Class: CS-320-R4843 Software Test Automation & QA

Assignment: 7-2 Project Two

Student: Margarita Kiseleva

Date: 04/16/2024

***Project Two – Reflection Paper***

1. ***Testing approach***

The final project for this class incorporated the code and unit tests that I have been developing for the past six weeks, and it became a logical conclusion of the mobile application for the Grand Strand Systems’ customer. I was provided with a list of requirements for six Java classes: Contact, Contact Service, Task, Task Service, Appointment, and Appointment Service. My task was to deliver those classes and develop a number of JUnit tests verifying that the customer’s requirements were met.

For this project, I worked on developing unit tests. **Unit testing** is a type of software testing that involves testing individual components of an application. A component is the smallest tested part of an application, for example, a function, method, procedure, module, or class. I made sure each of my tests investigated exactly one happy or one failing path of the code. Together, my unit tests formed a system that provided an extensive investigation of the application as a whole.

Unit testing is a key step in the software testing process for several reasons. Firstly, it allows for early detection of bugs. Unit tests detect errors at early stages of development and prevent complication of dependencies between program parts. One error in the code can affect many other components of the program. If we do not perform unit testing and catch errors at early stages, fixing related errors will take a lot of time and effort. Additionally, it helps us write better code. When unit testing approach is adopted, we are expected to follow best practices for writing code and ensure that the code is easy to maintain. Unit testing requires that each component of the code performs a specific function that can be tested independently of other components. Moreover, unit testing favors debugging. If a unit test fails, we can easily understand where the problem occurred in the program. This makes debugging faster and more efficient. Finally, unit tests act as an example showing how individual code blocks work and what behavior is expected from them. They provide perfect documentation for the entire software logic [5].

1. ***JUnit Tests***

The concept of **code coverage** is used to better validate the software under test. This approach measures how much code is executed when tests are run. Using code coverage results, we can determine how well the tests cover the entire program code and identify vulnerabilities. Code coverage is of great importance in software development because it helps to detect those code sections that have not been tested and may contain errors. Code coverage helps developers focus on writing quality tests because it helps determine how much of the program is being tested. If code coverage is low, it may mean that developers have missed some of the program's use cases and have not tested many possible bugs [4].

It is important to note that complete code coverage cannot be achieved because complex programs usually contain various branching and state switching. However, the developer's goal should be to have the most complete code coverage possible to reduce the likelihood of bugs in the program and improve its reliability and quality. I was able to check my code coverage in Eclipse through JUnit plugin. My overall coverage for the application was 88.5%, which was acceptable for the purposes of the project.

1. ***Code Efficiency***

It should never be forgotten that code is not only written for computers. It is also written by and for people. Missing this fact can lead to further problems. **Code quality** is interpreted differently depending on the project and the team. But it can be understood as a set of characteristics that define how well the code is written and how well it meets certain standards and expectations. Code quality includes such concepts as operability, readability, maintainability, extensibility, and testability. Quality code allows us to make changes, fix bugs, and add new features quickly and painlessly.

Following the concept of **operability**, I made sure that the working code is correct, reliable, and efficient. For the purpose of making the code readable, I adhered to the common styles and conventions, and I ensured that it has a good structure, logic, and comments. **Readability** of code affects its understandability, maintainability, and extensibility. I follow the same naming, indenting, and spacing conventions throughout all Java classes. I decided to use the camel case, and to adhere to using meaningful names for all my methods and tests. I specifically ensured that test method names were descriptive in what they were testing to provide clarity on passing and failing conditions in addition to the specific attribute or method. Moreover, I put an emphasis on **maintainability** of the code. Code maintainability means the ability of code to be easily changed and adaptable to new requirements or conditions. Maintainable code is modular, flexible, and compatible with other system components, which allows not only changes but also additions, extractions, and extensions. Thus, I made sure that my code allowed new features, classes, and methods to be added without modifying the existing code. In addition, I tried to use **extensive comments** that are concise, relevant, and informative. Most of my comments are single-line and can potentially be used to create documentation with tools such as Doxygen, Sphinx, or JSDoc. Finally, I organized code into modules, packages, and classes. Such code structuring helps to make it more modular, reusable, and easily maintainable. An important part of code structuring is decomposition wherever it is applicable. After all, carefully breaking down entities into separate components helps speed up development.

1. ***Testing Techniques.***

In reference to testing according to scenario types, I had the opportunity to test both **positive and negative** scenarios. Positive scenario testing confirms that a software operates as intended in typical circumstances. Negative scenario testing examines the behavior of the software in erroneous or uncommon circumstances. These kinds of situations demonstrate how the software handles failures appropriately and prevents users from doing actions that are not intended during regular application use. For instance, I checked the program's capacity to establish a new appointment when all the supplied variables were appropriate for my Appointment Service milestone. I tried a few unfavorable scenarios in addition to that positive one. I investigated the behavior of the software when I entered variables that were null or overly lengthy.

I was able to perform both **static and dynamic** testing based on the code execution requirements. Software testing that is done without executing the program's code is known as static testing. Here, I examined syntax, indentation, and naming standards when examining the program's code. Software testing that is done while the application is operating is known as dynamic testing. This allowed me to examine the program's behavior and determine if it adhered to the specifications.

Regarding the degree of automation in testing, I used both **automatic and manual** methods. Software can be tested manually—that is, without the use of automated technologies. The tester uses the program in the same way that an average user would. I performed a variety of use cases and test scenarios, input data, reviewed the outcomes, and looked for errors or strange behavior while manually testing. Automated testing is the process of conducting tests automatically, without the need for human intervention, using specialized software tools. In this case, I wrote test scenarios with instructions on how to carry out certain tasks and evaluate the outcomes. These scripts were then executed using JUnit, a specialized test automation tool, which mimicked human input and examined the output.

In relation to the testing objects, the **unit testing** concept served as the foundation for the tests I wrote. Unit testing is typically carried out when the source code is being created. At this point, the application's separate components are tested. I created those unit tests with the intention of ensuring that every software element functions and produces accurate results when given varying input data. As a result, I switched to integration testing, where my attention was on the interoperability of the application's components. **Functional testing** includes both unit and integration testing. Functional testing confirms that a system or application satisfies specified functional expectations and requirements. Functional testing was done by me primarily to make sure the software ran successfully and without errors, and that it fulfilled its functions and processes as specified. I was not able to use **acceptability testing**, **system testing**, or **end-to-end testing** in this project. This is because system testing necessitates that experts examine every software component as if it were a separate application. As of right now, the application is incomplete. As a result, end-to-end testing is currently not feasible. Additionally, acceptance testing could not be done because the client or customer must test the product at this time. They verify that the project satisfies their needs and expectations. Additionally, they guarantee that the application operates error-free and produces accurate results.

In addition, the white box concept served as the foundation for my testing. **White box testing** works on the presumption that the tester has access to the program's internal workings and source code. He looks into the program's operation "from the inside out" to ensure that all parts and functions are created appropriately and adhere to specifications. I did not use the **black box testing** methodology since it suggests that the tester is unaware of the internal workings or specifics of the software. He views it as a "black box," observing how the system operates and engages with the outside world. By avoiding the internal workings of the software, you can concentrate on testing the way it interacts with the user and the environment [3].

1. ***Tester Mindset***

A tester's job is basically to find defects, which means eliminating problems with the product. A tester's problems can be more complicated than an incorrectly triggered button and an incorrectly written report. Thus, aside from the hard skills that software testers must possess, there is a number of soft skills that are just as important. The first soft skill I tried to adopt was **critical thinking**. It means the ability to analyze information, evaluate all the pros and cons, draw the right conclusions, and make informed decisions. A person with critical thinking is able to see potential risks, evaluate and weigh different approaches and choose the optimal one. A tester with critical thinking thoroughly investigates defects, analyzes all relevant data, and makes the right decisions. Another soft skill I strived to implement was **creativity**. Ability to look at things creatively, from an unexpected angle, and generate new approaches. For example, non-standard testing scenarios. The ability to think outside the box is important in the agile and rapidly changing field of QA. It is a scarce skill. A creative tester is able to see and document a problem where others don't see anything. Additionally, I did my best to implement a **systems approach** to developing tests. This approach implies the ability to understand the interrelationships in complex systems and to anticipate potential changes due to decisions made. People with this skill are able to see the root causes of problems, predict the distant results of actions, create preventive solutions that affect the entire system, rather than "patching holes". A system-thinking tester understands the results of the team's work, even in a low position, is able to optimize his work, adjusting to the situation in the project [7].

There are times when developers, just like travelers, explore their own code mazes. They follow guides, such as documentation and tests, to find treasures in the form of fixed bugs and optimized algorithms. Sometimes they get lost in their own loops and conditions, but with determination they continue their journey, sincerely hoping that the next commit will lead them to the treasure. The truth is that some **developers may be overly optimistic and protective** of the product of their work. They think their code is flawless and, therefore, may miss some details of the program or overlook errors [1]. For the purpose of this project, I was both developing and testing the code. Thus, I needed to make sure I could overcome the bias that could potentially prevent me from finding and fixing issues. To do so, I kept coming back to the list of requirements and making sure they were met by the program.

1. ***Commitment to Quality***

**Technical debt** is a metaphor that describes the cost of future changes or revisions due to incorrect or poor decisions made during the development of a software product. It is an analogy to financial debt: when you take a loan with favorable terms for a short period of time, but in the end, you have to pay back much more money because of accumulated interest. Consequences of technical debt are numerous including reduced code quality, increased development complexity, and increased maintenance costs.

Technical debt can be avoided and minimized with the following practices and strategies: planning and assessment, investment, adherence to best coding practices, continuous attention to quality, transparency, and communication [6]. It is important to assess the complexities and risks, allocate sufficient time and resources to get the work done. **Careful assessment and planning** will help avoid situations where developers lack time for quality development. Companies should prioritize the technical aspects of product development. This includes **allocating time and resources** to create quality code, update development tools, train the team and improve the product architecture. The investment will help reduce the likelihood of its occurrence. Developers should **follow best practices and standards** in their work. This includes using consistent coding conventions, conducting code testing and refactoring, and implementing robust development processes. Continuous attention to quality includes regular code reviews, static code analysis, automated testing, and performance monitoring. Attention to quality helps detect and fix problems early on. Transparency and communication means maintaining open communication within the development team and with stakeholders, looking at risks and problems, and taking preventive measures [2].

***References***

1. Margheim, S. (2020, December 17). *How to Combat Bias in Software Testing*. Test.io. Retrieved April 14, 2024, from https://test.io/resources/blog/how-to-combat-bias-in-software-testing#:~:text=Automated%20tests%20are%20biased%20by%20their%20author%2C%20a,tightly%20related%20to%20the%20specific%20code%20they%20wrote.
2. *Mastering the Art of Clean and Efficient Code: 7 Best Practices for Developers*. (n.d.). FullScale.io. Retrieved April 14, 2024, from https://fullscale.io/blog/write-efficient-code/
3. Pittet, S. (n.d.). *The different types of software testing*. Atlassian.com. Retrieved April 14, 2024, from https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing
4. Reyes, A. (2019, March 21). *JUnit Code Coverage*. JavaCodeGeeks.com. Retrieved April 14, 2024, from https://examples.javacodegeeks.com/java-development/core-java/junit/junit-code-coverage/
5. *Software Testing Strategies and Approaches.* (2023, June 23). BrowserStack.com. Retrieved April 14, 2024, from https://www.browserstack.com/guide/software-testing-strategies-and-approaches
6. *What is technical debt? How to pay it off (with examples)*. (2024, January 3). Asana.com. Retrieved April 14, 2024, from https://asana.com/resources/technical-debt
7. *What Skills Does a Software Tester Need?* (n.d.). Tealhq.com. Retrieved April 14, 2024, from https://www.tealhq.com/skills/software-tester