Assignment 6: White-Box Testing

(INDIVIDUAL ASSIGNMENT)

Goals:

- Get familiar with white-box testing.
- Understand some subtleties of structural coverage.

Preliminary Steps:

- Download the archive assignment6.tar.gz.
- Extract the archive in the root directory of the **personal** repo we assigned to you.
  - This will create a directory called `Assignment6` in the root directory of the repo. Hereafter, we will call this directory `<dir>`.
  - This will also create the following files:
    - A skeleton of the class that you need to complete for this assignment:
      `<dir>/src/edu/gatech/seclass/GlitchyClass.java`
    - An example of the kind of JUnit test classes you need to create for this assignment:
      `<dir>/test/edu/gatech/seclass/ExampleTestSC1.java`
    - A JUnit library to be used for the assignment:
      `<dir>/lib/junit-platform-console-standalone-1.9.1.jar`

Instructions:

To complete the assignment, perform the following tasks (after reading the instructions in their entirety):

- **Task 1**: Create in class `GlitchyClass` a method called `glitchyMethod1` that contains a division by zero fault and at least two branches, such that (1) it is possible to create a non-empty test suite that achieves less than 100% statement coverage and reveals the fault, and (2) it is possible to create a test suite that achieves 100% statement coverage and does not reveal the fault.
  - The method can have any signature.
  - **See note 6 below for prohibited language constructs.**
  - If you think it is not possible to create a method meeting both requirements, then:
    - Create an empty method.
    - Add a comment in the (empty) body of the method that **concisely but convincingly** explains why creating such a method is not possible.

    - Conversely, if you were able to create the method, create two JUnit test classes as follows:
        - `GlitchyClassTestSC1a` should achieve less than 100% statement coverage of `glitchyMethod1` and reveal the fault therein.
        - `GlitchyClassTestSC1b` should achieve 100% statement coverage of `glitchyMethod1` and **not** reveal the fault therein.
        - Both classes should be saved in the directory `<dir>/test`. (The full actual path should obviously also reflect the package structure, and the same holds for the test classes in the subsequent tasks.)

- **Task 2**: Create in class `GlitchyClass` a method called `glitchyMethod2` that contains a division by zero fault and at least two branches, such that (1) it is possible to create a test suite that achieves 100% path coverage and reveals the fault, (2) **every possible** test suite that achieves 100% branch coverage does not reveal the fault.
    - The method can have any signature.
    - **See note 6 below for prohibited language constructs.**
    - If you think it is not possible to create a method meeting both requirements, then:
        - Create an empty method.
        - Add a comment in the (empty) body of the method that **concisely but convincingly** explains why creating such a method is not possible.
    - Conversely, if you were able to create the method, create two JUnit test classes as follows:
        - `GlitchyClassTestPC2` should achieve 100% path coverage of `glitchyMethod2` and reveal the fault therein.
        - `GlitchyClassTestBC2` should achieve 100% branch coverage of `glitchyMethod2` and **not** reveal the fault therein.
        - Both classes should be saved in the directory `<dir>/test`.

- **Task 3**: Create in class `GlitchyClass` a method called `glitchyMethod3` that contains a division by zero fault and at least two branches, such that (1) **every possible** test suite that reveals the fault achieves 100% statement coverage, and (2) it is possible to create a test suite that achieves 100% branch coverage and does not reveal the fault.
    - The method can have any signature.
    - **See note 6 below for prohibited language constructs.**
    - If you think it is not possible to create a method meeting both requirements, then:
        - Create an empty method.
        - Add a comment in the (empty) body of the method that **concisely but convincingly** explains why creating such a method is not possible.

- Conversely, if you were able to create the method, create two JUnit test classes as follows:
  - `GlitchyClassTestSC3` should achieve 100% statement coverage of `glitchyMethod3` and reveal the fault therein.
  - `GlitchyClassTestBC3` should achieve 100% branch coverage of `glitchyMethod3` and **not** reveal the fault therein.
  - Both classes should be saved in the directory `<dir>/test`.

- **Task 4**: Class `GlitchyClass` contains the following method `glitchyMethod4`:

```
public static int glitchyMethod4(int a, int b, int c, int d, boolean e) {
  int result = 0;
  if (a == 0) {
    if ((b == c) || ((d < 0) && (e))) {
      result = 1;
    } else {
      result = 2;
    }
  } else {
    result = 3;
  }
  return result;
}
```

  Create two JUnit test classes as follows, **where each test case performs a single invocation of glitchyMethod4**.
  - `GlitchyClassTestSC4` should achieve 100% statement coverage of `glitchyMethod4` and contain at most **3** test cases.
  - `GlitchyClassTestMCDC4` should achieve 100% Modified Condition/Decision Coverage (MC/DC) of `glitchyMethod4` and contain at most **8** test cases.
  - Partial credit will NOT be awarded if `GlitchyClassTestSC4` and `GlitchyClassTestMCDC4` do not satisfy their respective requirements.
  - **Bonus Points**: If `GlitchyClassTestMCDC4` achieves 100% MC/DC coverage with the minimum possible number of test cases, then you will receive 10 bonus points. Determining the actual minimum is part of the bonus task (but you may safely assume the minimum is less than or equal to 8).
  - Both classes should be saved in the directory `<dir>/test`.

- **Task 5**: Method `glitchyMethod5` in the provided class `GlitchyClass` contains:
  - The code below, commented out:

```
public boolean glitchyMethod5 (boolean a, boolean b) {
    int x = 1;
    int y = 0;
    if(a) {
        x -= y;
    } else {
        y -= x;
    }
    if(b) {
        x += y ;
    }
    if(y>=0) {
        x += 1;
    }
    x = 6 / (x + y);

    return (x > 0);
}
```

  - The following table, where you can provide your responses as strings (please note that the table in the code has a slightly different format because we want to be able to run that part of the code):

```
// ================
//
// Replace the "?" in column “output” with “T”, “F”, or “E”:
//
// | a | b |output|
// ================
// | T | T |   ?  |
// | T | F |   ?  |
// | F | T |   ?  |
// | F | F |   ?  |
// ================
```

  - The following sentences, where you can again provide your responses as strings (also in this case, the format in the code is different to make the code executable):

```
//
// Replace the "?" in the following sentences with “NEVER”,
// “SOMETIMES” or “ALWAYS”:
//
// - Test suites with 100% path coverage “?”
//   reveal the fault in this method.
// - Test suites with 100% branch coverage “?”
//   reveal the fault in this method.
```

```
// - Test suites with 100% statement coverage “?”
//   reveal the fault in this method.
// ================
```

Fill in your answers in the method, as follows:

- For every possible input, replace the "?" in column “output” of the table, with "T" to indicate that the return value is true, or "F" to indicate that the return value is false, or "E" to indicate that the method exits with a division by zero error.
- In the sentences following the table, replace the “?” with “NEVER”, “SOMETIMES”, or “ALWAYS” to indicate whether a test suite with 100% coverage for the specified criterion NEVER reveals the fault, SOMETIMES reveals the fault, or ALWAYS reveals the fault in the `glitchyMethod5` method.

Notes (important–make sure to read carefully):

1. When we say “create in class `GlitchyClass` a method called `glitchyMethodN`”, we mean that you should complete the placeholder for that method provided in the skeleton class, by either modifying its code (and possibly signature) or providing a suitable comment and an empty body.
2. Explanations of why a method cannot be created, if any, must be written using the format provided in the method examples in the `GlitchyClass` skeleton class.
3. Committing (or omitting) class `ExampleTestSC1` has no effect on your grade.
4. Similarly, leaving or removing the example methods provided in the `GlitchyClass` skeleton has no effect on your grade.
5. By “reveal the fault therein”, we mean that **the tests that trigger the division by zero fault should FAIL with an uncaught** ArithmeticException **fault**, so that they are easy to spot. Make sure that the correct tests fail, and show the correct type of exception when they fail. See ExampleTestSC1 for an example.
6. **Every one of these methods, if the task is possible, can be written with simple, short code. In particular, you cannot use any of the following constructs in your solution:**[1]
   a. **Compound Predicates.** That is, only use simple predicates in the form (<operand1> <operator> <operand2>), such as “if (x > 5)” or “if (a > b)”. In other words, **you cannot use logical operators (such as &&, ||) in your predicates**.
   b. **Nested if statements (including “else if” constructions).**
   c. **Loops**. (while, for, do, etc.)
   d. **Method calls**. Your method should be self-contained and not call any other methods (including itself).

[1] Using any of these constructs would result in a zero for that task.

e. **Conditional (trinary) operators**. (No use of the "<condition> ? <case if true> : <case if false>" construct.)
f. **Switch statements**.
g. **Streams, lambdas,** or other newer complex features of the language.
h. **Dead or unreachable code**.

7. Read the requirements carefully. For example, "**Every possible** test suite..." means **all conceivable test suites** for your method, not only the example test suite you write.
8. Your code should compile and run out of the box with Java 11.
9. Use the provided JUnit library for your JUnit tests.
10. **We strongly recommend computing coverage manually**, as tools sometimes use slightly different definitions of coverage and may have some quirks. Given that the solutions typically consist of very simple methods, it should be easy to compute coverage by hand (which is how we do it when we grade the assignment).
11. This is an **individual assignment**. You are not supposed to collaborate with your team members (or any other person) to solve it. We will enforce this by running a plagiarism detection tool on all assignments. Given the numerous different ways in which the assignment can be solved, similar solutions will be (1) easily spotted and (2) hard to justify.
12. Similarly, make sure not to post on Ed Discussion any solution, whether complete or partial, and also to avoid questions that are too specific and may reveal information about a specific solution, or imply which tasks you believe are possible. You can obviously ask these types of questions privately to the instructors.

Submission:

- As usual, commit and push your code to your individual, assigned private repository.
- Make sure that all Java files are committed and pushed (i.e., class `GlitchyClass` and any created JUnit test classes).Also make sure to commit and push the provided libraries (lib directory). To do so, you may need to force add the jar files (i.e., "`git add -f lib/*`"), which are typically excluded by the "`.gitignore`" file.
- You can check that you committed and pushed all the files you needed by doing the following:
  - Clone a fresh copy of your personal repo in another directory
  - Go to directory `Assignment6` in this fresh clone of your repo
  - Compile your code. One way to do is to run, from a Unix-like shell:
    ```
    javac -cp lib/\* -d classes src/edu/gatech/seclass/GlitchyClass.java
    test/edu/gatech/seclass/*.java
    ```
  - Run one or more of your tests. Again, from a Unix-like shell, you can run:
    ```
    java -cp classes:lib/\* org.junit.platform.console.ConsoleLauncher --
    ```

`select-class edu.gatech.seclass.<test class name>`[2]
(at least some of the tests should fail with an `ArithmeticException`)

- **Submit on Gradescope a file called `submission.txt` that contains, in two separate lines, (1) your GT username and (2) the commit ID for your submission.** For example, the content of file `submission`.txt for George P. Burdell could look something like the following:

`submission.txt`

```
gpburdell1
81b2f59
```

As soon as you submit, Gradescope will check your assignment by making sure that your files are present and in the correct location, compiling the code, running your tests, and performing some sanity checks on the outcome of such tests. If you pass all these checks, you will see a placeholder grade of 10 and a positive message from Gradescope. Otherwise, you will see a grade of 0 and an error message with some diagnostic information. Please note that **a positive response from Gradescope only indicates that you passed the sanity checks and is meant to prevent a number of trivial errors.** Please also note that **if your submission does not pass the Gradescope checks, it will not be graded and will receive a 0**, so please make sure to pay attention to the feedback you receive when you submit and keep in mind that **you can resubmit as many times as you want before the deadline.**[3]

[2] If using a Windows-based system, you may need to run java -cp "classes;lib/*" org.junit.platform.console.ConsoleLauncher --select-class edu.gatech.seclass.<test class name> instead.

[3] Although we tested the checker, it is possible that it might not handle correctly some corner cases. If you receive feedback that seems to be incorrect, please contact us on Ed Discussion.