Thomas Conahan

CSC 320 Project 2

2/14/23

All the assignments had relatively the same layout so most of the code I wrote in regards to testing was almost the same, switching out variable names and method names. For string validation, which was in almost all of the work, I simply checked that the string's length did not exceed the set limit. An example of what code is used would be from the Contact assignment:

**public** **void** lessThanLimit()

{

Contact contact =

**new** Contact("0000000000", "Thomas", "Conahan", "1234567890", "123 Fake Address" );

assertEquals(**true**, contact.getAdd().length() <= 30);

assertEquals(**true**, contact.getPhone().length() <= 10);

assertEquals(**true**, contact.getLast().length() <= 10);

assertEquals(**true**, contact.getFirst().length() <= 10);

assertEquals(**true**, contact.getID().length() <= 10);

}

This checked that all the required strings from the contact.java file were under their size limit. In addition to this, I had a method that checked that strings that were not allowed to be null, were not null.

**public** **void** notNullID()

And an example of the code

assertEquals(**true**, contact.getID() != **null**);

In a similar fashion to the string length check, I used the boolean and checked that the value of ID within the contact object was not equal to null. I used the same exact Contact declaration to ensure that both of these checks would work at the same time. In regards to the hold of the objects that was required in all of the assignments, I used an arraylist due to it being easily able to grow and shrink in size with the addition and subtraction of elements. In order to check that these arraylist were able to add and subtract elements, I used public void sizeUp() and public void sizeDown(). In these method I used a boolean to check if after use of the delete method I created, that the size of the arrayList was either less than or greater than the original size of the arrayList. In the contact service and task service assignment we were asked to have some variables be accessible and updateable. I created a method, public void updated(), which checked to see that the fields changed after using the created update methods. Using booleans I made sure that the original value when compared to the changed value returned false.

**public** **void** updated()

{

Contact contact = **new** Contact("0000000000", "Thomas", "Conahan", "1234567890", "123 Fake Address" );

String orgFirst = contact.getFirst();

String orgLast = contact.getLast();

String orgAdd = contact.getAdd();

String orgNum = contact.getPhone();

contact.updateLast("NewLast");

*assertEquals*(**false**, contact.getLast().equals(orgLast));

contact.updateFirst("NewFirst");

*assertEquals*(**false**, contact.getFirst().equals(orgFirst));

contact.updatePhone("NewPhone");

*assertEquals*(**false**, contact.getPhone().equals(orgNum));

contact.updateAdd("NewAdd");

*assertEquals*(**false**, contact.getAdd().equals(orgAdd));

}

The task service document had similar coding with the variable names switched to match the variables of that file. I believe that all my Junit tests came back with zero errors so I believe that it was very effective in testing the code. The only testing that I believe I should have done differently would have been the String length testing. I should have set a limit on the string within the declaration of the objects, and had the constructor not allow the user to send something over the limits. However, this would require the use of a UI that was not meant to be constructed within the assignments. I believe that the way I tested them works for the way I code the objects, but I believe there is a better way I could have tested the lengths of the strings had I created the constructors and/or method differently for the classes.

There are a variety of different types of software testing. In these assignments, I used mainly unit testing, which is essentially breaking down the code into small units and testing each unit. While I could have not tested every update method for each assignment, due to the methods being constructed in the same way, unit testing is the testing of the smallest units and I felt that it was the safest bet to test each method and that they worked correctly. Other examples of testing would be white and black box testing, which tests more on the backend side that everything internally is running correctly. The difference between the two would be that black box is more testing for the user and white box is less concerned with the user. Overall, they both check that the internals and backend of the application runs correctly and efficiently.

My mindset was not much different when acting as a software tester. I created code with the confidence I normally have. When it came to creating the actual tests I was a bit cautious as it was new territory to me. As the term went on however, I became more confident in my ability to create tests and by the last assignment I was creating tests without any outside help. It is important to understand how code interacts in order to create the best possible tests for that code. This is shown by how the tests are created for the data storage that holds the objects we created. To create the best test possible, we have to understand how the data structure stores and access its objects, and how you can access the methods of that object. When it comes to bias, I tried to look at code as if it wasn’t my own and constantly questioned if there were better ways I could have constructed the classes as well as the tests of those classes. Being biased when testing can lead to huge gaps in security as well as functionality due to you feeling the way you initially wrote as being the best possible way. Cutting corners in code can lead to underlying issues further down the line. Keeping up with the industry standards is a good way to stay out of technical debt. Keeping true to the basics will also help to keep your code clean and well written.