Project Two

Kyle Dale

SNHU: CS-320-Software Test Automation & QA

Professor Jacks

April 15, 2023

Summary.

As a developer or tester, understanding the purpose, goal, and requirements are all important pieces of the development of a product. This familiarity allows one to better plan their work, prioritize and guide requirements, and develop efficiently with care. When approaching the development of each of the three features of the project, I made sure to constantly reference the requirements. This allowed me to better ensure that the client requirements were being properly addressed in the functionality being implemented, and guide the JUnit tests created to ensure implemented functionality as well as adequate testing coverage and relevancy.

The approach to each of the code units was fairly similar, as the functionality that was required was mostly shared, and followed an incremental pattern. This began by a review of the requirements, and developing an idea as to how it could be accomplished. Once I had a general outline of how to tackle the project, I began with ensuring the Appointment, Task, and Contact.java units functioned upon importing. Once this was completed, I continued onto the Service class for each of the units. There was only one minor error in the ContactService class: not knowing which contact class to reference, as my previously completed milestone was present in the same Eclipse workstation, also in a default package. This was remedied by moving my project to a new workstation. Alternatively, I could have renamed the default package of my project to something more relevant.

The JUnit tests I developed are effective at testing the proper functionality for each of the units of code, and the integrated code as a whole. I am confident in this, as the development of the JUnit tests was done by employing best practices, as well as often referencing the requirements to ensure the tests assured proper functionality (Dilmegani, 2023). This included test planning, ensuring proper coverage, ensuring isolated tests, and implementing some

test-driven development. In addition to the Unit and Integration testing performed, I utilized the embedded JUnit coverage scan to ensure that the tests created utilized a sufficient portion of the code, defined as 80%. This was fortunately successful, with 85.8% coverage, which further bolstered my confidence in the relevancy and accuracy of my tests.

Concerning whether my code was technically sound, I followed a logical, incremental approach to development to ensure that the tests created were isolated and relevant to the program requirements and verified that integration had occurred successfully. With an incremental approach and referencing the requirements often, I was able to ensure proper functionality of my code units, the integrated code, as well as my JUnit tests. To start, I ensured that the Appointment, Contact, and Task.java files, and their respective JUnit tests all function properly. As mentioned previously, this merely required solving the error with the Contact and ContactService class's incorrect referencing.

Once I had ensured the base functionality of each unit of code together, I could continue onto the integration of the JUnit tests for each unit into a testing suite. This way, I was able to continue to ensure that the requirements were met and input validation was verified. This started by making sure each of the JUnit tests functioned properly as individual unit testers. Once I verified that the Appointment, AppointmentService, Contact, ContactService, Task, and TaskService unit tests maintained relevancy and successfully passed all tests, I integrated all JUnit tests into the testing suite, AllTests.java. This iterative process allowed me to better track what I had already addressed, while remaining cognizant of the requirements, leading to a more efficient and seamless development.

All of these tests served to ensure that my program adhered to the requirements outlined in the assignment, and provided valuable feedback as a developer to the accuracy at which I had

met those requirements. Altogether, this provided me with greater confidence in my work, and helped to assure that it was technically sound.

To ensure that my JUnit tests did not have a significant impact on the efficiency of my code, I attempted to write smaller, precise test cases to target one specific piece of functionality per test. This allowed me to make sure that each test remained small and straightforward, which helped in organization and ensuring each requirement was met. For example, each of the tests operates in isolation, addressing a single aspect of the program. This can be seen in the final three tests testInvalidFirstName(), testInvalidLastName(), and testInvalidNumber() found on lines 66-88 in the ContactServiceTest.java file, which are each designed to target one specific requirement. This minimizes the complexity of each test and guarantees that no test repeats computation, wastes memory, and also does not rely on results of other tests. Keeping my test cases simple and concise allowed me to much more easily monitor what I had already corrected and tested, and target each requirement I still had to test.

Reflection.

Testing Techniques.

Throughout the course of the project, I relied on Static, Unit, Coverage, and Integration testing to ensure proper functionality. The reliance on these forms of testing was sufficient for the purposes of this assignment, as I guaranteed the proper functionality of the various classes required, as well as ran coverage tests to ensure a large portion, or the entirety of my code, was being tested adequately and functioning properly.

The purpose of Unit testing is to ensure that each piece of code functions as intended. Generally, this involves the use of automated test cases that compare behavior against expected

results. An example of this can be seen in my submitted AppointmentTest.java file, lines 27-32, where the testNullId test case has the expected condition of failure. This means that the test will succeed if the input is in fact invalid. An example of a test not meant to fail can be seen in the TaskTest.java file, on lines 15-22, where the test is designed to ensure that a valid task can be added. This test will pass if a task that meets the requirements is successfully added. Unit testing has compounding benefits for the development efficiency of the project. By ensuring each individual unit of code functions as expected, one can prevent costly errors from persisting into the Integration phase, as well as ensure confidence and stability moving forward. This project, specifically, was made significantly easier due to the diligence paid during the milestones. Proper Unit testing allowed me to experience a seamless Integration phase for this project.

Alongside Unit testing, Static and Coverage testing were employed to ensure that the test cases had been created and functioned properly, as well as addressed the code as intended. This is important to ensure the validity of Unit testing. Additionally, Static testing was utilized after the integration process had begun to help ensure no additional errors had persisted.

I was satisfied that, unlike the milestones and reflections, we were able to practice Integration testing in this assignment. It was important for my experience as a developer, and readily highlighted the importance of Unit testing. Having seen a real example of how proper early development can set the benchmark for a successful product, going forward, I will make sure to pay proper diligence in the Unit testing phase.

Although this project went further in the development process and was larger in scope than the milestones, not every phase or form of testing was utilized. For example, security testing was not implemented as this project is a hypothetical assignment and not actually meant for a

client. That being said, inherent security is part of best practices in coding, which were utilized to develop this project in an effort to reinforce personal proper practice.

Another form not utilized is Performance testing. As the integrated code is still small in scope and the Unit tests needed were fairly basic, the effect on the program's performance was negligible. Additionally, as there is no active system yet with which to strain, performance testing was not necessary for the scope of this assignment.

Lastly, it was not possible to perform System or Acceptance testing on the project, as their scope was not large enough. This project does not need to be integrated with any external technology, nor is it relevant to government regulations. As such, System testing was not needed. Acceptance testing by definition is beyond the scope of this project, as there is no client or product to present to the client. A form of Acceptance testing can be seen in the grading of the assignment, as the professor will analyze the assignment for accuracy based on the requirements. Additionally, there is no active interface for a user to interact with the system on, so Security testing was not essential. I did however attempt to develop the units of code and JUnit tests with best security practices in mind, such as utilizing the concept of encapsulation, seen in my Appointment.java file.

Although each of these additional security phases and techniques are important to understand and are relevant to development, they were not immediately necessary at the current stage of our project. Unit, Coverage, Integration, and resultantly Functional testing were the only necessary phases for me to ensure that the specified requirements had been met.

Although each of the techniques and phases of testing were not utilized in these milestones, they remain critical to the success of a product during development. Each phase is necessary to ensure the relevance and success of a project, beginning with Unit testing. As

mentioned earlier, Unit testing is done to ensure each piece or ‘unit’ of code functions as expected prior to integration with the rest of the program. Unit testing is a dynamic form of testing, generally utilized toward the beginning of development. Ironing out defects at this stage is extremely important, as the cost to correct them will increase exponentially beyond Unit testing. This is one of the reasons why Coverage testing is often associated with Unit testing, whereby the tester will run an automatic coverage test alongside their Unit tests, to determine the total percentage of code utilized in the tests. The importance of proper Unit testing was cemented during this project, and Integration was made vastly easier thanks to my due diligence in Unit testing.

Once each unit had been successfully tested for functionality and coverage assured, Integration testing could begin. Integration testing is the process of ensuring that each unit continues to function, as well as integrates successfully into a larger code base without introducing new defects. Integration is critical, as even with properly functioning units of code, unexpected or unintended behaviors can arise upon integration. This was personally seen in the minor changes needed in my project and workstation upon integration. During this phase, it is determined if integration of the units was successful and if the program still behaves as intended. It is critical to ensure proper functionality and behavior during the integration phase, as should defects persist to the Design and Implementation phases, they could prove extremely costly to correct.

It is worth mentioning that Security testing can occur throughout the course of development. A secure software application begins with best design and coding practices. Active steps such as vulnerability assessments during Unit testing as well as Static vulnerability testing prior, can continue through Integration to the System and Acceptance testing phases. As such, it

is worthwhile to constantly be aware of the security demands for a specific project and ensure one is utilizing best practices for a secure application.

After Integration comes the System testing phase, whereby the entire system is tested for proper behavior, as well as correct integration with any external or third party systems utilized. This stage is present to determine that integration was successful, that the system can function independently as well as tied into other systems, and adheres to functional and nonfunctional requirements. As such, this phase generally includes Performance testing for the product as part of the System testing phase. This phase was beyond the scope of our project, and therefore not necessary to perform. However, it is a critical and valuable stage to strengthen the chances that a product successfully passes Acceptance testing.

Finally, Acceptance testing occurs to ensure that the application meets the requirements and specifications of the stakeholders and is ready for deployment. This phase generally occurs after development and testing has been completed, and is generally undertaken by the stakeholder or end-user. There are two types of Acceptance testing to consider during the phase, including User Acceptance Testing (UAT) and Business Acceptance Testing (BAT). UAT is generally where the end-user performs various tests to validate proper function and requirement expectations. BAT is the process where the software is tested to ensure compliance with any relevant business or government regulations that dictate functionality. This phase of testing, aside from being graded, is also outside the purview of the assignment but remains important to be familiar with.

Although the scale of the project was not grand enough to warrant each of the testing methods, it was beneficial for me to keep the principles of each stage in mind, as this helped to identify any defects that could alter function and expectation when reaching these stages.

Ultimately, it is important to be aware of the entire process for the efficiency and efficacy of development.

Mindset.

While working on this project, I adopted a confident, yet wary attitude. This was due to the fact that the large majority of my unit code had done exemplary and required very little correction before integration into a whole. Although I was confident that the project would not require much debugging, I still regularly employed caution. This took the form of duplicating the project immediately upon integration, to provide a fallback should I catastrophically ruin something. Additionally, I applied the same iterative and methodical pattern to my development, as I had done throughout the milestones. Beginning with ensuring the base classes and then service classes were functioning properly. Then, I moved onto ensuring each JUnit test functioned properly, addressed requirements adequately, and passed all tests. Once each JUnit component was completed, I constructed the test suite to run the entire JUnit test at once, and ensured functionality redundantly. Maintaining this cautious and iterative approach ultimately led to a smooth and nearly error-free development. It also served to aid me in correcting the one major error I had, where the program could not reference the correct Contact class, as my milestones and project were in the same workspace all contained in the default package of their project. This caused a referencing error that I have now learned to avoid, and was easily corrected.

When writing and reviewing any code, it is important to be aware of the bias we have as individuals, and how this can affect the program and its relevance to the requirements. When developing my project, I attempted to curtail my bias as much as possible by following best

practices for secure and efficient development, as well as constantly referencing the requirements. Another strategy I often find helpful is a collaborative review, which in my remote state often involves sitting my partner down, and explaining each piece of code and how it fits the requirements. Although they personally have little knowledge of coding or design, the process of sharing aloud allows me to better understand and analyze my work when compared to the requirements.

It is important to be committed and disciplined to any work one chooses to undertake; otherwise, the choice was made in vain. This is especially true with software development and engineering, as the defects and errors as a result of negligence can often have extremely costly and dire consequences. Creating adequate, substantial tests for a project is an essential aspect of development. When code is not thoroughly tested, a host of consequences can result. Some of which were shared in our discussion post this week resulted in death. Although not all defects have such dire consequences, any defect can erode the goodwill, patience, and understanding of a user or client. Maintaining a serious effort and commitment to best practices is the first step to ensuring a successful product, and ultimate career in software development.

The primary active method I utilize to avoid burnout would be iterative development. I learned the hard way, early on, that developing code while procrastinating is a potent combination often resulting in failure. By developing an iterative style to development, I am able to apply my focus and attention across more numerous, shorter development chunks of time. This allows me to not only avoid frustration, but see my work with ‘fresh’ eyes repeatedly, rather than only once. Ultimately, with proper planning and care, the stress of development can be sidelined in lieu of the joy associated with creating and learning. Keeping the passion to continue

to grow and improve as a developer is the most essential step for me to ensure success in my career.

References

Dilmegani, C. (2023, January 3). *Top 10 best practices for software testing in 2023*. AIMultiple. https://research.aimultiple.com/software-testing-best-practices/