Benjamin Abbott

CS 320

Software Test Automation & QA

April 13, 2022

**Project Two**

**Summary**

The testing approach that I took was fairly well aligned to the software requirements. Each requirement given, created limitations on the inputs. This created a need for each of these limitations to be tested to see if they work properly. For example, in the Contact.java file there is a requirement that states, “The contact object shall have a required firstName String field that cannot be longer than 10 characters. The firstName field shall not be null.” This requirement created two separate limitations for the input of the string and therefore requires two tests to confirm that they have been implemented properly. As you can see the tests were a direct result of the requirements. So, it can be said that every requirement must be tested, but the requirements are not the only tests that need to be used. I also had to test to test the other aspects of the code. I needed to confirm the creation of the objects when called. I needed to test that each of the attributes were correctly assigned to the object. I had to test that the constructor and each mutator and accessor worked properly. These tests were not aligned with the requirements but were necessary to ensure that the program worked as intended. Therefore, while all requirements did have to be tested, they were not the only parts of the code that had to be tested.

I believe the JUnit tests that I created for this program were of high quality. I believe this because I was able to confirm that the coverage of each of these tests surpassed 99% coverage in the IDE. This means that the JUnit tests that I wrote were very effective and offered complete coverage of the program.

I was able to ensure that my code was technically sound by testing every part of the program to make sure that it performed exactly how I expected it to. I tested every class constructor, mutator, accessor, and function to do this. To do this I initialized a new class with each of the constructors. I then used assertTrue statements to make sure that they were properly assigned. For example, the line “assertTrue(contact.getContactID().equals(“12345”));” made sure that the attribute contactID was properly assigned “12345.” I wrote these statements for each attribute in each class. I then completed the same process with the mutators of the class by using them to change attributes and tested to make sure that they were in fact changed. I also tested each of the functions in the code to make sure they worked correctly as well. To do this I had to call on each function and compare the results with expected results. For example, to add contacts to the Contact Service I had to call the function and then make sure that the contact was actually added to the Contact Service; “contactService.addContact(testContact);” and then “assertEquals(1, contactService.contacts.size());.”

To ensure that my code was efficient, I refactored my code as much as I could to cut down on unnecessary operations. I limited the loops and if/else statements to as few as I could. I made sure to use functions from built in libraries when I could. Using built in functions will help reduce total lines of code by letting me avoid writing them by scratch and will result in more efficient run-time of the application. For example, in my function addContact in the Contact Service java file, I used .isEmpty(), .add(), and .equals():

public void addContact(Contact a) {

if(contacts.isEmpty()) {

contacts.add(a);

return;

}

for(Contact b: contacts) {

if (a.getContactID().equals(b.getContactID())) {

throw new IllegalArgumentException("Contact ID already in service");

}

} contacts.add(a);

**Reflection**

In order to fully test the software that I created, I had to use multiple testing techniques which included both static and dynamic testing. The static techniques that I used were review techniques which include looking for errors in logic and syntax. Through this I was able to find some critical logic errors which included adding an element to a list inside of a for loop that would have prevented the function of working as intended. Through the help of the Eclipse IDE, I was also able to fix syntax errors as they were made. These included missing parentheses, semicolons, and improper spelling.

I also used dynamic testing to test the software and I used both white box and experience-based techniques to do this. White box techniques inspect and verify the inner workings of a system by running the software and potentially uncovering structural problems, errors, and problems with specific components. I used this to test each that each function, constructor, accessor, and mutator worked correctly and as intended. To do this, I used assertions to compare the actual results with the expected results. For example, I was able to test that the constructor and the accessors worked at the same time on the Task.java program by creating an object with legal arguments. I then used an assertTrue statement to test that each attribute was correctly assigned and that the accessors for the attributes work correctly. This is done by comparing the string that is returned by using the accessor to the string that was supposed to be assigned to that attribute. This test is passed if both strings are identical. I did this for each attribute and accessor and used a variety of other assertions to check the rest of the program’s inner workings. To assure that all parts of the program were tested I checked the coverage percentage.

I also used experience based testing by using my experience on the first program dealing with the ContactTest.java and ContactServiceTest.java files and used the same techniques for both the TaskTest.java, the TaskServiceTest.java, and the AppointmentTest.java and AppointmentServiceTest.java files.

One testing techniques that I did not use was black box testing. This is testing the system by running it without any prior knowledge of its internal workings to identify how the system responds to user actions. The programs I created are not ready to test as a user since there is no main or inputs for the user to use. After these steps are completed, I would be able to use this technique on top of the ones I already used to successfully test the entire program.

Using all of these techniques helped me ensure that the program I made worked as intended. Both static and dynamic testing are essential to the complete testing of software. Static testing will make sure that there are no issues with spelling, syntax, and even logic. By reviewing the code, it will save money and time in the development of the software by cutting down on the costs of testing. Dynamic testing is done by running the code and finding structural problems or errors. Using black box testing allows the tester to use the program as a user and see what the results are. This will help make sure that the program works correctly for the user. White box testing allows the tester to find errors in specific parts of the code which can then be fixed and tested again. These tests ensure that the software works and does what the developer intends.

While working on this project I worked in the mindset as a software tester. I was constantly thinking about inputs and using exception throws to catch errors while writing the java files. Then, while writing the JUnit files I was going line by line to make sure I had full coverage over every single part of the code. I made sure to appreciate the complexity and interrelationships of the code and how they work together to complete the program. For example, the TaskService class file helps organizes the tasks from the Task class file. If the Task class had any defects, then those would be passed onto the TaskService. I employed caution by not believing that the code was bug free, I went into it assuming that everything I didn’t test has the chance to be defective. By working in this mindset, I was able to continue testing the program until I believed that every part of the code was tested.

While reviewing the code on several occasions I tried to limit my own bias by looking at the code as if I didn’t write it. I would read it character for character and make sure everything from syntax to logic was correct. I was able to catch a logic error early in the ContactService file. I created a getContact function to locate a contact based on its ID. Through review, I was able to find that I had the return inside the for loop and caused the function to not work properly. If I had glossed over that without looking at every character of code, I would not have found the error as early as I did.

It was also very important to remain disciplined and fully committed to quality. Using testing to cover your code as a software engineering professional is the only way to make sure that your software will function as you intend. Testing will make sure that the quality is of the highest caliber. There have been many examples in the past and there will be many more in the future where even the most established companies will release products with bugs and defects in them. There are a variety of reasons for this including but not limited to arrogance, incompetence, and laziness. It is not possible to fully ensure that all of every program is tested but making sure that the most important aspects are covered will result in a high quality software. All of this testing and these precautions will help prevent technical dept. It will ensure that shortcuts were not taken, and the program does what it intends to do.