Reese Thurman

CS 320

Project Two

1. Summary
   1. Describe your unit testing approach for each of the three features.
      1. **To what extent was your approach aligned to the software requirements? Support your claims with specific evidence.** The testing approach for project one was closely aligned with software requirements. This was accomplished by designing a specific test for each software requirement. For example, the task service component was required to add and delete tasks by unique ID. Accordingly, JUnit tests were created for each specific software requirement. This testing approach ensured that each software requirement was tested by a specific test case.
      2. **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?** The coverage percentage represents the percentage of code that is executed by the JUnit tests. This percentage is a strong indicator of the quality of the testing approach. The overall coverage percentage for both the contact service and task service projects exceeded 80%, which indicates that a significant portion of the code was successfully tested. A coverage percentage above 80% provides confidence that the JUnit tests in the current project are high quality.
   2. Describe your experience writing the JUnit tests.
      1. **How did you ensure that your code was technically sound? Cite specific lines of code from your tests to illustrate.** Ensuring that code is technically sound requires several approaches. For project one, I ensured my code was technically sound using manual code reviews and JUnit tests. If any of my code failed a JUnit test, I would manually review the code block to determine the source of the error, which would be followed by additional testing. The following block of code is an example of a JUnit test that tested whether the code block responsible for deleting task services was technically sound:

A screen shot of a computer code

Description automatically generated

Specifically, the JUnit test would create a new task service and then attempt to delete the service. If the software failed to delete the task service, a manual review was conducted to determine the source of the error. This approach ensured that the codebase was technically sound.

Additionally, I ensured the code was technically sound by adhering to industry standard practices. These practices included minimizing redundant code, ensuring code reuse, implementing comments, and confirming that all variables were initialized and used in the final project.

* + 1. **How did you ensure that your code was efficient? Cite specific lines of code from your tests to illustrate.** JUnit tests can evaluate the efficiency of code in addition to functional testing. Basic performance testing can be evaluated by reviewing the time it takes for JUnit to execute a block of code or run a particular test. The following test summary displays the amount of time taken to run each JUnit test:

A screenshot of a computer program

Description automatically generated

JUnit tests should be designed in an efficient manner. This can be accomplished using several techniques, including maintaining isolation and independence for each test, minimizing dependencies, using setup methods, and using tests that maximize coverage. The following block of code is an example of both setup use and test independence:

A screen shot of a computer

Description automatically generated

1. Reflection
   1. Testing Techniques
      1. **What were the software testing techniques that you employed in this project?** Describe their characteristics using specific details. Several software testing techniques were employed for each of the milestones. These testing techniques included both static and dynamic tests. Static testing is the evaluation of software and related documentation without execution of any underlying code. This process is used to prevent, identify, and correct defects and inconsistencies, thereby increasing product quality and reducing development time and costs. Early static testing in the software development lifecycle also plays a critical role in preventing errors from being built into the software.

The primary static testing technique used for each milestone was manual code review along with document review. The objective of the manual code reviews was to identify errors in the codebase and ensure the code adhered to industry standards. In contrast, the goal of document reviews was to ensure the code was consistent with software requirements.

Like static testing, dynamic testing is a process used to detect and correct software defects. However, dynamic testing requires the execution of code or completed software to identify errors. Although static testing can be completed before any code is written, dynamic testing requires the completion of at least some code for proper execution. The dynamic testing employed for each milestone included both white box and black box tests. White box testing evaluates the internal structure of code, whereas black box testing focuses on the external behavior of software.

The white box testing included for each milestone included unit testing, decisions testing, and code coverage analysis. Unit testing evaluates a portion of code before integration with other units of code, and decision testing validates logical conditions that control branching within the code. Code coverage analyses the percentage of code that is executed during testing. Together, these testing techniques validate that the internal code structure is logical and error free. Unit testing, decision testing, and code coverage analysis were completed using JUnit, a system in which test cases were designed to test every component that was developed for each milestone. Upon successful completion of unit testing and decision testing, JUnit was used to ensure that the code coverage rate was at or above 80%, which ensured that most of the code was tested for each milestone.

The black box testing technique primarily used for the milestones was functional testing. Functional testing is used to ensure that an application meets software requirements. This testing technique was accomplished using JUnit tests. Each JUnit test case was specifically designed to evaluate whether each software requirement was met for every individual milestone.

* + 1. **What are the other software testing techniques that you did not use for this project?** **Describe their characteristics using specific details.** Some software testing techniques were not used for the milestones. Some of the techniques that were not used included integration testing and acceptance testing. Integration testing ensures that different code components work together correctly as a part of a larger system, and acceptance testing ensures the software functions according to the end-user’s expectations. The software is not ready for integration testing since the code developed for each milestone was not fully integrated, and acceptance testing is not typically done until after the software is completely developed.
    2. **For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.** The testing techniques used for each milestone have practical uses and implications for different software development projects and situations. Unit testing is beneficial for early error detection before units are integrated with larger segments of code, which increases code maintainability. This form of testing is also particularly helpful for testing complex functions and complex software projects, thereby enhancing code reliability. On the other hand, decision testing is critical for testing complex decision logic. This technique is helpful for reducing the risk of logic errors and increasing coverage confidence.

Code coverage analysis evaluates the effectiveness of test cases, which is beneficial for industries or software projects that require specific levels of code coverage. Code coverage also helps to identify and resolve testing gaps, which leads to higher quality assurance. Finally, functional testing ensures that software works as expected and meets business requirements, thereby increasing stakeholder confidence and user satisfaction.

* 1. Mindset
     1. **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? Provide specific examples to illustrate your claims.** Caution was employed extensively for each milestone to mitigate risk and increase confidence in the code quality each week. For example, software testing with JUnit oftentimes requires making changes to underlying code, which can potentially introduce new defects. Therefore, it was important to employ caution to minimize the risk of introducing new bugs after JUnit testing was completed for each milestone.

Understanding the complexity and interrelationships of code is critical for identifying software defects. For example, understanding the relationship between the task object and task service in project one was necessary to resolve underlying problems in the interactions between the two classes. Additionally, understanding the complexity of code ensures adequate test coverage when developing JUnit tests.

* + 1. **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? Provide specific examples to illustrate your claims.** Limiting bias is critical to ensuring objective software testing and analysis. Bias can lead to code blind spots, subjective self-assessments, and confirmation bias. Several techniques were employed to limit bias during the code review of each milestone. These techniques included establishing objective code review guidelines that focused on specific software requirements and best coding practices. These guidelines provided frameworks for neutral evaluations that reduced the likelihood of bias. Automated tools such as JUnit were also employed to minimize bias while maintaining consistent coding standards. Finally, external feedback from the professor was implemented to update and enhance code using a neutral and fresh perspective, thus minimizing potential bias.
    2. **Finally, 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? Provide specific examples to illustrate your claims.** Maintaining discipline in one’s commitment to software quality is essential for risk mitigation, ensuring long-term code maintainability and cost savings, as well as building an excellent professional reputation. Cutting corners leads to increasing technical debt, software defects, and security risks. These vulnerabilities can adversely affect software quality and can lead to costly software updates, data breaches, and eroded trust from stakeholders.

There are several techniques that will be used to minimize technical debt, including regular code reviews using automated testing, such as JUnit, extensive use of documentation, and peer code review to limit bias in testing. Together, these approaches offer comprehensive testing and active management of technical debt, leading to the development of high-quality code and success in the software development field.

References

Hambling, Brian, et al. Software Testing : An ISTQB-BCS Certified Tester Foundation guide

- 4th edition, edited by Brian Hambling, BCS Learning & Development Limited, 2019. ProQuest Ebook Central, https://ebookcentral-proquest-com.ezproxy.snhu.edu/lib/snhu-ebooks/detail.action?docID=5837074.