Michael R Peck Jr

Southern New Hampshire University

CS 320 – Software Test, Automation QA

Project Two – Summary and Reflections Report

**Summary**

Throughout this project there were three pieces, or modules that had to eventually come together to create a cohesive project, with those being the contact service, the task service, and the appointment service, each with a corresponding object class. Each object class was merely a container for data such as name, customer identification, or IDs for the service class to identify each object to retrieve or modify any data. Per the requirements, these object classes’ variables had to maintain some level of encapsulation with certain fields being mutable and some unique. For example, the IDs could only be unique and not null, or otherwise the corresponding service class could not locate the class by its ID, so it would be lost to the void. So, to counter this and to ensure that each id is unique, the service class would assign this ID as the object is created, rather than have the object create a random one each time it’s generated with its constructor. The other part of encapsulating the data, the data that is stored in these object classes are private and are only obtained or mutated using public functions that can then return or change the private value within each class. This prevents other classes from unknowingly changing data across classes or even within other internal functions. There are certain fields that could not be changed and only returned, such as the IDs, and even the date function can only be validated if the date is in the future, and not the past.

If we are basing the quality of the JUnit tests based on the overall coverage, some of the classes are better than others. I certainly got better with getting higher coverage as I got more and more practice writing these exhaustive tests. The first two classes, the Contact and the Contact Service classes, got the lowest scores on the coverage, mainly because there were more fields here than in the other classes, so more tests needed to be performed in order to be comprehensive enough. For the next iterations for the Task, Task Service, Appointment and the Appointment Service class, I did refactor these newer classes to be more efficient which may have led to better and more comprehensive JUnit tests, and I never went back to the Contact Service class. However, when it comes to JUnit tests and its coverage, it’s not always possible to reach 100% coverage, unless the project is very small and then attaining 100% coverage may be attainable, but that is a lot of tests to write, so the goal here was to get at least 80% coverage, and two of the three test classes did get at least 80% coverage, where Contact Service got somewhere around 60% coverage, if I remember correctly.

When writing these JUnit tests, to ensure that each test was technically sound, I had to write the tests to catch both good and bad values for each of the requirements. So, if a value had to have a certain length, tests were written to check that a given length when changing a value or creating an object met the desired length. So, if a name field had a max field length of 10, writing a test for a length of 9, 10, and 11 respectively meant that the first true would assert to true, and the last would assert to false. Now we do a test for a length of 10 because it is a boundary value, which can cause issues, so we need to evaluate that test to ensure no misbehaving occurs. Now with the phone number value, the length needed to always be of length 10, so a series of 9, 10, and 11, would mean that 9 and 11 would assert to false, and 10 would assert to true, so we would need all three tests, whereas the previous series of tests probably could have skipped the boundary test. We also had to test null and empty values as they are evaluated differently. Null values have a different assertion test for not null, and the empty strings are asserted as equal to “isEmpty” or empty literals (“”), and just about every field was evaluated for both null and empty values based on these two assertions.

**Reflection**

Of the more traditional software testing techniques, there are black-box and white-box testing, both of which tests different aspects of the software. White-box testing refers to the testing of the internal system with test cases or static testing because access to the source code is only present in white-box testing. Black-box testing is more designed to operate when no access code is given, and the testing is strictly done from a user point of view, with a user interface or command line interface. Since no user interface was designed as part of this project, all testing was done with white-box testing in mind. JUnit tests are automated tests that can be written by the developer that can imitate data flow and through the use of assertions, can test good and bad data. These tests can also be kept and used later for regression testing, so there is a good reason to use these tests, so long as the developer writes a list that’s exhaustive enough to get good enough coverage, but not so exhaustive that it consumes an ungodly number of resources, such as computing resources, time or money. These JUnit tests behave closely to black-box testing, since there is no user interface available, but we still have access to the source code, so it may be classified as gray-box testing, which is a mix of both.

Because these tests are meant to coverage all possibilities within the code, there may be many, many tests that need to be written and tested, so depending on the complexity of the project, there may be hundreds of tests that need to be completed, or more. Because this project was smaller and less complex, less than a hundred tests were created to create a coverage level around 60% across the project. However, when software is needed for a project that includes some risk, especially to human life, more testing coverage is needed to ease the concerns of stakeholders, including customers, engineers and shareholders alike.

One concern with self-testing your own code or project is code bias, where the developer is oftentimes blind to their own errors, and often things go unnoticed by the developer. Which is why doing pair programming, or having another programmer evaluate or test another’s code is often used to counteract this bias and oftentimes find errors or opportunities to refactor code to improve performance or reduce any potential risks.

Writing large or complex programs or applications can be stressful and time-consuming, but as mentioned above, when it comes to any risk that involves human life, a developer must be thorough with their code and write their tests to achieve a certain level of coverage. Because not every application will ever achieve 100% coverage on every test, the company will need to evaluate each application against any risks that may be involved and set a level of coverage that is acceptable to all. When starting off with writing these tests, if you write these in iterations, write efficiently with the first few, and then you can those as templates that can be used in future tests, so you can save time that can be used to refactor if necessary, or write more tests than required to further reduce risks.