Domingo Polonia

CS-320-T2646 Software Test Automation & QA

Southern New Hampshire University

December 19, 2021

The summary and reflections report for the mobile application contain the unit testing approach, experience writing JUnit tests, and quality of JUnit tests for the contact, task and appointment services requested. This report explains the analysis of various approaches to software testing based on requirements and the application of appropriate testing strategies to meet the requirements while developing the mobile application.

Requirements is the first step to producing test cases. Without requirements, it is not possible to design or implement tests. For the mobile application, the software included only what was written in the list of requirements. For example, one requirement was: the contact object shall have a required unique contact ID String that cannot be longer than 10 characters. The contact ID shall not be null and shall not be updatable. This is just one requirement of the project specification.

A screenshot of a computer

Description automatically generated with medium confidence

After implementing the requirements for testing, the JUnit tests were run, calculating the coverage, to ensure the overall quality of the JUnit tests. By running the JUnit test with coverage, I can surmise how well the tests have covered the SUT by confirming every method and instruction has been touched and tested. As can be observed, the ContactService JUnit test covers all of the ContactService class file. This is a perfect basis unto which I can conclude that the tests are well written and that every method and instruction is well covered by the test.

A screenshot of a computer

Description automatically generated with medium confidence

Testing code is a method that can be used to ensure code is technically sound. Take the Contact constructor method for example. As observed, all of the conditionals have inline exception checks that are thrown if the checks do not pass. These could have been written in separate methods to be called upon. Instead of largening the codebase, I chose to write these inline as the code would be better written and more technically sound, since the code is called upon as the Contact object is being constructed.

A screenshot of a computer

Description automatically generated with medium confidence

Code efficiency can be produced by removing unnecessary or redundant code. Within the confines of the ContactService and TaskService class files, I have decided to use a HashMap data structure to hold the contacts and task objects. My reasoning behind this is because for a HashMap all runtime operations can achieve O(1) and do not require you to iterate over it to search and update an entry into it. They can be accessed with a single line of code. To update a value inside the map all I need to do is instantiate another object and throw it into the table with the updated values that I want and assign it the same key of the element I want to change.

Text

Description automatically generated

There were multiple testing techniques used throughout the project. One of the software testing techniques implemented was the single responsibility principle, stating that every class should be responsible for a single functionality. In other words, every test case should only work out the given system requirement of the SUT per test case. Another software testing technique, and possibly one of the most significant, is keeping tests simple. Test cases should be a sample of the SUT, not a test of the system as a whole in a single test case. This can cause recursive issues out of logic and result in meta-testing, which you want to avoid because it can require additional time. This will keep tests clean and easy to maintain.

Besides the testing techniques that were used in the project, there were others that were excluded. For example, although the tests run quickly due to the short test case, this would need to be managed for a much more complex SUT as time is an essential factor in testing, and therefore not included in this project. Another technique that was not used was testing solely for the sake of coverage. Just because some tests have touched every part of code does not mean that they have been thoroughly tested. Tests should be analyzed in terms of risk reduction for this to hold true. It is not necessary in this project since requirements and test cases are specified.

For the techniques listed, one that immediately stands out for practical use is that testing should not be done solely for the sake of coverage. When it comes to a SUT with many intricate parts and designs that rely on many different inputs, one must be sure that those parts are thoroughly analyzed for risk reduction and apply the tests accordingly. Having each test case have a single responsibility while running tests can also reduce the risk of running into errors or defects while testing, saving time during the testing phase. Time management is critical regarding testing because time is usually limited. Not implementing simple tests will save time because testing dummy code such as setters and getters is a waste of time, and there is no need to assess these parts of the codebase.

The project begins and ends with the consumer. Because the requirements and test cases were determined at the beginning of the process, a waterfall approach would be the best to work with, mostly employing caution that the requirements and test cases are met. However, it is important to appreciate the complexity and interrelationships of the code being tested. For example, this would reduce the number of errors in the code and catch more in the testing phase because you would be able to notice errors as the code is being written or reviewed. The goal is for errors to be corrected promptly and the failure stage to be avoided completely.

In order to limit bias in the review of the code, I didn’t stress perfectionism because normally when I am coding, I would polish the code multiple times until it looked how I wanted it to look, prioritizing form over function. Another way that I limited my bias was to not settle on code that did not work 100% to my liking and being willing to rewrite code for it to be easier instead of settling on code that didn’t produce the results desired in an effort to finish the program faster. Bias is a concern if the software developer is responsible for testing their own code because they believe the code is correct if it passes testing and is successfully delivered. For example, the developer can deliver code that meets consumer requirements, but they can fail to understand what the consumer needs, and unexpected problems can arise.

Paying attention to detail and writing functionally sound code goes a long way as a developer. Being able to spot errors as you are reviewing code can drastically reduce the number of defects a project can have. It is important not to cut corners when it comes to writing or testing code because not properly and thoroughly testing a SUT will result in there not being enough coverage, which could end up as defects due to the SUT not being properly tested. As explained previously as one of the ways of limiting bias was avoiding taking the easy way out, which is technical debt, because instead of drawing it out and properly getting it done, technical debt wants the quick solution. An example would be this very class. While writing the applications, instead of taking the easy way out, I looked for a way that was technically sound so that the program would work as intended.