**Project Two**

Summary

My unit testing approach was aligned with the software requirements as much as possible because the requirements were the foundation for both the program and the tests used to exercise it. For example, for the Contact, Task, and Appointment classes, each instance variable had their own requirements, such as they cannot be null or must be a certain length, and the tests checked for these conditions to verify the input. This also included making sure that appointments were not scheduled with a date in the past. ContactService, TaskService, and AppointmentService similarly had requirements as well, like needing to add, delete, search, or modify the objects from the data structure holding the them. All tests made sure that objects were correctly added to the structure and that the required actions could be performed on them. Another part of aligning to software requirements was making sure that I did not add any extra functionalities that were not asked for, such as a user interface.

I know that the tests were effective because the overall quality of my tests was improved in the project submission as compared to the milestone submissions. For the milestones, the tests for object classes (ContactTest, TaskTest, and AppointmentTest) covered only around 80% of the object classes themselves which I thought was okay because the tests for the service classes covered the parts of the code that were previously missed. However, I realized that my program could be refactored to be more efficient so after looking at the provided example and implementing code based on some of the techniques it used, I was able to achieve 100% coverage for all classes. Based on the total coverage and the improvement from the milestone submissions, I am confident that the unit tests were effective in testing the required functionality of the program.

I ensured that my code was technically sound by making sure that I followed best practices and appropriate coding conventions when writing the classes and their tests. Each class had its own instance variables which were all declared as private, and helper methods used only within the class were made private as well. This way gaining access or making changes from other classes could only be done by using public functions like setters and getters, ensuring no unintended alterations could be made. The hash maps for each service class were private as well so only the class itself could perform actions on the data structure and the objects within. The code was free of any critical errors or warnings, constructors were used, and comments were included to describe steps and processes.

I ensured that my code was efficient by implementing only the functionalities that were required and not adding any variables, classes, or libraries that were not needed. Repeated code was avoided by creating and calling on functions that could take different parameters, for example using one validate input function to check multiple variables. I also avoided repeated code in tests by creating a global service object instead of doing so for each test, or by only testing things once. No lines of unreached or unused code were used.

Reflection

The software testing techniques that I used consisted of performing JUnit tests and were mostly specification-based. By using the JUnit framework I was able to easily create automated and repeatable tests, making use of assertions and exceptions to make sure the code was functioning correctly. I used equivalence partitioning by using one input from a group of similar inputs as a representative, eliminating the need to test all other inputs of the same type. For example, when checking the appointment dates I only needed to test one date that was in the past as opposed to checking every date before today. I also used boundary value analysis, since that is where errors often occur, when verifying the length of an input by checking right at the upper boundary instead of a random large number. I did not use certain specification-based techniques like decision table testing to determine cause and effect, state transition testing to test changes in state, or use case testing to see how the system would work for a user because they were not necessary in this particular case. I also did not really use any structure-based techniques though I did make sure that decisions were exercised and that code had adequate coverage, and I used the compiler to check for errors and warnings before code was executed.

There are many practical uses and implications for these software testing techniques. For example, equivalence partitioning can be used to greatly reduce the amount of input testing needs to be done which saves ton both time and cost. Other partition testing can be done as well like for output or for other functions more specific to a program. Boundary value analysis is useful because errors often occur around the boundaries, even if the rest of the input further within or outside of the boundaries functions correctly. An example is accepting input from 0-100 and making sure that 99 and 100 are valid, and that 101 is not. Like with equivalence partitioning, using this technique to find these errors quickly saves both time and money.

The mindset that I adopted while working on the project was one of objectivity, specificity, and efficiency because I believe that a developer with these three qualities can take a program a long way. Caution is also very important, making sure that each requirement is tested thoroughly and completely. Caution also include understanding how the code works and how it interacts because without a solid understanding of what is going on behind the scenes it can be more difficult to perform tests and they won’t be as efficient or as effective as they should be. An example could be the relationship between the object and the object service classes because the service classes can’t be fully tested without accessing methods from the object class even though the object class isn’t directly imported into the test class.

Caution also carries over to bias. When testing one’s own code it is difficult to be completely free of bias so that is something that a developer needs to be aware of. Without an objective view of the program and its testing, errors can slip through the cracks because the tester might subconsciously not view the testing phase as trying to find defects. A developer, for example, may be more inclined to perform tests that are more likely to behave according to the way they are expected to. Using an outside source to review code can be helpful because they can approach it with the mindset of actively trying to expose problems. Therefore, bias is a concern that should be acknowledged when one’s own code is being tested.

Besides bias, being disciplined in commitment to quality is also extremely important. It’s important to not cut corners because this reduces the quality of the code and makes it more likely that problems will arise. Additionally, this gets in the way of maintainability, which means that any updates to the program will be that much more costly to carry out. If a developer cuts corners it is also likely that someone will need to fix the issue later on, adding to the cost. In order to avoid technical debt in the future, I plan on managing my time as deliberately as possible and doing the proper planning before writing a program, including creating flowcharts and/or pseudocode. When writing the code, I will use coding best practices and follow object oriented principles so that I can promote quality assurance.

References

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter. (2015). Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition). BCS The Chartered Institute for IT. https://app.knovel.com/hotlink/toc/id:kpSTAIST01/software-testing-an-istqb/software-testing-an-istqb