wedge readIndex < elems.length$
- $(\forall\ i > readIndex)\ elems_i = null$

Es decir, si las **todas** las condiciones **se cumplen**, la instancia es válida, y `repOK` debería devolver **`true`**.
Por otro lado, si **alguna** de estas tres condiciones **no se cumple**, la instancia es inválida, y `repOK` debería devolver **`false`**.

En el método vemos que hay dos líneas donde se hace **`return`**: la línea 3 y la línea 6. En la línea 3 se devuelve **`false`** explícitamente, con lo cual la línea 6 es la única línea donde `repOK` puede devolver **`true`**. Para que la ejecución alcance la línea 6, se tiene que haber ejecutado el ciclo **`for`** que se encuentra en la línea 1.

Este ciclo comprueba si existe algún elemento entre las posiciones $readIndex + 1$ y $elems.length$ que sea $null$. Si existe, devuelve false. Con lo cual, este ciclo se encarga de verificar la tercer parte del invariante.

En la línea 6 se retorna false si `this.elems == null`, con lo cual, esto verifica la primer parte del invariante. En el resto de la línea 6 se verifica la segunda parte del invariante.

Luego de implementar el método `repOK` ejecutamos Randoop (esta vez durante un minuto) y notamos que tenemos **246 failing tests**.

Ejercicio 4

Si observamos el archivo de failing tests generado por Randoop en el ejercicio 3 (`ErrorTest0.java`) vemos que en todos los failing tests siempre pasa que, luego de ejecutar el método `pop` sobre una instancia de `StackAr`, el invariante de representación se deja de cumplir en esa instancia. Esto lo notamos ya que Randoop escribe el comentario `// Check representation invariant` luego de cada llamado a `pop`, y después hace fallar el test intencionalmente. Con lo cual, **es probable que el método `pop` tenga un *bug* de implementación**.

Veamos en detalle el método `pop`:

```
    public Object pop() throws IllegalStateException {
1.        if (isEmpty()) {
2.            throw new IllegalStateException();
3.        }
4.        Object rv = this.top();
5.        readIndex--;
6.        return rv;
    }
```

Rápidamente notamos que el método rompe esta parte del invariante: $(\forall\ i > readIndex)\ elems_i = null$. Más en específico, deja de cumplirse para $i = readIndex$, ya que nunca se setea **`null`** en el tope de la pila luego de obtener el elemento. Arreglamos esto introduciendo la siguiente línea al código:

```
    public Object pop() throws IllegalStateException {
1.        if (isEmpty()) {
2.            throw new IllegalStateException();
3.        }
4.        Object rv = this.top();
5.        this.elems[readIndex] = null;
6.        readIndex--;
7.        return rv;
    }
```

Luego de arreglar el *bug* ejecutamos Randoop nuevamente (otra vez, durante un minuto) y esta vez notamos que **no tenemos failing tests**. Es decir, o bien solucionamos efectivamente el *bug*, o bien esta nueva tanda de tests no testea nunca el método `pop`. Descartamos la segunda opción ya que vemos bastantes tests que ejecutan el método `pop`.

Ejercicio 5

PiTest es una herramienta que genera mutantes. Un mutante es una leve diferencia de código. Por ejemplo, cambiar un *mayor* por un *mayor-o-igual*.

Para ejecutar PiTest utilizamos el comando `mvn clean install org.pitest:pitest-maven:mutationCoverage`.

Luego de ejecutar el comando y esperar a que termine, PiTest nos crea el directorio `stackar/target/pit-reports/` con información del *coverage* de las mutaciones. Además, nos informa por consola que **se generaron 49 mutantes, de los cuales se mataron 31.** Es decir, **el mutation score es de 0.63 (63%)**.

```
----------------------------------------------------------------------------
> Total  : 18 seconds
----------------------------------------------------------------------------
============================================================================
- Statistics
============================================================================
>> Generated 49 mutations Killed 31 (63%)
>> Ran 80 tests (1.63 tests per mutation)
============================================================================
- Mutators
============================================================================
> org.pitest.mutationtest.engine.gregor.mutators.ConditionalsBoundaryMutator
>> Generated 6 Killed 3 (50%)
> KILLED 3 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 3
----------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.IncrementsMutator
>> Generated 2 Killed 1 (50%)
> KILLED 1 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 1
----------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.ReturnValsMutator
```

Nos gustaría ahora aumentar el mutation score. Para ello debemos aumentar el número de de mutantes que se matan. Decimos que un matamos un mutante cuando existe un test que lo cubre. En este caso vamos a generar tests **manualmente** para asegurarnos que cubran la mayor cantidad posible de mutantes.

Para ver los mutantes generados, abrimos el reporte que nos generó PiTest. Allí nos encontramos con las siguientes líneas de código sin cubrir:

```
@CheckRep
public boolean repOK() {
	for (int i = this.readIndex + 1; i < this.elems.length; i++) {
		if (this.elems[i] != null) {
			return false;
		}
	}
	return (this.elems != null && this.readIndex >= -1 && this.readIndex < elems.length);
}
```

Para cubrir este método creamos un test haga uso de él. Por ejemplo, podemos hacer un trest que haga `assertTrue(stackAr1.repOK())`, donde `stackAr1` es un `StackAr` vacío.

```
@Override
public int hashCode() {
	final int prime = 31;
	int result = 1;
	result = prime * result + Arrays.hashCode(elems);
	result = prime * result + readIndex;
	return result;
}
```

Para cubrir este método creamos un test haga uso de él. Por ejemplo, podemos hacer un test que haga `assertEquals(<número>, stackAr1.hashCode())`, donde reemplazamos número por el hashCode de la instancia `stackAr1`.

```
@Override
public boolean equals(Object obj) {
	if (this == obj)
		return true;
	if (obj == null)
		return false;
	if (getClass() != obj.getClass())
		return false;
	StackAr other = (StackAr) obj;
	if (!Arrays.equals(elems, other.elems))
```

```
            return false;
      if (readIndex != other.readIndex)
            return false;
      return true;
}
```

Para cubrir esta línea simplemente creamos un test que haga `assertTrue(stackAr1.equals(stackAr1))`, siendo `stackAr1` cualquier instancia válida.

El mejor mutation score que pude obtener fue de 0.94 (94%), habiendo matado 46 mutantes de 49. Las líneas que no pude cubrir son las siguientes:

```
@CheckRep
public boolean repOK() {
      for (int i = this.readIndex + 1; i < this.elems.length; i++) {
            if (this.elems[i] != null) {
                  return false;
            }
      }
      return (this.elems != null && this.readIndex >= -1 && this.readIndex < elems.length);
}
```

El motivo por el cual no pude cubrir estas líneas es que no pude construir una instancia inválida de `stackAr1` con los métodos públicos que expone la clase.

En cuanto a mutantes equivalentes, el único que encontramos es el siguiente:

```
@Override
public int hashCode() {
      final int prime = 31;
      int result = 1;
      result = prime * result + Arrays.hashCode(elems);
      result = prime * result + readIndex;
      return result;
}
```

En este caso, la mutación lo que hacía era reemplazar la operación de multiplicación por una división.