CS 320 Module Five Journal

Adam Kulp

Southern New Hampshire University

**Q: To what extent was your approach aligned to the software requirements?**

**A:** The entirety of all projects literally relies on being aligned with all software requirements. In a real-world setting; software requirements would be the “rules” for the behavior of the project. If a customer comes in saying that they want a program that will count to one million by a factor of two and you code a program that counts to two million by a factor of five you have totally ignored the customers wants and needs and the program is essentially useless to that person. In other words, requirements, which are determined early (usually in the initial stages) in the SDLC determines why the client needs a specific app and ultimately provides a vision for the final goal. Without these specific requirements, poor decisions can be made which can result in problems during development or even after. It’s also important to note that there are two types of requirements functional and nonfunctional with functional being essentially a list of must needed functionality in the program and nonfunctional being requirements necessary to define development and testing conditions. In our specific projects we were given clear cut requirements of what was needed. Functional requirements were given such as collect first and last name with nonfunctional requirements such as names cannot be longer than 10 or 15 characters. These requirements were later tied in when performing Junit testing, to ensure that all requirements were met and any irregularities or deviation from the requirements would be revealed during testing.

**Q: Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were effective based on the coverage percentage?**

**A:** A unit test essentially instantiates a minute part of the built application and creates a verification for its behavior independently from all other parts. A unit test is said to be made up of three independent and distinct phases. Phase one being the initialization of a small piece of the application to be tested, phase two applies a sort of stimulus to the system under test and phase three observes the resultant of the behavior. If the resultant behavior is consistent with expectations then as expected the test will pass otherwise an error will occur either in the form of an error, both of which indicate undesirable results but occur for different reasons. A failure triggers due to an invalid test result whereas errors indicate unexpected results. With all of this in mind I set out to ensure that each requirement was tested in a manner that was appropriate. For example, in the appointment services app I tested for invalid ID, invalid Date, checked to see if the date was in the past and tested for an invalid description. These would hopefully catch and determine any data that wasn’t fitting of the requirements (ie. The ID string was too long) and it was determined that all tests passed indicating success. Other tests were also performed including testing actual valid data to ensure that data would take properly, an existing ID error was tested to ensure that ID’s were unique and couldn’t be changed and a test was also implemented for deleting a task. All of these tests passed as well. Since each element of the code was tested and accounted for, I could comfortably say that this testing was nearly perfectly effective with a nearly 100 percent coverage percentage; but this is on a relatively small scale compared to what might be encountered in the real world meaning that it would be easy to test each element, for larger projects I’m not convinced that each element can be tested in this manner.

**Q: How did you ensure that your code was technically sound?**

**A:** Unfortunately, I’m still fairly new to coding so the fact that I can purely look at code and determine that something is perfect is nearly impossible at this point but there are a few ideas that I can follow and a checklist that I can refer to in determining code viability. Is the code syntactically accurate? Is it logical and concise? And is it Modular? These are many of the questions I ask myself when writing code. For these instances I determined that they were indeed modular and organized as such, they were logical meaning that all requirements were satisfied for the assignment, style and syntax appeared to be accurate as no errors occurred and what I believe to be proper formatting was used and all testing appeared to be thorough and ultimately passed. A specific example of technical control can be the use of the following lines of code:

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (this.getClass() != obj.getClass())

return false;

Task t = (Task) obj;

return getId().equals(t.getId());

}

This section of code checks for the equality of two tasks and returns true if both are equal and false if null. It also returns false if the getclass method fails to be equal and returns true if the id of both objects equates or are false. This example helps to protect the code from erratic behavior in the event that unforeseen elements are input.

**Q: How did you ensure that your code was efficient?**

**A:** Again, being fairly new to coding, efficiency is relative… is efficiency the idea that it works without issue? That it is achieving the best time complexity for the type of program? That it’s properly documented and looks nice? For me personally, I have another list that I try to follow when writing code that leads to efficiency. These include the creation of functions as opposed to repeated blocks of code, the elimination of unessential operations, avoiding the declaration of unnecessary, unneeded or ultimately unused variables, the use of appropriate algorithms for solving the tasks at hand, minimizing if-else statements, breaking loops when appropriate and avoiding the declaration of variables in the global (this isn’t a huge deal at this stage but when code becomes unmanageably long, this can create potential issues.) I believe that I used coding best practices, all errors were resolved and the program ran as intended but my one regret was not commenting appropriately. Comments can definitely aid in efficiency when thinking about peer review, others building on to your code and even for yourself in the event that something goes awry. I have always been terrible with commenting and strive to be better in this aspect.

**Q: What were the software testing techniques that you employed in this project?**

**A:** With regard to the previous milestones, it could be argued that a few different types of testing techniques were used in conjunction to ensure that the program was functioning correctly. It is first important however to understand the actual principles of testing. Testing should meet all customer (or in this case rubric requirements), exhaustive testing is impossible and therefore testing must be completed based on risk assessment, all tests should be planned prior to implementation, understand the pareto 80/20 rule (80 percent of errors come from 20 percent of program components) and start testing on a small scale and extend. For these particular assignments we used unit testing which focuses on individual unit or corresponding group. This is often done by utilizing sample input and determining output. This was done through the use of Junit testing using test cases. We also utilized a type of stress testing which utilizes unfavorable conditions in an effort to see how a system deals with those conditions. This was also simulated via Junit testing to ensure proper handling of exceptions and other unforeseen errors. We also used integration testing to a degree with black box testing being used for validation and white box testing for verification as we added additional elements to our programs in an effort to produce favorable results.

**Q: What are** the **other software testing techniques that you did not use for this project?**

**A:** Other forms of software testing that weren’t directly utilized in these milestones are regression testing which ensures that a whole component works after adding additional components to a complete program, smoke testing which ensures stability and readiness for further testing, alpha testing (in most cases) which is a type of acceptance testing that is done prior to final release. The reason I say in most cases here is we have self-tested our programs which is a type of alpha test but usually a true alpha test is done by QA people. Beta testing which usually involves the community in testing the software for errors (example a game may come out in the beta phase where select people are allowed to play it and must report bugs back to the company. System testing which ensures that the program works efficiently and properly on different operating systems and performance testing which is used to test speed and program effectiveness and is often called load testing.

**Q: For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.**

**A:** Each type of software technique has equally important yet different roles in the software development cycle. Below is a small list of the aforementioned techniques along with some possible

**Unit Testing:** Allows refactored code and upgraded system libraries later on while making sure that the module is still working. Allows a quickly identified fault whenever a change is made that is improper.

**Integration Testing:** The purpose is to expose any and all faults with regard to interaction between integrated units. Test stubs and drivers are utilized in this type of testing. This type of testing is also used to evaluate compliance of a system with specific functional requirements.

**Regression Testing:** A type of testing that ensures the integrity of a code change. In other words, if a code is changed the software shouldn’t be impacted in its current functionality. Essentially this allows the addition of functionality whilst ensuring old functionality remains intact.

**Smoke Testing:** This type of testing is also known as build verification testing and simply ensures that the most important functions work. This is considered to be a non-exhaustive set of tests and the resultant of these tests are used to determine if the build is stable enough to continue.

**Alpha Testing:** A type of acceptance testing to identify any and all potential issues or bugs before release to an end user. This type of testing tends to use black and white box techniques in an effort to simulate real user interaction.

**Beta Testing:** Also, a type of acceptance testing that utilizes real end users in an effort to determine functionality. This type of testing also aids in determining scalability, performance and reliability in strictly real-world scenarios.

**System Testing:** Validates a complete and fully integrated product. The purpose of this type of testing is to evaluate end to end system specifications.

**Stress Testing:** A risk management technique that verifies stability and reliability of an application. The overall goal of this test is to ensure capability under load without crashing as well as proper error handling.

**Performance Testing:** Tests the speed, response time, resource utilization as well as stability and reliability of an application under a specified work load.

**Q: Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ caution**? **Why was it important** **to appreciate the complexity and interrelationships of the code you were testing?**

**A:** When creating the code for the projects, I found it important to employ caution in an effort to ensure that all requirements were met and in a manner that was both efficient and technically sound. This caution would extend to the testing phase as I found it of utmost importance that my code would not only function but know how to handle any errors or irregularities that may arise. So, it was important to ensure that all angles of testing were covered and this means understanding the complexity as well as how each block of the code forms relationships with one another. It has always been said that a good QA tester requires unique skills such as great communication skills when considering development issues, great critical thinking skills by allowing themselves to fall into an end-user’s shoes and ensure that all requirements are being met, flexibility is required due to the multitude of different testing techniques and they must have impeccable collaboration and social skills with an uncanny ability to learn quickly. All of these elements are undoubtedly true because testing cannot be rushed, thrown to the side and corners cannot be cut to ensure a proper end user experience.

**Q: Assess the ways you tried to limit bias in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code?**

**A:** Until reading about this ideology I didn’t even think of it as a problem but apparently, it’s quite prevalent when developers test their own code. Often there is a tendency to execute the maximum number of positive tests as possible to provide a portfolio of working as intended cases. This is usually done by favoring regression testing and specifically choosing test cases and data that will almost certainly pass automation testing to limit bug detection. I could definitely see this being a problem and I feel that solving this would require first the acknowledgement that it does indeed exist and it would probably be of interest to create multiple hypotheses and gathering information from alternate sources in an effort to combat any pre-existing thought or notion. As far as my code is concerned, I felt that I created a test for any and all possible deficiencies that could occur when matched with the requirements, but someone else may look at the same code and say that I missed all kinds of postulates. Now that I’m actually aware of this ideology though, I feel that I could be much more diligent in ensuring that this won’t happen.

**Q: Evaluate the importance of being disciplined in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field?**

**A:** I think the best way to approach this question is to take a look at the comment’s others made in the last discussion post. Cutting corners and “not being all in” can result in many disastrous circumstances (as seen in the posts) such as financial loss, property loss, company devaluation and even death. In a different sense it’s also important to test code because it can actually teach you to write better “good” code because fixing your own bugs allows you to find different ways in solving problems. It can also save time on debugging, cause you to be engaged before and as you write code, pave the way for efficient code, provides documentation and allows for smooth deployments to name a few. Also, you never know who might use your code or need to build upon it etc. so by not giving it your all and being as neat and complete as possible will only make it a nightmare for the next person. I can only hope that as my skills advance that I treat every project as such and give it my all and ensure that I have done the best I can possibly do in an effort to avoid technical debt. Unfortunately, though this will most likely happen at some point because it’s simply a learning experience but the goal is to limit these hiccups over time.

Reference:

Types of Software Testing. (2020, September 23). Retrieved December 06, 2020, from https://www.geeksforgeeks.org/types-software-testing/