Joshua Love

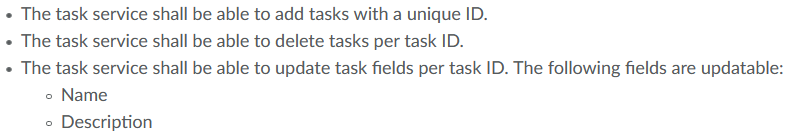
Project Two

SNHU

1. **Summary**

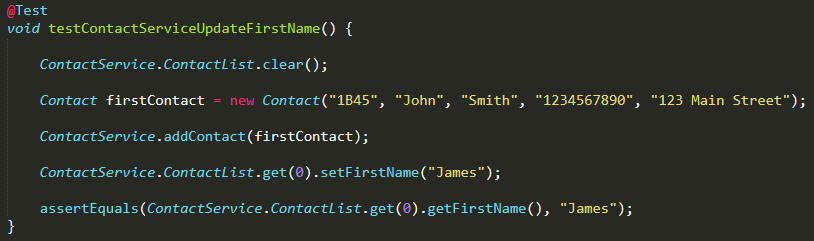
As we completed the components of the mobile application for our most recent customer, JUnit testing played an extremely important role in ensuring that the software met the customers requirements. By taking this method of quality control, this team feels confident that the software will perform to the customer’s expectations. The following report will discuss the quality of the JUnit tests, technical analysis of the tests, overall testing techniques chosen as well as the mindset behind these decisions.

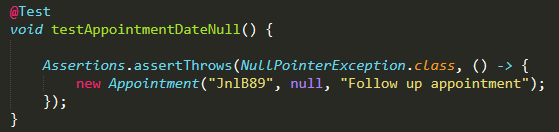
When looking at the quality of the tests written, it’s helpful to compare an example of tests to the requirements provided. One of the requirements was for the software to provide a *Task Service* which added, updated and deleted *Task* objects from memory. The *Task* objects had unique requirements however focusing on the service, it had to be able to perform the following actions:



The JUnit tests were written to address these exact requirements for the service including capturing the more subtle ones such as the unique ID requirement. Specifically, the task service test consisted of six tests: *Task Service* class test, add unique IDs only, successfully add object to array, successfully delete object from array, update name field and update description field. These six tests completely align with the requirements above and helped ensure that all of the functionality was tested appropriately. It’s also worth noting how thorough the testing performed was. In all three components of this software, the testing exceeded the 80% coverage requirement. Each component had two portions: an object class and a service to work with that class. Each of those also had a JUnit test file in order to test the specific functionality. In all three components, all of the testing was above at least 90% with both *Task* & *Task Service* tests both being at 100% and the *Appointment* tests very nearly 100%. This demonstrates that the tests effectively covered the code written as they interacted with more than 90% of the functionality.

While coverage is important, it’s also imperative to be able to demonstrate that said coverage was technically sound. Hitting 90% or above would be fruitless if the tests don’t actually achieve anything. Take for example this code written for the *Contact* component:



This example demonstrates that within five lines of code, we can be certain that the function to update the first name behaves correctly. The test shown here first clears the array in memory, creates a new *Contact* object, adds that to the array and sets the first name to “James” when the original name was “John”. It then uses the *assertEquals* function to test that the object in the first position of the Array now has the first name “James”. This function will return true if it does equal or false if it’s not. This test successfully passed, meaning it returned true, and thus demonstrated a solid understanding of how to use the assertion functions. This example and others also demonstrate a degree of efficiency as previously mentioned. This test accomplishes a definitive statement in five lines of code: the function to update the first name of an object absolutely works as intended. Another example of efficient code is the following: 

Since the appointment class requires us to the use the *Java.Util.Date* library, this example tries to test what happens when a null date is provided. Rather than the typical *IllegalArgumentException* used by a number of the other tests, this one is looking for a *NullPointerException* when a new appointment is created with a null date. Thus, this test can accurately say within one line of code that an *Appointment* object cannot be created with a null date since this test passed. It’s important to note that efficiency should never be sacrificed for quality – tests shown in these examples were carefully thought out to ensure that they were testing a specific requirement outlined by the customer.

1. **Reflection**

This section will be dedicated to a critical analysis of the testing techniques used as well as well as the thought process involved in making testing decisions. The first area of focus will be what exactly was used during this project which is referred to as dynamic testing. In the simplest form, dynamic tests are ones which are run on executed code. The actual developed product is the subject of the tests. This form of testing can be described as “Dynamic analysis techniques, by contrast, depend on analysis of the behavior of programs through their execution and are routinely used in software development” (Bishop, Bloomfield & Cyra, 2013) which highlights the importance of them in the software development lifecycle (SDLC). This method of testing provides the most clear-cut results as to the requirements for the software. However, there’s a deficiency in this method which is that any defects found will be expensive to fix. They’re expensive because there’s already technical debt behind the code at this point and refactoring will increase that number. While this stage is incredibly important, professional testers should be focused on trying to prevent defects from the beginning of the SDLC in order to reduce costs and burdens on the project.

In contrast, this project did not include any form of static testing. Static testing addresses the main pitfall of dynamic testing which is that defects are expensive to fix. Instead of testing the actual code, static tests include tests of the documentation, requirements and the code (without executing) to ensure that they’re compliant with the customer’s needs. Where this fits into the testing environment are categorized as one where “Static techniques find the causes of failures rather than the failure itself…” (Hambling, et. al). This project didn’t employ any of these techniques due to the timeframe of the project, instead the focus was on dynamic testing. Since the documentation from the customer was simple and repetitive, it was possible to skip that form of testing all together. It’s important to note, however, that in a production environment this will lead to significant waste of resources. Enterprise software is incredibly complex and not everything will be accurately passed from the customer down to the developer. The goal of static testing is to intercept those errors and correct them before they become deficiencies. Each static error which is caught saves the project a significant amount of money and technical debt. Static testing can also take on the form of static analysis which involves examining the code without executing. This is the middle of the road between static testing and dynamic testing where the code is already in some sort of form but not one which can be executed yet. While catching an error here does bear some costs, it’s less than when the product is functioning and again helps to improve quality.

Testing techniques aside, it’s also important to reflect on the mindset used while making these decisions. While testing the application, it was crucial to constantly be checking the requirements against what testing code is being written. For example, a very subtle requirement of all three components was the fact that they needed to contain a unique ID. The statement for the requirement was always similar to: “The *X* service shall be able to add *X objects* with a **unique ID**.” This is a crucial requirement that could be easily missed if caution isn’t taken during review. Why it’s so important can be shown in the *Appointment* component where we can’t overlap ID’s since that might lead to the wrong appointment being deleted. This also ties into the fact that it’s important for a tester to understand the code they’re preparing to test as thoroughly as possible. What this implies is that if a tester follows the code through the software development lifecycle and has a firm understanding of its content, they’re more likely to understand what components need to be tested and how they circle back into specific requirements from the customer.

As the developer of this code as well as the tester, it can be challenging to put your code through a fair testing regimen. Taking on both roles means that bias in inevitable however it’s possible to minimize the impact of said bias. One way which bias was reduced was to first create a checklist of the customers requirements prior to any development. This checklist was built to comply with the requirements of the customer as well as the requirements of the assignment. Following that, the code was developed strictly along those guidelines. If a change was required, it had to be confirmed through some form of requirement and then added to the checklist. This checklist then served as the basis for all of the tests conducted. This method ensured that both the development process and the testing process tied back into the most important part – the requirements. If, instead, I had chosen to base my tests off of the code I had already written then I would likely be biased toward parts that “I knew already worked,” which doesn’t employ a stringent testing regimen. Failing to take these ideas into account will likely lead to defects down the road.

Finally, defects are the bane of any piece of enterprise software. Not catching errors prior to their implementation into the code vastly increases the cost to resolve them and the technical debt associated with them increases. Thus, it’s crucial that, as we continue our careers, we remain committed to developing clean, professional code. In today’s world, it’s easy to be working on a project which may be so important that there’s human life at risk on the other end. An example of this is the Boeing Maneuvering Characteristics Augmentation System (MCAS) which had corners cut by being a software that relied on the input of a single sensor rather than building in any form of redundancy (Gates & Baker, 2019). Unfortunately, this type of corner-cutting led to the loss of 346 individuals in two preventable accidents. While this example is an outlier of software usage, it’s important nonetheless that software development professionals and testers take their job seriously. Aside from potential hazards that can result from poor discipline in development or testing, as previously mentioned – defects are expensive to fix. Fixing these defects increases technical debt which in turn increases the cost for the customer. While complete removal of technical debt is impossible on any software of substantial size, being aware of the consequences of too much technical debt is an important step in minimizing it. Additional testing *may* lead into technical debt as it’s deemed unnecessary however any thoroughly investigated testing regimen should pay dividends in reducing expensive defects.

1. **Conclusion**

In conclusion, software developers and the testers behind them should work hand in hand to develop an effective and complete project to their customers. Some key qualities of a testing professional are being efficient, technically sound, cautious and disciplined. A testing professional should also think critically through out the entire lifecycle of the software, align their thought-process with that of the customer requirements and be committed to quality. Testers play an important role in any environment where time, energy and capital are put toward achieving a piece of software. Without this side of software development, complex software would be riddled with issues and defects that simply would not be acceptable. Thus, it’s in the industries best interest to focus on quality since the consequences of cutting corners or lacking in testing could be grander than any organization would like to bear. So long as that is understood, developers and testers will continue to be deliver high-quality, enterprise software which shapes the world.

**References:**

Bishop, P., Bloomfield, R., & Cyra, L. (2013). Combining testing and proof to gain high assurance in software: A case study. *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. doi:10.1109/issre.2013.6698924

Gates, D., & Baker, M. (2019, June 24). The inside story Of MCAS: How BOEING'S 737 Max system gained power and Lost safeguards. Retrieved February 21, 2021, from https://www.seattletimes.com/seattle-news/times-watchdog/the-inside-story-of-mcas-how-boeings-737-max-system-gained-power-and-lost-safeguards/

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