CIS 573 Fall 2015
Homework #4

Introduction

This assignment consists of two parts, related to the most recent testing-related topics we've covered in class.

In the first part of this assignment, you will create JUnit tests using black-box testing and strong normal equivalence class coverage.

Then in the second part, you will test a piece of code using dependency injection and mock objects.

You must work **alone** on this assignment. As before, 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.

Because of the short timeline, this assignment is worth only 75 points (other assignments have been worth 100 points).

Part 1: Black-box Testing (25 points)

In this part of the assignment, you will write unit tests for a method called *numFlights* that is part of an airline reservation system and allows the user to search for flights between two cities.

Specification.
The method takes the following parameters:

- *home*: a String representing the user's home airport
- *dest*: a String representing the intended destination airport
- *direct*: a boolean indicating whether or not the user only wants direct flights
- *timeLimit*: an int representing the maximum duration to consider (for all segments of the flight in total)

The method then returns the number of flights that match the search criteria by searching through a list of Flight objects provided by the static Flight.getAllFlights method (which will be provided to you, so you can see what the valid data is). The *home* and *dest* parameters are **not** case sensitive.

If *direct* is true, the method should return 1 if there is a flight from *home* to *dest*. It should also put the appropriate Flight object (which comes from Flight.getAllFlights) into the FlightFinder object's *directFlights* List as a side effect; you can query that by using the *getDirectFlights()* method.

If *direct* is true and there is no flight from *home* to *dest*, then the method should return 0 and the *directFlights* List should be empty. Regardless of whether any flights are found, if *direct* is true then the *getIndirectFlights()* method in the FlightFinder should return an empty List.

If *direct* is false, then in addition to considering any direct flights as described above, the return value of the method should also include combinations that consist of two segments (e.g. *home* to X and then X to *dest*). However, in those cases, it will only report flight combinations in which the total combined time of the two flights is less than or equal to the specified *timeLimit*. If so, these are considered by the return value to be a single flight, even though they consist of two flight segments. If any indirect flight combinations are found, the two Flight objects should be placed into a two-element Flight array, with the Flight representing *home* to X in the first element, and the Flight representing X to *dest* in the second, and then the array should be placed into the *indirectFlights* List as a side effect.

If the *home* or *dest* parameter does not specify a legal airport code (i.e., one of the codes used in Flight.getAllFlights, keeping in mind that they are not case-sensitive), or if *direct* is false and *timeLimit* is negative, the method should return -1.

Testing Activities.
Using the specification above, identify equivalence classes and then write JUnit tests for the *numFlights* method.

So that your tests will compile, we have provided an empty implementation of the method an Eclipse project called Flights-testMe in Canvas. Note also that there is a JUnit test class in the project to get you started.

Please adhere to the convention of only exercising one test case per test method, i.e. don't put all the test cases into a single super-gigantic method. In this particular case, each JUnit method should only call the method you're testing *once*.

Note that you are *not* being asked to implement *numFlights* method here; you are just writing tests against its specification, as you would in Test-Driven Development.

This part of the assignment will be graded based on the amount of “weak robust” black-box coverage that your test set achieves. You do not have to worry about boundary conditions but you do need to make sure that you cover all of the equivalence classes, including the robustness cases, and your tests of course must be sound.

Please be careful about accidentally posting solutions on Piazza! If you have a question about the specification, please ask a member of the instruction staff. If you feel that something is not addressed in the specification, then consider it a robustness case, write a test for it, and use your best judgment to determine what the expected/correct behavior should be.

Part 2: Dependency Injection and Mock Objects (50 points)

In this part of the assignment, you are asked to write tests for two methods that relate to simple social networking concepts: suggesting friends and finding other students who take the same classes.

The two methods have a dependency on external data sources, e.g. some sort of database, but rather than implement a real back-end, you will use mock objects instead.

Getting Started.

Download the Friends-testMe Eclipse project in Canvas. This contains the FriendFinder class, which contains the two methods you will test. You will notice that the Eclipse project also contains an interface called DataSource, and three empty classes that implement that interface and are only provided so that the code will compile.

Step 1.

First, write tests for **FriendFinder.suggestFriend**. This method takes a String representing the name of a student and then returns the name of another student who has the most friends in common. For instance, if the argument to the method represents me, and:

- I am friends with Alice and Bob
- Alice is friends with me, Carol, and Dan
- Bob is friends with me, Carol, and Eve

then the method should suggest Carol as my friend, since we have two friends in common, whereas I only have one friend in common with Dan and Eve.

There is no rule for tiebreaking if two or more other students have the same number of friends in common with me. For instance, if in the above example Bob were instead friends with me, Carol, Eve, and Dan, then the method could correctly suggest either Carol or Dan, since I have two friends in common with both of them.

If the argument to the method is null or is an empty String, the method should return null.

If the student named in the argument has no friends, or their friends have no other friends, then the method should return null in those cases, too.

Additionally, the method should not suggest the person or any of his or her current friends. If there are no new friends to suggest, the method should return null.

As you can see, this method has a dependency on the FriendsDataSource.get method. This method takes a String and returns a List containing the names of that person's friends; if input is null or an empty String, or if the person has no friends, the method should return null.

As discussed in class, there are various challenges to testing a method that has a dependency on an external data source, so we will instead use mock objects.

Start by refactoring the FriendFinder.suggestFriend method so that you can use dependency injection. Then, using the specification above to determine the expected output, create a JUnit test class called SuggestFriendTest that contains tests that cover the following cases (assuming the argument to the method represents "me"):

1. I have no friends (aww! so sad!)
2. I have at least two friends; they have no friends except for me (also kinda sad)
3. I have at least two friends; those friends are also friends with each other, and with me, but with no one else (kinda like on the TV show "Friends")
4. I have exactly one friend; she has exactly two friends (me and one other person)
5. I have exactly one friend; she has at least three friends (me and at least two other people)
6. I have exactly two friends; they have exactly two friends: me and another person (both of my friends are friends with the same other person, who is not one of my friends)
7. I have at least two friends; they all have at least three friends (me and at least two other people); none of those friends-of-friends are the same person, i.e. my friends don't have any friends in common except for me
8. I have at least three friends; they all have at least three friends (me and at least two other people); all of my friends are friends with person A, who is not one of my friends; all but one of my friends are friends with person B, who is not one of my friends; none of my friends have any friends in common except for A and B
9. I have exactly three friends; they all have exactly four friends (me and exactly three other people); all of my friends are friends with both person A and person B, who are not one of my friends; none of my friends have any friends in common except for A and B

For grading purposes, please do not combine any of these test cases (even though it may be possible to do so), and please keep them in this order in SuggestFriendTest.java.

Step 2.
Next you will write tests for **FriendFinder.findClassmates**. This method takes a String representing the name of a student and then returns a List 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
- Carol is taking CIS550 and CIS573
- Dan is taking CIS550

then the method should return a List containing Alice and Carol, since both of them are taking the same classes I am; however, it should not contain Bob or Dan since Bob is not taking CIS550 and Dan is not taking CIS573. The ordering of the elements in the list is non-deterministic, so both {Alice, Carol} and {Carol, Alice} would be correct.

If the argument to the method is null or is empty, the method should return null. If the student named in the argument is not taking any classes, or if there are no other students taking the same classes, the method should return null in those cases, too.

As you can see, this method has a dependency on both the ClassesDataSource.get and StudentsDataSource.get methods. You can infer the behavior of each by looking at the FriendFinder.findClassmates code.

To test this method, start by refactoring FriendFinder.findClassmates so that you can use dependency injection. Then, using the specification above to determine the expected output, create a JUnit test class called FindClassmatesTest that contains tests that cover the following cases (assuming the argument to the method represents me):

1. I am taking no classes
2. I am taking one or more classes, but no other student is in those classes
3. I am taking two or more classes; one or more other students are taking a non-empty subset of the classes I am taking
4. I am taking two or more classes; exactly one other student is taking the exact same set of classes; one or more other students are taking a non-empty subset of the classes I am taking
5. I am taking two or more classes; two or more other students are taking the exact same classes as I am; one or more other students are taking a non-empty subset of the classes I am taking
6. I am taking two or more classes; two or more other students are taking a proper superset of the classes I am taking; one or more other students are taking a non-empty subset of the classes I am taking
7. I am taking two or more classes; two or more other students are taking the exact same classes as I am; two or more other students are taking a proper superset of the classes I am taking; one or more other students are taking a non-empty subset of the classes I am taking

For grading purposes, please do not combine any of these test cases (even though it may be possible to do so), and please keep them in this order in FindClassmatesTest.java.

General Guidelines.
Aside from refactoring the code to use dependency injection, you should **not** change the implementation of FriendFinder.suggestFriend or FriendFinder.findClassmates. If you feel it is necessary to do so, please contact a member of the instruction staff.

Likewise, you should not change the DataSource interface or the three empty classes that implement it. Any implementations of the DataSource interface should exist only as mock objects in your test cases.

Although there may be unexpected faults in the FriendFinder implementation, the test cases described above **should** pass so your tests must be sound. If you write a test that fails, make sure your test case is sound and contact a member of the instruction staff if you think you've discovered a bug. Remember, we are testing the FriendFinder methods,

not the DataSource implementations.

You are not expected to create additional test cases beyond the ones described above. Of course, if you want to keep testing, no one is stopping you!

A few words about using Piazza...

Please be careful about how you use Piazza for this assignment. It is important that you do not reveal (accidentally or intentionally) which test cases you've created since, after all, that is the point of all this.

Please use Piazza for clarification questions regarding the intent of the assignment and clarification questions regarding the specification of the methods you're testing.

Please do not post questions that reveal your solutions about which tests you created or how to construct mock objects for testing. 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.

Deliverables

Please submit a *single* zip file containing:

- the FlightFinderTest JUnit test class from Part 1
- the refactored version of FriendFinder.java from Part 2
- the SuggestFriendTest and FindClassmatesTest JUnit test classes from Part 2

Please do **not** submit the entire Eclipse project for either part of the assignment; just submit the four files listed above.

Submission

All deliverables are due in Canvas by Friday, November 6, 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.

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.

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!

Last updated: 27 Oct 2015, 8:21pm