**Project Two: Summary and Reflection Report**

Shaun Ryan

Southern New Hampshire University

CS-320: Software Testing, Automation, and Quality Assurance

Dr. Cross

06/19/2022

The unit testing approach for the three features of this app were aligned with the software requirements. The requirements were used to develop the unit tests and care was taken to develop the tests in a way which covered all aspects of those requirements. For example, some of the requirements for the Contact Class were that the first and last name strings could not be longer than ten characters and could not be null. These requirements were broken down into four separate tests in the Contact Test class; one for each variable being set to over ten characters and one for each variable being set to null. By writing the unit tests to specifically address each requirement, I ensured that the tests were aligned with the intent of each requirement.

The quality of the unit tests improved over time as I became more familiar with the procedure and comfortable structuring the tests. The process of running the tests as I developed them helped to find not only issues with the code being tested, but the tests themselves. Coverage was an invaluable tool in determining how effective my tests were and identifying areas where they needed to be expanded. After the final revision, the project had an 87.9% total coverage, with the package being 98.7% covered. I feel that this is adequate for assuring the code is sufficiently tested. The class with the lowest coverage percentage was the Appointment class at 93.0%. Within the logic trees of the set methods, the final branch after error catching, which serves to set the data, was not covered. Unit tests to ensure the data within an instance changed could have been implemented to increase the total coverage, however I felt that the tests concerning the failure conditions were more pertinent to the requirements for the class.

While test coverage is an important factor in determining the effectiveness of the unit tests, it is only one piece of the picture. It doesn’t matter if a section of code is covered if the test isn’t technically sound. Hambling et al. state that boundary value analysis is important because the most likely faults will be values just inside or outside the accepted range (2015). When I first wrote the tests for data fields being longer than the maximum allowed value, I used arguments of an arbitrary length so long as it was longer than the maximum allowed value. Upon later revisions, I changed this so the arguments would be a single character longer than allowed, thereby analyzing the boundary value. This could catch a hypothetical fault due to the use of a “greater than” as opposed to a “greater than or equal to” in the classes error-catching logic.

The efficiency of the tests was an area where I was able to improve through revision and experience. Tests should be granular enough that a failure narrows down what the specific problem for the programmer. While it is generally good practice to have a single assert per test, there are scenarios where multiple asserts can be combined which test the same logical object. In the Appointment Test class, there is a requirement for the appointment description to not be greater than 50 characters. There are two ways of setting the description: through the constructor and through the set method. I created the unit test for this requirement to assert that both the constructor and the set method throw an exception if a value is entered with is too large. While there may not be a practical advantage to this approach from a purely performance perspective, I believe that it makes the code more readable and therefore more efficient to the programmer.

The testing techniques used were primarily specifications-based (or black box) techniques, which are developed based on the documented requirements for the classes. This includes both functional and non-functional aspects of the project. They were not written using any specific knowledge of the internal structure of the component or system under test. All the methods utilized were public, and the assertions tested return values of the functions, exceptions thrown, or values of publicly accessible data structures.

A testing technique which was not utilized in this project was structure-based (or white box) techniques. These would be used to evaluate the components of the application at a structural level. All the tests performed were conducted via the inputs and outputs of public methods without any consideration or required knowledge of how they are performed. As this application matures and the data structures which hold the information for each class become more complex, white box techniques could become more useful in testing these data structures effectively. Another technique which was not used in this project was experience-based testing. Hambling et al. explains that experience-based techniques rely on the testers intuition and experience with similar systems and include error guessing and exploratory testing (2015). As the size and complexity of the application grows, these may become more useful, but for the current classes, specification-based testing is sufficient.

The three testing techniques mentioned each have strengths in different situations. For applications where the requirements are clearly outlined, black box testing is the logical choice. This was the case for the classes in this project, as the requirements clearly stated what outputs should be expected for user inputs. Had the internal workings of the classes been more complicated, white box techniques would be appropriate. Where a black box test would tell the tester whether the component produced the correct or incorrect output, white box testing can be used to interact with the structure of the components and narrow down where an error may be occurring within that component. Experience based testing can be useful in situations where the component does not have clearly defined requirements. For example, if the requirements stated simply that the application had to accept a password for a new user account, that could be interpreted in a variety of ways. Experience with similar features in a myriad of other programs would inform the tests we create. Issues such as case-sensitivity, illegal characters, and required character types would be tested, even if they aren’t explicitly called out in the requirements.

Throughout this project, my mindset was one of learning and improvement through iteration. Being the first time I have worked with unit tests, I initially concentrated on creating tests which would ensure the code satisfied the requirements. As I continued to learn through class reading, external references, and hands-on experience, I revised my previous tests to address clarity and efficiency in addition to purely addressing the requirements. Caution was applied when assessing coverage. Where a high percentage could give a false sense of security, I endeavored to make sure my tests were technically sound and thorough.

When it comes to testing your own code, bias is an issue that you need to be aware of and actively work to counteract. Initially I only tested the constructor and neglected to test the set methods for each attribute. This was partly due to inexperience, but also due to the assumption that the individual set methods would be sound so long as the constructor was sound. This type of bias can lead to ineffective testing and undiscovered errors. One method which I found helpful for mitigating bias was to not write the classes and tests in the same sitting. I would write the code for the classes early in the week, then write the tests for those classes a couple of days later. This helped me approach them with a fresh set of eyes and avoid assumptions I might have made had I worked on them concurrently.

It is my belief that attention to detail is the greatest trait of a software engineer, and that applies to both programming as well as testing and quality assurance. Robust requirement development, meticulous programming, and early testing takes extra time and effort but will be beneficial over the course of development (Hambling et al., 2015). Cutting corners, on the other hand, may lead to short term productivity gains but can ultimately lead to accruing large amounts of technical debt and having to make extensive rewrites late in development and numerous hours spent troubleshooting. In addition to frequent periodic testing, my plan to avoid technical debt can be distilled down to robust planning. By determining the requirements, features, dataflow, and behaviors of a class before development, the coding process will be more efficient (Valacich & George, 2019). An example for this application would be if we wanted to have the ability to attach a certain task to an appointment. Developing the data structures for these class with that functionality in mind from the start would be easier and lead to less defects than developing them independently and trying to integrate them later. While UML diagrams may seem time consuming and tedious, it is my responsibility as a software engineer to approach development in an ordered and professional manner.

References

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter. (2015). *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition) - 1.4.2 Testing and Risk.* BCS The Chartered Institute for IT. Retrieved from   
https://app.knovel.com/hotlink/pdf/id:kt00UC2GM2/software-testing-an-istqb/testing-and-risk

Valacich, J. S., & George, J. F. (2019). *Modern Systems Analysis and Design* (9th Edition). Pearson Education (US). https://mbsdirect.vitalsource.com/books/9780135172841