Noah Martin

Professor Wilson

CS 320

20 October 2024

**Project Two – Summary and Reflections Report**

My approach for this project was highly aligned to the software requirements. I made sure that I had a JUnit test case written that covered each requirement listed for each class file. I also made sure that I had every possible outcome tested. For example, for the first name field in the contact class, I was sure to create a test case for a legal input value, as well as a null value, and a String of over 10 characters. This ensures that any input that the user enters for the first name will be accounted for. To know that I had tested every part of my code, I referred to the JUnit test’s coverage percentage. This ensures that my tests were effective in considering all lines of code, as well as all possible decision trees. This allowed me to be confident that my code was well tested. To ensure that my code was technically sound and efficient, I made sure to use standard coding protocols and read back over my code after I finished writing it. I made sure that there was no extra code that was not needed, so that my classes were as concise as possible without sacrificing any functionality.

The main software testing technique that I used for this project was unit testing. Unit testing refers to testing each individual unit, or the smallest testable sections of code. I used this technique to ensure that every part of the code worked exactly how I intended it to. For example, for the contact class, the requirement was that the phone number for each contact must be a String of exactly 10 characters. I made sure that all possible options were covered for this variable, whether the variable was null, less than 10 characters, exactly 10 characters, or more than 10 characters. I created a separate test case for each of these situations. I used the same idea for all the other classes as well, testing them exhaustively until I was sure each possibility was covered. Unit testing is a form of functional testing, which is testing the functional requirements of the software to ensure that they are met. I only used functional testing during this project, I did not perform any non-functional testing. This type of testing refers to non-functional requirements such as performance, security, and usability. Non-functional testing is more helpful in the later parts of a project, once there is more of a product. I used static testing as well as dynamic testing while working on this project. Static testing is any kind of testing that does not require execution of the actual code. This can include peer reviews, walkthroughs, and technical reviews as well as static analysis of data flows and data structures. I used static testing when I looked over my code to ensure that all project requirements were met and eliminated unnecessary excess code. I analyzed the control flow of the project, which refers to the structure of how program instructions, like conditions, iterations, and loops, are executed. I also used Eclipse, the compiler I worked on, as a static analysis tool. When compiling code, Eclipse will point out syntax errors that need to be fixed. This tool was extremely helpful to me as it eliminated a lot of frustration by quickly finding out why the code is not working as expected. I also used dynamic testing during this project, which is testing that runs the actual code. One form of dynamic testing is white box testing, which gives the tester access to the source code. This is the type of dynamic testing that I performed on my program. I mainly used dynamic testing to ensure that I hade a good code coverage percentage. The coverage percentage tells you what amount of code is covered by structural testing techniques out of the total code written. Using JUnit in Eclipse, I was able to run my tests as coverage and it was easy to tell exactly how much of the code was tested. Eclipse also highlighted the lines of code or decision situations that were not covered by my JUnit tests, making it easy to go back and create new tests to increase my code coverage. Another form of dynamic testing is black box testing, which refers to testing completed without access to the source code of the product. This was not a viable testing method at this point in the project, however. Black box testing is used later in a project, when there is a user interface already created. Black box testing can be very useful in that case as the tester can act as the end user and see exactly what they would see.

While acting as a software tester during this project, I feel that I employed a lot of caution. I was very careful to ensure that my code was well tested and that all requirements were met. However, I understand that this was possible because the project was of a relatively small size. For a much larger project, it would be very difficult to cover all parts of the code. I imagine that bias would be a concern if a software developer was responsible for testing their own code in a real-world example. I might be more inclined to ignore code coverage percentages or small errors in syntax in my own code rather than someone else’s. In my opinion, it is always good to get a second set of eyes to look at something. The software tester will look at the code from a different perspective, which allows them to catch things that the developer might not. Regarding code quality, it is important to be disciplined in commitment to quality. Cutting corners when it comes to either developing or testing code could result in major errors in the product. A small bug left unchecked early in the development process could result in a huge amount of rework after it is caught late. This can be avoided by testing early and often in the development process.