Summary and Reflections Report

To what extent was your testing approach aligned to the software requirements? Support your claims with specific evidence.

I believe my testing approach was directly aligned to the software requirements. For example, one of the specified Task/Appointment/Contact class requirements were that when an object is created, its ID cannot be null nor exceed a certain character limit. My first JUnit tests for each module verified that when these specifications were met, the objects were created. Additionally, negative tests were also written to verify that when illegal arguments were used to initialize an object, a null pointer or illegal argument exception was thrown, ensuring the associated object was not created.

Defend the overall quality of your JUnit tests for the contact service and task service. In other words, how do you know that your JUnit tests were effective on the basis of coverage percentage?

Using the Eclipse IDE, I was able to run “Coverage As” for a given JUnit test class. Then, in the “Coverage” tab at the bottom of the application, I was able to verify there was a 100.0 % coverage value for the associated class being tested, ensuring the JUnit tests were fully effective.

How did you ensure that your code was technically sound? Cite specific lines of code from your tests to illustrate.

I first ensured my code was technically sound by verifying there were no syntax errors. I used the Eclipse IDE, which underlined these errors in red. Secondly, I verified that the functions in my classes met the defined requirements (i.e. not creating objects if a user attempts to initialize with invalid member data). I also ensured to set the member data variables to private so they could only be set/updated through the constructor and update functions. Lastly, JUnit tests were written for each module class to ensure they functioned appropriately when both valid and invalid initializations were made.

How did you ensure that your code was efficient? Cite specific lines of code from your tests to illustrate.

I ensured my code was efficient by eliminating redundancy. For example, each module had a requirement that certain member data be updatable if valid (i.e. an Appointment description), requiring setter functions. In order to minimize code redundancy in the class constructor, these setter functions were called instead of their code being repeated (lines 21 and 22 in Appointment.java).

What were the software testing techniques that you employed in this project? Describe their characteristics using specific details.

One testing technique I employed in each of the milestones was equivalence partitioning, particularly for testing string lengths for IDs, names, and descriptions. For example, the appointment description could not be above 50 characters. Thus, one test initialized a description with a value between 0-50, and the other above 50. Since both of these inputs resulted in expected behavior, it could be assumed that all other numbers within these intervals would produce corresponding results. Another complementary technique used was negative testing, where an attempt was made to initialize objects with invalid values. Using assertions, it was then verified that the code threw an exception.

What are the other software testing techniques that you did not use for the milestones? Describe their characteristics using specific details.

One software technique I did not use in the milestone was boundary analysis. This is where the code is tested at the exact boundary values and is verified to work as expected. For example, in the task service assignment, a task ID could not be above 10 characters, so a boundary value test would initialize the IDs with values 10 and 11. In this case, it would be verified that when 10 characters are used, the object gets created, but for a length of 11, it throws an exception.

For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.

In my experience as a tester, equivalence partitioning has been the more practical testing technique. This is because the software being tested is large and testing every possible combination of the system is impossible. Negative testing is also a practical technique to ensure your software has safeguards against improper uses of the system. However, with limited resources, this technique may not be a priority. In these milestones, since Java was the selected programming language, assertions were used to verify that the code functioned as expected when equivalence partitioning and negative test cases were made. However, since there are plenty of other programming languages, their implementation for these tests would most likely be different.

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.

Caution and recognizing the complexity and interrelationships of code is important in software testing, because the different number of combinations the software can be used in is an astronomical number. For example, each of the contact, task, and appointment service classes were embedded with contact, task, and appointment objects, complexifying the code. However, employing a number of testing tactics, such as equivalence partitioning and negative testing, helped ensure the software was verified to function properly. Additionally, tracking the testing coverage of different test cases can be done through an IDE, which highlights the code covered in the 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.

Bias is definitely a concern during the code creation stage as a developer. As a tester, there have been a few occasions where I’ve taken on development tasks. While completing these tasks, there were points where I was confident my code was correct, resulting in insufficient unit tests being made, only for bugs to be found later on. One way to counteract this is to use an IDE that allows you to create unit tests and verify the coverage rate. In this way, you can verify that your unit tests fully test your code (if they have a 100% coverage rate).

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.

Commitment to quality in coding is an important value for every software developer to have. It’s important to not cut corners because the later a bug is found, the more time and resources it takes to fix it. This is especially true if the code with the bug is integrated with other software features. I plan to avoid this by utilizing an IDE to ensure my test coverage reaches a certain threshold, and to utilize a number of testing techniques, such as equivalence testing, to ensure tests are effectively written and verify proper functionality against multiple use cases.