Grand Strand Systems - Software Summary and Reflection

Noah M. Figueroa

Department of Computer Science, Southern New Hampshire University

CS 320: Software Test, Automation

Professor Bermudez

October 20, 2024

Grand Strand Systems - Software Summary and Reflection

My unit testing approach for the three services was highly focused and detail oriented. I worked on each test case class one by one, ensuring all functionality of the subject was being tested. The “TaskServiceTest” class which tests the task service functionality contains tests such as, “void testTaskAddition()”, “void testTaskDeletion()”, “void testTaskUpdates()”. These method headers convey that not only were the base functions tested, but also various aspects of each function such as the error handling. Following this, I then returned to each class one by one to ensure more than 80% coverage was being achieved. Aligning with the software requirements was of utmost importance to ensure all desired functionality and inclusions were present in the end result. My approach was fully aligned with the software requirements as I worked to include each function one by one in a modular fashion. My “ContactService” class contains a myriad of modularized methods including, “public static void addContact(String id, String first, String last, String phone, String addy)”, “public static void deleteContact(String id)”, “public static void updateFirstName(String id, String first)”, “public static void updateLastName(String id, String last)”, “public static void updateNumber(String id, String phone)”, “public static void updateAddress(String id, String addy)”. This goes to show how I worked towards including all desired functionality into the system one function at a time through the creation of modularized methods. Working alongside the software requirements guided my progress and work within the software ultimately providing me with a definition of done.

My JUnit testing and test cases were high quality due to the extent and thoroughness of their testing. Not only would I test expected input, I would also test error handling ultimately raising the quality of the testing. Furthermore, I ensured that all branches of methods were tested properly. This ensured that all possibilities got tested and higher values of coverage were

achieved. My testing went so far as ensuring values were being assigned properly for the various objects being created. Above 80% coverage of the codebase indicates to me that the testing was extremely comprehensive. Given that all test cases passed, all the covered code was correct as well.

The development of technically sound code was achieved through attention to various details and aspects. The code performed as intended and was verified through the usage of the JUnit testing. Edge cases were tested as well to ensure comprehensive coverage of input. An example of this testing is found within the “TaskTest” class. Within a test case specified for the task’s ID variable, I utilized an “assert throws” block surrounding two new task declarations with varying ID lengths. The first declaration contained an ID of exactly ten characters, “new Task("1234567890", "Wash Car", "Wash car both inside and out");”, and did not throw an exception as the length of the ID was exactly ten characters. The second declaration, “new Task("12345678901", "Wash Car", "Wash car both inside and out");”, did throw an exception as the ID was only one character longer. The data structure of choice for the three services were hashmaps as they provided the most efficiency within the given scenario. The following is the hashmap declaration found within my “ContactService” class, “public static HashMap<String, Contact> contacts;”. All variable names followed appropriate and consistent naming conventions. Comments were included throughout the program which was also structured in a very simplistic manner. As previously mentioned, the classes were written in a highly modular fashion making future modifications a breeze. Security best practices were adhered to as well. An example of this is proper limitations of variable exposure, such as the description variable of my “Appointment” class, “private String description;”.

The software testing techniques that I employed for the project were static testing, dynamic testing, and white box testing. Static testing consisted of thorough review of each line of code I wrote to ensure all the logic and variable names were correct. On a larger scale, this typically includes reviewing documents and other design documents associated with the project. Through my acts of static testing, I encountered a few grammatical and syntactical errors within the classes I created. It also dawned upon me at this time that a hash map would be the most effective data structure for the service classes, as I could tie the stored values to the value's associated ID, effectively serving as a key. I utilized dynamic testing through the usage of JUnit. With JUnit, I employed unit testing to test individual methods within the various classes of the project. To my surprise, these JUnit test cases encountered no errors within my system, however I did realize that I needed to create static instances of the services. Finally, I utilized White Box testing. One major aspect of White Box testing is ensuring there is thorough statement coverage. I reviewed statement coverage while I was performing the aforementioned JUnit testing, ensuring that each branch within each method was tested and entered appropriately. Checking the JUnit coverage revealed to me that I was simply using the "put" method of the hash map data structure as opposed to using the add method I created for each service, effectively lowering the coverage at the time of testing.

The software testing techniques that I did not use for the project included Black-Box testing, performance testing, security testing, regression testing, automated testing, exploratory testing, and usability testing. Black-Box testing consists of evaluating the inputs and associated outputs of the software, without looking too much into the internals of the software. This technique was not employed as this project does not contain a user interface. Performance testing tests the software under loads and extreme stress to ensure proper function. Performance testing

also evaluates the system's ability to scale based on varying workloads. Performance testing was not performed as the services themselves will not be subject to extreme or unusual loads. In other scenarios where the software would receive much more traffic, performance testing can help ensure and maintain positive experiences with the software regardless of the load experienced. This will also improve the stability of the software. Security testing involves testing the security of the system, ensuring that vulnerabilities are identified and their potential repercussions assessed. No security testing was performed, however security best practices were implemented such as restricting instance variable access as much as possible. Regardless, security testing is essential for any software to ensure user privacy and data protection. Regression testing simply involves retesting software to ensure that no new vulnerabilities have been added with changes and updates that have been performed. Automated testing automates testing tasks which are deemed repetitive in order to enhance the efficiency and coverage of the program. Exploratory testing is simply regarded as "intuition testing" in which the tester uses their experience to venture into and explore potential defects the system may have. Finally, usability testing involves determining how user friendly and "usable" the software truly is. Gathering real users for feedback on the user experience is a crucial aspect of usability testing to ensure the creation of a friendly and easy to use deliverable. Seeing as I am the only one using these services at the moment, no true usability testing was performed.

The practical uses and implications of these testing techniques vary vastly. Static testing is performed by examining the source code of projects to identify any potential errors or vulnerabilities. The implications, of course, are that the team performing the static testing knows exactly what to look out for. Dynamic testing is performed by testing various aspects of the system in an isolated manner through actual execution of the code, the implication being that the

code executes without error. Black-Box testing is performed with the implication that the software itself is intended to receive input as it essentially tests input and output. White-Box testing consists of the testing of all statements and decision branches to ensure proper functionality of the system, assuming there are decision branches that must be tested. Exploratory testing is as the name implies. Testers have no constructed test cases to work with, rather they explore the software with the goal of designing and executing tests as they go. Regression testing, assuming changes have been made to the software, ensures that newly implemented code has not negatively affected existing functionalities within the system. Performance testing tests the system through various load conditions to understand and correct the manner in which the system performs under load assuming that it will experience any form of load at all. Security testing aims to identify vulnerabilities within the system through penetration testing and vulnerability scanning. Usability testing evaluates the user experience and friendliness of the software, assuming it will be used by a more generalized, untrained population.

Working on this project required the adoption of a highly analytical mindset with the intention of "breaking" the software. The intention of "breaking" the software greatly assisted with the creation of various unit tests. Analytical thinking further assisted with deeper code coverage rates. Acting as a software tester, I employed deep caution in ensuring all aspects of the code were tested in an appropriate manner. It is relevant to appreciate the complexity and interrelationships of the tested code as it provides more testing opportunities and details to be wary of. An example of this is the appointment object, "public Appointment(String id, Date date, String desc)", stored within the appointment service, and containing a specific date assigned by

the calendar. The usage of various external libraries in correlation with objects, and structures to store them, allows one to truly appreciate the testing opportunities and challenges provided.

Limiting bias within the code required several strategies and a specific mindset to be employed. It was paramount to approach the testing aspect with the mindset that there were a multitude of errors within the program. I employed the strategy of spreading the testing over several days to ensure a fresh set of eyes was examining the project every time. This reduced the potential of overlooked errors. I can imagine how challenging testing your own software would be from a software developer's perspective as you essentially are challenging your own ability to overlook bias and ego.

Discipline in your commitment to quality as a software engineering professional is paramount to ensure highly valuable code which is both efficient and error-free. It is relevant to avoid cutting corners when writing and testing code as there is a very high chance that these "shortcuts" will inevitably lead to further issues down the line. These issues may range from improperly functioning software to highly vulnerable software which risks valuable information and satisfaction with the software. I plan to avoid technical debt as a practitioner within the field by adhering to all software regulations, software development best practices, software testing best practices, as well as developing clean and efficient code.