**Summary and Reflections Report**

During the testing phase of the Software Development Life Cycle (SDLC), each piece of development is tested to ensure that it conforms to the functional and non-functional business requirements. The fundamental test processes include planning, analysis, implementation, and evaluation(Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter, 2015). This isn’t limited to only code as it includes reviewing requirements documentation to see if any errors were made. Static testing is tests that are conducted without the execution of code while dynamic testing uses the code to discover errors during its processing. The longer an error is allowed to escape notice, the more the scope of lost time and effort increases. The sooner an error is caught, the lesser the negative impact and the easier to resolve. So, documentation should be reviewed so that it does not propagate, becoming an error in code, but rather be resolved in the same stage as the error was introduced (Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter, 2015). The result of good testing is the discovery of bugs and defects. The testing phase does not include remedying those errors, only finding and highlighting them to be fixed.

Throughout my software testing, I attempted full coverage of the given requirements for the assignment. Initially, this meant validating that the code worked as it should when using it correctly. Therefore, I used proper constructor values that were within the required length and tested the return of each value through the class methods. After this, I tried as many angles as I could think of to break the software for each variable, method, and conditional. I set up JUnit tests to expect thrown errors and also verified that the error messages themselves were the expected messages. For example, if the description was required to be less than 50 characters, I gave it a description longer than 50 characters. I caught the thrown exception in a variable and used assertEquals() to test the error message caught against the expected message. To verify that the code requirements were covered, I performed a coverage test and manually reviewed each line that pertained to the requirements document. Every conditional per the requirements were tested and verified. The overall coverage of Test.java and TestService.java were 90.2% and 92.2 % respectively. However, the conditions that were not verified were additional code beyond the requirements.

One particular variable test was a date, which was time sensitive. Requirements stated that it could not be a date from the past. To ensure that future regression tests would not fail due to poor programming, I had the test capture the current date (as of whenever the test is run), and added more time to it dynamically, so that it would always be in the future. If I had statically assigned the test date to be 1 year from now, the test would pass until a year later, when it would suddenly start to fail and it would be unnecessarily time consuming to find and resolve. In this way, I took caution of how my code might be affected in the future.

Its important to use a variety of methods to test. Static testing is important because it can be performed earlier in the development process than dynamic testing. The earlier bugs, vulnerabilities, and necessary clarifications can be detected the less waste there will be in time and effort. Any of these that are allowed to continue incur increasing costs of time and money. Static testing also tests with a different angle than dynamic testing and will reveal problems that might not be possible to identify through dynamic tests. For example, keeping code to standards and having good code organization will save time, increase communication, and prevent unnecessary frustration (Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter, 2015), but these may have no impact during execution and therefore undetectable during dynamic tests. Likewise, dynamic testing is better at detecting and understanding system bottlenecks, metrics, and user interactions. Therefore, both are needed to cover the full spectrum of tests to produce solid software. Mock tests are useful for more complex tests. They can be used when the method you are testing will call more methods, but you don’t care about the results of those other methods, so you “mock” a return result (McGlauflin, 2022). The return result is a statically defined value rather than a value that was actually returned by the extra method. Integration tests are useful when specifically testing the interactions between larger functional units of a system (Kaur, 2020).

Lastly, being vigilant in writing secure, functional code is of extreme importance because of the real damage that can be caused by being sloppy and making unnecessary mistakes. This is especially true when working on software that directly involves human risk such as software in transportation (self driving cars, airplanes, etc) or other mechanical operations. However, there are often unexpected consequences to poor programming that result in forms of harm that can be surprising such as leaked personal information. It requires constant discipline, testing, and practice to reduce these defects as much as possible to ultimately help others and not harm them. It can be helpful to use JUnit tests, participate in peer programming, and plan and communicate designs clearly to catch defects and bugs early which help prevent the exponential effects of technical debt.

**References**

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter. (2015). *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition) - 1.6.3 Early Testing.* BCS The Chartered Institute for IT. Retrieved from   
 https://app.knovel.com/hotlink/pdf/id:kt00UC2GX1/software-testing-an-istqb/early-testing

McGlauflin, B. (2022, April 20). *Mocking in java: How to automate a Java unit test, including mocking and assertions*. Parasoft. Retrieved June 1, 2022, from <https://www.parasoft.com/blog/mocking-in-java-how-to-automate-a-java-unit-test-including-> mocking-and-assertions/