CS 320 Project Two

Summary and Reflections Report

Joseph Caron

10/15/2024

**Summary**

As a software engineer for Grand Strand Systems, I recently developed and delivered some parts for a mobile application, namely the Contact Servies, Task Services and Appointment Services. During development of these features, I utilized JUnit tests to ensure that they complied with the customer requirements. One of the requirements, which was in place for all three features, was that each object created for each service, Customer, Task and Appointment, must have a unique ID of no more than 10 digits that could not be null and could not be changed after being initially issued. To meet this requirement, I used the Atomic Long java utility to issue a number for the ID, increment the counter after distributing the number, and then save the number for multithreading so that the next object created would be assigned with the next number in sequence. I also added in an IF statement that would throw an illegal argument exception if an ID number was tried that already existed in the data structure to make sure each number would be unique. To test that this worked, I created JUnit tests to test using NULL ID numbers. I also used the test ordering utility because each time an object was created for the tests, it would have the next sequential number assigned to it, so the tests needed to be run in a specific order to work correctly.

I know that the quality of my JUnit tests was adequate for the features being tested because I also used a coverage add-on in my IDE, which showed that all tests covered all classes with an average of over 90% coverage. The only things that missed being tested were attempting to use duplicate ID numbers, because of the nature of how it was coded making the program itself choose an ID number. I did, however, using manual testing to ensure that trying to use an ID number that was already in use properly threw the illegal argument exception. I made sure my code was technically sound and efficient by changing the data structure used to house the various objects from Array Lists to the more complex yet more efficient Map structure, using Map<String, Contact> contactMap = new HashMap<>();. I also improved the efficiency of my JUnit tests because originally, I had set up the service feature tests to each create their own instance of the service class. For instance, in my testEditFirstName() test, the first statement was ContactService serviceTest = new ContactService(); to instantiate a service object. Then, the first line of the testEditLastName() test was also ContactService serviceTest = new ContactService();**.** To improve efficiency, I changed it so that instantiating the service object happened as the very first statement in the ContactServiceTest class, so that it only needed to be created once for all tests, rather than for each.

**Reflection**

The testing techniques I used for these features were white-box testing methods. While ensuring that invalid input led to illegal argument exceptions as coded, I used Fault injections with JUnit tests. This was done by setting up the tests to attempt to run input that the program should consider invalid. I also used Code coverage testing, making sure that every branch of the code was tested to make sure that decisions were handled correctly and that there were no “dead-ends” which could have caused run-time errors or unexpected stops. According to Garcia (2017), there are many other testing methods that I did not use, such as black-box testing methods of Graphic User Interface testing, which tests the GUI of a program, or Model-based testing, in which tests are derived from a model which is itself derived from the requirements documentation. Neither of these were needed as the program has no GUI to test, and no model was made from the requirements documentation and instead the program was designed directly from the requirements documentation itself. There is also non-functional testing, such as Security testing, which tests the confidentiality and integrity of a system, which was also not a concern at this phase of the project.

I employed caution extensively throughout the creation of my program and the tests. At first, when creating all three features, I had an integer variable initialized to 0 as the unique ID for each object, Contact, Task and Appointment, the integer would then get converted to a String to be added to the object and used as the key in the map, at which point the integer variable would get incremented in anticipation for the next object to be created. The problem was this all occurred in the Service class rather than the actual object classes. I initially did this because I didn’t know how to have the system create a unique ID that could be incremented and easily tracked with it all taking place in the object classes themselves. After taking the time to do research, I found out about the java utility Atomic Long, which would allow a unique number to be incremented and saved to a memory that could be accessed through multiple threads. With this knowledge, I was able to transition the programs so the objects can create their own IDs rather than the services handling it.

Bias can be tricky when trying to test and validate your own code. As the code author, when manually parsing the code to look for errors and defects, it can be all too easy to overlook a glaring mistake. I had an issue in my Contact code when moving the ID creation method to the objects, as mentioned in the paragraph above, where the tests were throwing an unexpected error. It took far longer than I would care to admit to find out that I had accidentally left a String contactId parameter in the class declaration. I have to wonder if I would have caught the mistake faster had I not been the one to write it in the first place. Finally, discipline is invaluable to a software engineer. Cutting corners with code can easily lead to faults and errors, and as a program is written, it becomes harder and harder to find the problem areas after the fact. I have personally written work-arounds into my code in the past, and it always, without fail, leads to me needing to go back and read through every line of code to figure out why it isn’t working as I want it to later on. The time it has taken me to back track and fix my own stupidity and laziness has solidified to me that I should endeavor to always write as if it is a finished product from the get go.

References

Garcia, B., (2017). *Mastering Software Testing with JUnit 5: A Comprehensive, Hands-on Guide on Unit Testing Framework for Java Programming Language.* Packt Publishing. https://eds.p.ebscohost.com/eds/ebookviewer/ebook/bmxlYmtfXzE2MjY5NTBfX0FO0?sid=3d6cdffc-6fd9-4adc-a5c3-eb8fe792cfd4@redis&vid=0&format=EB&lpid=lp\_84&rid=0