CIS 573 Software Engineering – Fall 2017
Homework #5
Due Friday, Dec 1, 5:00pm

Introduction

This assignment covers various topics we've seen in recent weeks: fault-based testing, defensive programming, and integration testing.

As always, you must work **alone** on this assignment. Although you are free to discuss the intent of the assignment and ask your fellow classmates for clarification, the code that you write must be your own.

Part 1: Defensive Programming and Mock Objects (50 points)

In this part of the assignment, you are asked to modify and then write tests for a piece of code that relates to a simple social networking concept: suggesting friends to a student based on who is taking the same classes.

Getting Started.
Download the **Friends-testMe** Eclipse project in Canvas. This contains the *FriendFinder* class, which contains the *findClassmates* method you will modify and test. The Eclipse project also contains two empty classes (*ClassesDataSource* and *StudentsDataSource*) that are only provided so that the code will compile.

Even though *FriendFinder.findClassmates* generally works correctly, it is susceptible to the types of issues discussed in the lecture on reliability, in particular unexpected inputs and mistakes made by the programmer who is using this code and the programmer whose code we are using. So first you will modify the code to use defensive programming to make sure that the code does not throw any exceptions but still works correctly.

To show that your changes have made the code more reliable, you will then write test cases that cover your changes. However, the method you will test has a dependency on external data sources, e.g. some sort of database, but rather than implement a real back-end, you will instead use mock objects for your testing.

Step 1.
In this step, you will modify *FriendFinder.findClassmates* so that it uses defensive programming.

This method takes a Student object that includes the name of a student and then returns a Set containing the names of everyone else who is taking the same classes as that student. For instance, if the argument to the method represents me, and:

- I am taking CIS573 and CIS550
- Alice is taking CIS573, CIS550, and CIS555

- Bob is taking CIS573 and CIS555
- Chen is taking CIS550 and CIS573
- Dhriti is taking CIS550

then the method should return a List containing Alice and Chen, since both of them are taking the same classes I am; however, it should not contain Bob or Dhriti since Bob is not taking CIS550 and Dhriti is not taking CIS573.

We will assume for our purposes that the *FriendFinder.findClassmates* method works correctly for good/valid inputs.

However, this code does not make any attempt to handle values that are **null**, including the input to the method, the objects on which it depends, and the objects that are returned from the methods it invokes. In any of these cases, the current code will throw a NullPointerException if it tries to dereference a null object. Which is bad.

Using defensive programming, modify *FriendFinder.findClassmates* so that it does not throw any NullPointerExceptions. Specifically, the method should:

- Use IllegalArgumentException and IllegalStateException appropriately as discussed in class
- Return null if the input Student is not taking any classes or if there are no students taking the same classes as that student
- Ignore any other null values encountered during the operation of the method
- Otherwise, return the correct expected output according to the description above

Note that **we are intentionally not listing the different possible null values** that the code may encounter! It is up to you to determine what those may be and to write code to handle them. Please do not post public questions on Piazza that reveal possible solutions or that ask about specific test cases.

Keep in mind that it is not sufficient to simply put a try/catch block around the entire method and catch any NullPointerException that arises, since in some cases the method must throw a different exception, in some cases it must return null, and in others it must simply ignore the null value and return the correct output. Nice try, though. ☺

Step 2.
Now that you've modified the code to use defensive programming, how do you know that it won't throw a NullPointerException if it encounters a null value? Well, you'd write tests, of course!

In this step, you will test the different situations in which the method may encounter a null value. As you can see, this method has a dependency on both the *ClassesDataSource.getClasses* and *StudentsDataSource.getStudents* methods. As discussed in class, there are various challenges to testing a method that has a dependency on other classes, so we will instead use mock objects.

Create a JUnit test class called *FindClassmatesTest* that covers all of your defensive programming code in Step 1, i.e. the parts of the code that you added that checks for null values and handles them properly. Use mock objects to control the behavior of *ClassesDataSource.getClasses* and *StudentsDataSource.getStudents* as needed, and passing these mock objects to the *FriendFinder* constructor. You do not need to write tests for any of the "normal" functionality of this method, though you can if you'd like!

Please note that your tests must be **sound**: the expected output should be determined by the specification above, and not by the implementation itself. If you have questions about the specification as it relates to the handling of null values, you may post a *private* note in Piazza, but please do not post a public note that may give away a solution.

General Guidelines.
Aside from using defensive programming, you should **not** change the implementation of *FriendFinder.findClassmates*. If you feel it is necessary to do so, please contact a member of the instruction staff.

Also, you may not change the *Student*, *StudentsDataSource*, or *ClassesDataSource* code at all, nor should you change the *FriendFinder* constructor or the signature of *FriendFinder.findClassmates.* Likewise, you should not change that method's behavior for good/valid inputs, but rather should only be looking for and handling null objects that it may encounter.

Remember, you are writing tests for the *FriendFinder.findClassmates* method, **not** the *Students*- and *ClassesDataSource* implementations. Ask a member of the instruction staff for guidance if you are unsure what to do.

Last, you may **not** use mock object testing frameworks like Mockito without the permission of the instructor. Use anonymous inner classes in your test cases as discussed in lecture.

Part 2: Mutation Analysis (50 points)

In this part of the assignment, you will use a fault-based testing approach called mutation analysis to create a set of unit tests, and use a tool called muJava to measure how good your test set is.

Before You Begin

muJava systematically inserts mutations into your code, and then allows you to run your JUnit tests to see how many of those mutations are detected, or "killed." First, make sure you can install and run muJava before proceeding. This can be a bit challenging, especially if you're not comfortable using the command-line, so don't wait too long to try to set this up!

Setup/Configuration

Start by downloading muJava, which is available in Canvas.

You will also need to have Java 1.7 or later installed on your machine; this version of muJava will not work with Java 1.6 or earlier.

Before you begin conducting mutation analysis on the tests for this part of the assignment, make sure that you're able to run muJava on the example code that is provided in the distribution. There is online documentation at http://cs.gmu.edu/~offutt/mujava/ but here's what you need to do in order to set up your environment:

1. Unzip the file that you downloaded.
2. Edit **mujava.config** so that MuJava_HOME refers to the directory that contains that file. Be sure it's the full absolute path, not a relative path. If you're on Unix or Mac, be sure that you start the directory name with a leading slash. However, on all platforms, make sure the directory name does **not** end with a trailing slash.
3. In this step you will edit one of the scripts that runs muJava.
    - **Windows:** Edit line 3 of **genmutants.bat** and set the JAVA_TOOLS variable to include the full path to the **tools.jar** file on your computer. It should be part of the Java installation, and the entry in the file may already be correct. If the path contains a whitespace, be sure to put the path in quotes, as in the file we provided.
    - **Mac:** Edit line 9 of **genmutants.sh** and set the JAVA_TOOLS variable to include the full path to the **tools.jar** file on your computer. It should be part of the Java installation, and the entry in the file may already be correct, though the version of Java may be slightly different.
    - **Linux:** Edit **genmutants.sh** and set the JAVA_TOOLS variable to include the full path to the tools.jar file on your computer. It should be part of the Java installation, and the entry in the file may already be correct, though the version of Java may be slightly different. You can do this by editing line 6 and un-commenting it out; be sure to comment out or delete line 9.

On all platforms, if you have trouble finding the Java installation and the tools.jar file, try the command:

```
        java -XshowSettings:properties -version
```

which will show a lot of Java configuration information and give you some indication of where it is installed.

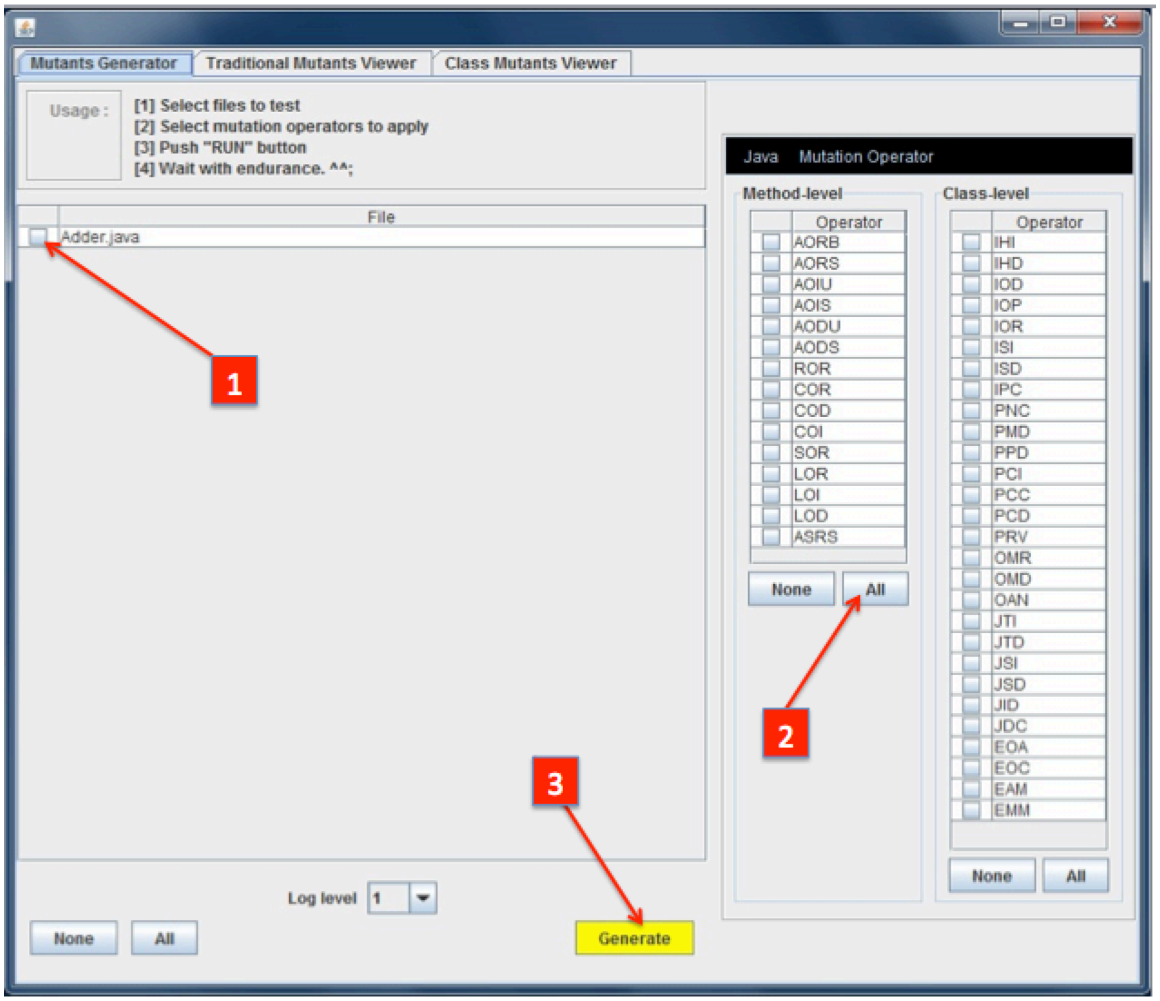
At this point, you should be ready to start generating mutants!

Generating Mutants

muJava uses the **src/** directory to hold the source code of the files it will mutate, and the **classes/** directory to hold the compiled code. You should see a file called Adder.java in src/ and Adder.class in classes/. We will use these to make sure that muJava works correctly before proceeding.

Open a command prompt or terminal window on your computer and navigate to the directory containing mujava.config. Then run the **genmutants.sh** (Linux/Mac) or **genmutants.bat** (Windows) script. For Linux/Mac, you may need to change the permissions so that the file is executable, e.g. using "chmod a+rx genmutants.sh" and you may need to run it as "./genmutants.sh"

You should see the UI start and look something like this:



Now do the following:

1. Select the name of the file (in this case, "Adder.java") from the list of files. This will be the file into which we'll insert mutations.
2. Click the "All" button for method-level mutation operators. Note that you only need to use method-level mutants, not class-level.
3. Click the "Generate" button at the bottom of the window.

You won't see anything happen in the UI, but in the console window from where you started the UI, you should see it say the name of the file and then, after a second or two, "All files are handled."

To check whether the mutations were generated, select the "Traditional Mutants Viewer" tab. In the "Summary" pane (on the left), it should say that 15 mutants were created. You can see what they are (i.e., the difference between the mutated version and the original) by selecting on each, e.g. AORB_1, AORB_2, etc. You can probably infer what the different mutant names mean, but they are documented at http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf

Once you're done, close the UI window.

If you get an error about Java versions, please notify a member of the instruction staff; it may be that your version of Java is older than the ones used to generate the class files.

If you get other errors, please check the Troubleshooting Guide in Piazza.

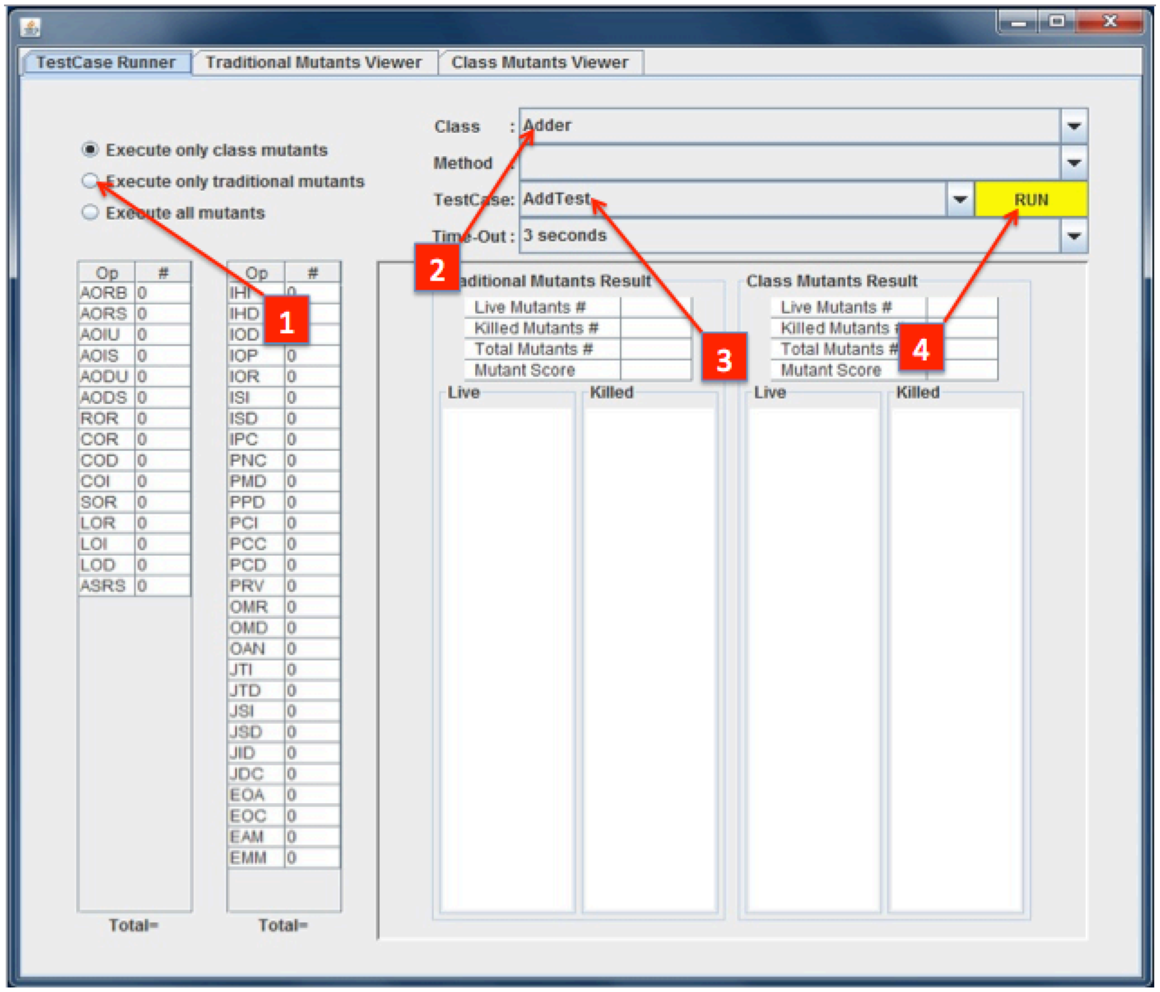
Conducting Mutation Analysis

Now we'll see how many mutants are killed by a simple test set. In the **testset/** directory, there is a JUnit class AddTest.java, as well as a compiled version AddTest.class. We will use the test methods in this class to attempt to kill the mutants.

A mutant is considered killed if all tests in the test set pass on the original (unmutated) version and at least one test fails on the mutated version.

From the directory containing mujava.config, run the **testmutants.sh** (Linux/Mac) or **testmutants.bat** (Windows) script. For Linux/Mac, you may need to change the permissions so that the file is executable, e.g. using “chmod a+rx testmutants.sh” and you may need to run it as “./testmutants.sh”

You should see the UI start and look something like this:



● Execute only class mutants
○ Execute only traditional mutants
○ Execute all mutants

Class : Adder
Method :
TestCase: AddTest
Time-Out : 3 seconds

Op	#
AORB	0
AORS	0
AOIU	0
AOIS	0
AODU	0
AODS	0
ROR	0
COR	0
COD	0
COI	0
SOR	0
LOR	0
LOI	0
LOD	0
ASRS	0

Op	#
IHI	0
IHD	
IOD	
IOP	0
IOR	0
ISI	0
ISD	0
IPC	0
PNC	0
PMD	0
PPD	0
PCI	0
PCC	0
PCD	0
PRV	0
OMR	0
OMD	0
OAN	0
JTI	0
JTD	0
JSI	0
JSD	0
JID	0
JDC	0
EOA	0
EOC	0
EAM	0
EMM	0

Traditional Mutants Result

Live Mutants #	
Killed Mutants #	
Total Mutants #	
Mutant Score	

Live Killed

Class Mutants Result

Live Mutants #	
Killed Mutants #	
Total Mutants #	
Mutant Score	

Live Killed

Now do the following:

1. Select the “Execute only traditional mutants” radio button.
2. Choose “Adder” from the “Class” dropdown list (it should be chosen by default).
3. Choose “AddTest” from the “TestCase” dropdown list (it should be chosen by default).
4. Click the “Run” button.

In the console from where you started the UI, you'll see the results of the test cases for each mutant. This should only take a few seconds to run.

When it’s done, you'll see the results in the “Traditional Mutants Result” pane. It should say that the Mutant Score is 73%.

If you got to this point okay, then you’re ready to move on!

Activity
In this part of the assignment, you will generate a set of tests specifically to kill the mutants that are generated by muJava, without using any of the black-box or white-box adequacy criteria seen previously.

The method you will be testing is in the *FlightSimulator* class. It contains a method called *simulateFlights* that takes the following parameters:

- *planes*: an array of Airplane objects, each of which has a location, bearing, and velocity
- *steps*: the number of discrete time steps to simulate
- *safeDistance*: the minimum allowable safe distance between two Airplanes

The method works as follows:

- First, it checks whether all of the Airplanes are at a safe distance. If not, it returns false.
- If the Airplanes are all at a safe distance, it then simulates one unit of time and moves each Airplane to its new location according to its location, bearing, and velocity. For our purposes, a bearing of 0 means due east; 90 means due north, 180 means due west; and 270 means due south.
- After moving each plane one step, it then calculates the minimum distances between the line segments caused by each Airplane's movement. If any two line segments have a minimum distance smaller than the *safeDistance*, the method returns false (this admittedly may lead to some false positives, but better safe than sorry when it comes to air travel!).
- If the planes are all at safe distances, the method continues to the next step by moving the Airplanes once again, and so on.
- If the method is able to go through the number of steps without the Airplanes ever being less than the *safeDistance* apart, the method returns true.
- If any of the arguments to the method are invalid (including illegal values for any Airplane's velocity), the method throws an IllegalArgumentException.

Your goal in this part of the assignment is to create a test set in a class called *FlightSimulatorTest* that kills as many method-level mutants in the *simulateFlights* methods as possible.

When generating the mutants in the muJava GUI, select "FlightSimulator.java" as the File. Then click the "All" button for method-level operators, but then **unselect** AOIS and LOI. That is, **you do not need to include the AOIS and LOI mutations** in this assignment, nor do you need to use the class-level mutants. After a few seconds, muJava should generate a total of 89 mutants (you can confirm this by looking in the Traditional Mutants Viewer pane after generating the mutants).

Note that you are *not* being asked to achieve any particular level of equivalence class or code coverage, and you should *not* be modifying FlightSimulator.java. You should *only* be creating test cases in *FlightSimulatorTest* in an attempt to kill all the mutants.

You can begin by adding tests to the *FlightSimulatorTest* class in the testset/ directory. This test should kill 4 of the 89 mutants.

For simplicity, you can edit the FlightSimulatorTest.java file in the testset/ directory and then compile it using the compiletests.bat (Windows) or compiletests.sh (Linux/Mac) script that is provided in the muJava distribution. Keep in mind that the compiled FlightSimulatorTest.class file must be in the testset/ directory.

If you're not comfortable with compiling and running your JUnit tests on the command-line, it may be easier to move everything into Eclipse, add your test cases to make sure that they behave as expected, and then move the modified FlightSimulatorTest.class file from the Eclipse workspace's bin/ directory to the testset/ directory in muJava.

Note that you'll often have one test case kill many mutants. That is fine! You don't need to write 89 separate test cases. ☺ But please adhere to good JUnit style and convention as discussed in previous lectures and assignments.

You're done with this part once you've killed as many mutants as possible. But it may be that it is impossible to kill 100% of the mutants! If so, submit a PDF in which you list each of the mutants that cannot be killed (using their IDs) and explain why.

As discussed in class, mutation analysis is mostly intended as a way of answering "am I done testing?" and is only meaningful if all tests pass. If you identify a sound test case that fails (i.e. not a "false positive"), please notify a member of the instruction staff. It may be necessary to fix any bugs before proceeding.

A few words about using Piazza...

Please be careful about how you use Piazza for this assignment.

Please use Piazza for (a) clarification questions regarding the intent of the assignment, (b) clarification questions regarding the specification for each part, and (c) getting help with problems using muJava.

Please do **not** post questions that reveal your solutions about which mutants can and cannot be killed or how to go about killing certain mutants in Part 1, or what changes you made or which test cases you wrote in Part 2. These are things that all students need to figure out on their own.

Before you post anything on Piazza, think to yourself "is my question giving away an answer?" If you think it is, please email the instruction staff directly and we'll post it if we think it's appropriate.

Academic Honesty and Collaboration

You are expected to work **alone** on this assignment.

You are free to discuss the intent of the assignment with your classmates and ask clarification questions, and also to help other students with setting up muJava.

However, you may not discuss or share solutions or findings with other students. In particular, *you absolutely must **not** share or discuss your test code or specific test cases*. Please see the course policy on academic honesty (posted in Canvas on the "Syllabus" page) for more information.

As always, if you run into problems, please ask a member of the teaching staff for help before trying to find solutions online or asking your classmates!

Deliverables

For this assignment, submit a *single* zip file containing:

- The modified *FriendFinder.java* and your *FindClassmatesTest.java* source code from Part 1
- *FlightSimulatorTest.java* from Part 2
- A PDF listing surviving mutants from Part 2 (if any) and an explanation of why they could not be filled

Please do **not** submit the entire muJava installation or Eclipse project; just submit the required Java source code and PDF.

Submission

All deliverables are due in Canvas by **Friday, December 1, at 5:00pm**. Late submissions will be penalized by 10% if 0-24 hours late, 20% for 24-48 hours late, and so on, up to one week, after which they will no longer be accepted.

Last updated: 11 November 2017, 4:47pm