**Project Two**

Joshua Brown

Department of Computer Science, Southern New Hampshire University

CS – 320: Software Test Automation & QA

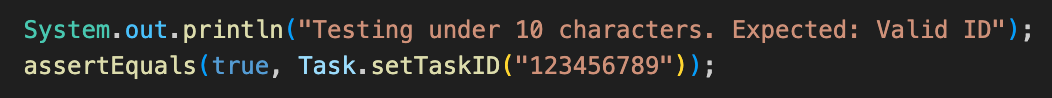
Professor Ramsey Kraya, Ph.D.

October 13, 2023

**Project Two**

**Summary**

* To what extent was your approach**aligned to the software requirements**? Support your claims with specific evidence.
  + For each class that I was tasked with creating there were necessary requirements for their completion. For the contact service class, I had to make sure that the elements of the contact class followed the specifications upon creation, such as a unique contact ID that was no longer than 10 characters, but at the same time not able to have a null value. My first attempt at doing this back in the milestone, I ended up creating a Contact Class that did not have any restrictions on the creation of a Contact object. I instead had the restriction on object creation placed in the Contact Service class, as individual functions, to check to make sure that the field being created for contact met the requirements, and then added these field checkers to the add contact function in the Contact Service class. Though the end result was the same, it did not follow the exact request of the software requirements. This was especially evident when testing the Contact class, where I had to incorporate functions from the Contact Service class in order to pass the test for the Contact class. After completing this milestone, and working on the other two, I had the opportunity to go back and reassess the code for the Contact and Contact Service classes and refactor them so they would match the software requirements exactly the same as with the Task and Appointment classes. Doing so allowed the test for each class to be a more genuine check to ensure that each element of the classes properly passed the software requirements, and were not dependent upon other functions to do so.



* Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were **effective** based on the coverage percentage?
  + Though the Junit test did not have 100 percent test coverage (which is not possible), they did test the priorities, which in this case were the requirements for the classes, which were tested effectively by testing each border case for each field. Example: for the String TaskID, the requirements were that the string be no more than 10 characters. As a result, I tested 9 characters, 10 characters, and 11 characters with the assumption that the test for 9 and 10 would pass and the test for 11 would fail.
* How did you ensure that your code was technically sound? Cite specific lines of code from your tests to illustrate.
  + For certain tests like the add Task test, which was for the Task Service file, there was a need to call on multiple functions from the Task file. It is through these tests, that you are really able to understand how well your code works as a whole. If one of the component functions fails, then the main function being tested will fail as well. To make sure everything worked properly, I made sure to print out the result for all component pieces, so I would know which component test caused the main test to fail.

A close-up of a white background

Description automatically generated

* How did you ensure that your code was efficient? Cite specific lines of code from your tests to illustrate.
  + After I started creating the test cases, I realized that there were areas that I could improve when writing the code. For example, rather than return void for a function that I didn’t need any output from, I could return a Boolean which could further help with my test cases. For testing's sake, I also chose to print to the screen what I’d expect from the test, as well as the results; this helped to quickly flag down where the test failed and what caused it to fail.

A white screen with black text

Description automatically generated

**Reflection: Testing Techniques**

* What were the software testing techniques that you employed for each of the milestones? Describe their characteristics using specific details.
  + The testing technique that I used for the developed Contact, Task, and Appointments classes has all been black-box testing, specifically testing for boundary values. Blacked-box testing will primarily test the system requirements. For each of the milestones, the requirements for each class were specified; for example, a variable cannot be null and must be within a certain character limit. By writing code to meet these specifications and testing to make sure the test passes the requirements we are essentially creating black-box tests.
  + One type of black-box test that I used for most of the assignments was boundary testing. Boundary testing is essentially testing the edges of the partition for a parameter. Say for example a function accepts an integer for a parameter within the range of 1 through 10, with 1 and 10 included in the range. In this case, there is both an upper and lower partition and both should be tested with at least 3 values; the upper partition should test the above value of 11 (which is outside of the partition), the value of 10 (which is on the edge of the partition), and the value of 9 (which is right before the partition). Because errors tend to occur at and around the boundaries of partitions this form of testing is well-suited to test these kinds of requirements.
* What are the other software testing techniques that you did not use for the milestones? Describe their characteristics using specific details.
  + A testing technique that was not utilized was White-box Testing. While Black-box testing seeks to test that the code meets the requirements, White-Box (or Structure-base) Testing, tests the structural components and the integration of the system. This form of testing uses tests like statement testing, decision testing, and branch testing to test the system as a whole and how its parts interact with one another. White-box testing could very well be used to test the application as a whole once we start piecing together the components of the individual classes. For example, if we were to combine the three services (Contact, Task, and Appointments) and use them in an application for a doctor’s office to create patient appointments, then we could use White-box testing to ensure the parts come together seamlessly.
  + Another form of testing technique that was not employed in the assignment was Experience-based testing, which is testing that fully utilizes a tester's experience to create test cases. These tests include error guessing, exploratory testing, and checklist-based testing. As the name implies this form of testing requires knowledge of testing techniques, procedures, and known areas of common errors. It would more commonly be used in a situation when the requirements for a system aren’t entirely clear.

**Reflection: Mindset**

* Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ **caution**? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.
  + As I progressed through the milestones I started to become more familiar with software testing, and with each task that I completed I was able to go back and identify areas that I could improve when it comes to my software testing technique. For example, after doing the Contact class I noticed the need to modify my code so that the software test could directly test the functions that it was supposed to be testing. After completing the Task class, I noticed that it would be a good idea to add confirmative tests after certain test actions were done, such as testing to make sure an object still exists after deletion. And lastly; after doing the Appointment class I realized I tended to test the same sequence of items and go through the same steps upon testing the elements of a class; I realized here that I was beginning to develop a bad habit, and went to try and vary my tests to make sure I wasn’t always testing the same things. Doing so allowed me to catch some critical errors in my code that I would have never noticed before, due to me continuously using the same testing procedures over the span of all my tests.
* Assess the ways you tried to limit **bias** in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims.
  + One way I tried to limit the bias for testing my own code, was to test the code on a different day that I wrote the code. I found during these assignments that after writing the code, I was normally in a relaxed state of mind and had a sense of a job well done. As a result, I was more liable to miss critical errors in my testing, and or write tests that did not fully test the requirements, which was the situation I experienced with the Contact and Contact Services classes. After writing the code for these classes, I immediately proceeded to write the tests for the classes. It was then that I noticed that my code could not pass the test if they tested each component individually, I stubbornly defended my code by incorporating the other functions in the test so that it would pass instead of refactoring my code as I should have. Though I did go back and fix the code in the end, I still acknowledge the issue with testing one’s own code, and how it can lead to one not wanting to essentially tear down everything they spent hard work building up; however, to test one’s own code, there is essentially a need to figuratively distance yourself from your own creation and look upon it as if you are seeing it for the first time.
* Finally, evaluate the importance of being **disciplined** in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.
  + Software and computer systems have become key elements in the functioning of society. As software engineers, we must realize that the code we write will have an effect on people’s everyday lives; cutting corners when it comes to writing code or testing it could very well result in tremendous financial loss, the complete disintegration of a company’s reputation, and even the needless loss of lives. I plan to avoid technical debt by approaching every project where I write or test code like there is someone’s livelihood on the line, because whether I see it or not, it likely is.