**Summary and Reflections Report**

Samuel Meade

Southern New Hampshire University

[CS-320-T6629 Software Test Automation& QA 22EW6](https://learn.snhu.edu/d2l/home/1116032)

L. Gaston

August 14, 2022

Summary

For this project I was tasked with completing the contact, task, and appointment back-end service deliverables. To address these deliverables, I studied the user requirements and analyzed what was needed in each portion. Before implementing the requirements fully, I took advantage of a continuous integration technique to build out a rough structure for the application that would run with the least amount of code. Once I had functionality and a running application, I slowly began to incorporate the user requirements, ensuring to run my code with each modification. For example, in my Contact, Task, and Application classes, I wrote a generic constructor without any validations. Once that constructor was successfully running, I adapted it to include the validations seen on line 56-71 of the Contact class in compliance with user requirements.

Implementing the JUnit tests was a similar approach to the development of my classes. For the JUnit tests, I prioritized user requirements before writing test-cases to cover code quality. My JUnit tests originally tested for the bare minimum user requirements before writing a few more tests to more fully cover bounds and limits. My test cases for the Contact, Task, and Application classes were straightforward – test for the constructor and the getters and setters to work as expected, and ensure they work within the user specifications. These classes were light, so the testing for them was not overly complex. For the service classes, the tests were a little more thorough. I tested each method to ensure unique IDs could not be duplicated, invalid IDs could not be deleted, errors were being thrown where expected, and our data structures were accurately reflecting the information provided.

Writing the JUnit tests felt intuitive – cover cases that may cause your code to fail. The coverage checker within IntelliJ and Eclipse is a valuable tool to ensure your tests are thorough, covering the base cases for all your methods. Once those coverages are met is where testing becomes a bit trickier. A lot of code breaks due to unforeseen circumstances and outliers, testing for which may seem challenging. Something I could have done to cover my code more thoroughly would be to test for wider ranges around my boundaries. It is better practice to take the time to test cases over, at, and under boundaries, and even more effective to test some wider variants.

Unfortunately, there is no coverage checker for code efficiency. To address efficiency, I like to analyze my code for excessive and unnecessary blocks that do not relate to the project’s requirements. After combing through my Classes, all the code is written with intent and commented in a manner that is informative without being overbearing. The code is written in a readable structure for maintainability, avoiding complex blocks that may be hard to follow.

Reflection

The testing techniques I employed in this project were JUnit Testing, Continuous Integration, and Acceptance Testing. The JUnit and Continuous Integration techniques are dynamic forms of testing, implementing them as you program. Acceptance testing is a more static approach, gathering user requirements in the planning phase, earlier in the Software Development Life Cycle. While these tests seemed to be successful for me, there are plenty of additional techniques that may have provided value to my SDLC.

Static testing techniques are powerful resources early in development. Utilizing visuals like UMLs to hash out designs and relationships speeds up the development process and helps keep a team of developers on the same page. These visuals are also great resources for non-developer team members, allowing them to participate in the design phase without understanding how to write code. Along with UMLs, code reviews are also a great static testing technique. Code reviews vary in formality, ranging from pair programming reviews to technical lead reviews (Hambling et al. 2015). These reviews are great opportunities for improvement, as well as another filter before code reaches deployment. Code reviews may catch bugs that evaded the developer, alleviating risk for the release.

A dynamic testing technique that I did not employ is test driven development. After gathering user requirements, the developer can then take those requirements and write out the test cases before touching the actual application. This encourages the developer to write component-based code to meet the test cases, in turn meeting the user requirements. This strategy would have been a great match for this assignment because the user requirements were gathered early on in a descriptive manner.

I was not overly cautious in my testing techniques. I wrote tests that covered my code to the bare minimum of a release. I spoke earlier about the ways that those tests could be improved to be more thorough. Though these tests may not cover all kinds of outliers, they are well covered for the purposes of the assignment and an initial minimal viable product. This assignment clarified the potential of testing and how in depth it could be. I had trouble understanding how one could “over test” before this assignment but can now see the potential to get lost in the process of covering your code.

To limit bias, I took time away from my computer to re-evaluate my code with fresh eyes. I write code with intent, trying to avoid redundancies and poor practice, but nobody is perfect. Leaving my code and returning to it later in the day provided me with more clarity and less attachment. This allowed me to see potential issues and areas where more coverage was necessary. Bias will always be present, especially within larger team projects. It is important to separate yourself from the code, keeping the deliverable and user requirements as the priority. While writing tests for your own code may feel easy because of your comprehension, having more eyes on it maintains accountability and quality of the deliverable.

At the end of the day, accountability is everything. Holding steadfast to writing ethical, quality code is the job of the developer. This may be in the form of code quality, redundance, efficiency, and testing. Releasing bugged code can be detrimental to both the client and to civilians. Depending on the software, inferior quality can be fatal. Mitigating these risks through thorough testing and collaborative techniques saves money, lives, and is the responsibility of every developer.

Resources

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. Retrieved from  
https://app.knovel.com/hotlink/toc/id:kpSTAIST01/software-testing-an-istqb/software-testing-an-istqb