James Lee

CS-320-T4208 Software Testing, Automation, & QA 23EW4

April 11, 2023

**Project Two – Summary and Reflections Report**

I recently completed a project that entailed developing and delivering contact service, task service, and appointment service features for a mobile application for a client. As part of the delivery, I developed a unit test for each feature to verify that each meets the client’s requirements. Below is my report of summary and reflections regarding the project, on my unit testing approach, my experiences writing JUnit tests, and the overall quality of my JUnit tests.

**Summary**

The level of effort in software testing varies from project to project, depending on factors such as the complexity of the software, client/stakeholder involvement, and the level of effort in the software development process. In general, my approach to software testing aligns closely with the software requirements to ensure that all specified functionalities are verified and tested. Having said that, other aspect such as the software security is also considered in my testing approach – i.e., the adherence to the requirement should not compromise the security of the software. As stated in the Association for Computing Machinery (ACM) Code of Ethics, one of the professional responsibilities of a computing professional is to “[d]esign and implement systems that are robustly and usably secure” (ACM Code 2018 Task Force, 2018).

Other aspect such as the software performance is also considered in my software testing approach. For my project, this meant going beyond the requirements to ensure the software behaves as intended. For example, the Appointment Service feature required the ability to create appointments with dates. The only requirements specified were that date cannot be in the past, and the date field cannot be null. However, from my experience I knew that I needed to code decision branches for checking appropriate dates for each month, as February has 28 or 29 days (leap year) while other months have 30 or 31 days. This also required testing all decision branches to verify proper software functionalities.

The effectiveness of my JUnit tests is evident by the coverage percentage of the tests – all of my JUnit tests achieved 100% or near 100% code coverage. In addition, the quality of the test cases and the accuracy of the test scenarios are indications of effective testing. For my JUnit tests, I analyzed each of the feature requirements to identify the boundary cases to develop tests that are efficient and effective. For example, for the task service feature, the task object required a unique task ID that is not null nor longer than 10 characters. This required developing test scenarios where the task ID is 10 characters long, as well as longer than 10 characters, to verify the software behaved as specified in the requirements.

Analyzing the requirements and developing proper test scenarios, such as in the example provided above, is one way I ensured the technical soundness of my code. In addition, adhering to the best software development practices such as following the proper coding conventions and reviewing code are other ways in which I ensured my code is technically sound.

Ensuring the efficiency of the code can be achieved in various ways. In my case, I achieved efficiency in my code by analyzing the requirements to identify the boundary cases to be tested, which allowed the code to be comprehensively tested with the least amount of test cases. Another way I ensured that my code was efficient is by performing code reviews to optimize the algorithms and remove redundancies in the code.

**Reflection**

For this project, I employed black-box testing, white-box testing, and experience-based testing techniques. I used black-box testing, specifically boundary value technique, to derive test cases for the unit tests from the project requirements. For example, for the contact service feature, the unique contact ID string had to be no longer than 10 characters in length. This yielded in the upper boundary values of 10 and 11, where 10 is the highest value within the boundary and 11 is the lowest value outside the boundary. I utilized white-box testing and experience-based testing for the appointment service feature. The appointment service required date field that could not be null nor be in the past. The requirement did not state that only the correct dates could be accepted as inputs, but I knew from experience that my unit tests needed to verify that the software accepted only the valid date inputs (e.g., February 30th is not a valid date entry).

One of the testing techniques I did not employ for this project is the decision table testing technique. This requires an analysis to identify the business rules from the requirements, to then identify the conditions each rule required to generation actions. Since there were no business rule requirements for this project, this technique was not applicable. In addition to this, the white-box testing was used only at the component level for each of the three features. This was because the three features have not been integrated into a system, and therefore the overall system cannot be tested using the white-box testing.

Black-box testing techniques are suitable when the software’s required behaviors are clearly specified and the test basis are well defined. White-box testing techniques are suitable when there is limited documentation available for the developers. In this case, the structure of the software including the statements, functions, decisions, etc., can be used to derive the test cases. Finally, the experience-based testing techniques are employed when the test cases cannot be derived from the specifications. This requires collaboration between the users and the developers to have a deep understanding and identify the critical components of the system that need to be tested.

For this project, I employed healthy amount of caution in my testing process. It was important for me to analyze the code carefully having tester’s mindset in order to identify all possible program behaviors that needed to be tested and verified. To reiterate the appointment service feature example from earlier, I needed to carefully determine all possible acceptable dates in order to determine the test-fail scenarios. Suppose that I wasn’t cautious in my test case development, and did not account for leap years in my testing scenarios. I would have then tested entering February 29th to verify that the program throws an exception. However, that would mean that the program cannot create appointments for February 29th, 2024 despite this date being valid since 2024 is a leap year. In addition, employing caution helps me as a software tester to avoid cases of the “pesticide paradox” – continually running same set of test cases decreases the effectiveness of the tests because the developers figure out to test for these cases before delivering the software (Hambling, Morgan, Samaroo, Thompson, & Williams, 2019, pg. 22).

One of the ways I tried to limit bias in review of my code is by taking many breaks in between developing the code. This helped me to revisit the previously written code with a fresh set of eyes and mindset, so as to limit the amount of previous context. There are certain advantages in having software developers test their own code, such as the ability to promptly fix the bug as soon as it is discovered. Conversely, one of the disadvantages is the bias that comes into play – often times the flaws or bugs in the code are not so apparent to the developer due to their having a different mindset than that of the testers. It is much harder for developers to be critical of their own code.

As a software engineering professional, it is important for me to remain disciplined in my commitment to quality. This is because even a single bug, despite how insignificant it seems, can lead to a detrimental outcome. Case in point – Compound, a decentralized finance staking protocol, mistakenly handed out about $90 millions worth of crypto tokens to users as a result of a “one-letter bug” in an upgrade that they released (Sigalos, 2021). Released defects, such as the one in the Compound example, lead to incurring significant technical debt. The cost of correcting defects during the live running stage is thousand times that of correcting the defects during the program testing stage (Hambling, Morgan, Samaroo, Thompson, & Williams, 2019, pg. 20). In fact, the technical debt increases exponentially as the software development progresses through its various stages. As a software engineering professional, I can avoid technical debt by remaining vigilant, avoiding complacency, and avoiding cutting corners in writing and testing codes.

References

ACM Code 2018 Task Force. (2018). *ACM Code of Ethics and Professional Conduct*. Association for Computing Machinery. <https://www.acm.org/code-of-ethics>

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing: An ISTQB-BCS certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.

Sigalos, M. (2021). *DeFi bug accidentally gives $90 million to users, founder begs them to return it. CNBC*. https://www.cnbc.com/2021/10/01/defi-protocol-compound-mistakenly-gives-away-millions-to-users.html?\_\_source=androidappshare