**Project Two:**

Jared Cotherman

Southern New Hampshire University

CS-320 Software Testing, Automation, and Quality Assurance

**Prof. Cross**

**August 15th, 2024**

**Alignment to Requirements**

The approach that was taken to align the software requirements to the implementation of the code for this project was having a full understanding of what the requirements were. For example, for each of the different classes in the code there were requirements that data fields were to not be null, nor exceed a certain number of characters. Through implementation of conditional IF statements in the constructors for each class, these requirements were achieved by not allowing a user to enter invalid data into the program without generating errors. This is evident in the first method of the Contact, Task, and Appointment classes in which the default constructor verifies each data field for correctness.

**Effective Tests**

The quality of the JUnit tests that were created achieved 98% coverage of all six of the classes combined. These tests were effective in testing the constructor methods for each class by checking that a valid object was created. This was effective as the data which was returned by each “getter” method was to match what was entered when each object was created. There were also tests that attempted to create new objects with invalid data, such as a null name field for one of the classes. If an exception was thrown, the test would correctly pass in that the object was not created due to an invalid entry. Running the JUnit coverage for each test successfully showed which lines of cade were fully covered, partially covered, or not covered by highlighting each line with a different color. This helped to develop each test case to achieve the maximum amount of code coverage for this application.

**Technically Sound Code**

The code for this application was technically sound in the sense that best practices were used such as using Java “getter” and “setter” methods to modify and retrieve any object data. Data fields for each class, such as the “Contact Id” field in the Contact class were required to not be updateable after being created. To prevent this data field from being edited, the data field was created as a private final variable in line 15 of the Contact class. The code also has error exception handling to handle unexpected input from a user without causing a runtime error. This was achieved by using illegal argument exceptions if the input was either null or exceeded the maximum number of characters allowed for a specific field.

**Efficient Code**

Efficiency of the code was kept in mind by preventing redundant code and using proper whitespace techniques was implemented to help with readability and maintainability of the code. Brief descriptions were entered throughout the code to explain the purpose of each method and JUnit test to help with troubleshooting. The use of the “this” keyword was implemented in the constructor and all setter methods to standardize and differentiate between the method parameters and arguments. Examples of this code are evident in lines 60 & 68 of the Task.java class file. Class fields were kept private to prevent access from other classes in the program as seen in lines 15 through 19 of the Contact.java class file.

**Techniques Employed**

The primary testing technique used for each milestone has been white-box testing using the JUnit 5 platform. This can also be viewed as unit testing, because according to (Jakubiak, 2022), “Unit testing is a form of white box testing in which test cases are based on internal structure”. The use of conditional testing was employed to test whether each IF statement returned an exception or was false. Conditional testing is a form of white box testing used to verify the correct logical conditions of IF-ELSE statements (Garg, 2023).

**Other Techniques**

Some of the other test techniques that weren’t used for this project due to the scope of the work include black box testing techniques such as equivalence partitioning and boundary value analysis. According to (Morgan et al., 2019, p. 102), equivalence partitioning groups inputs of equivalent data types (strings, integers, etc.) to help reduce the amount of different test cases that need to be developed for each specific instance or input. An example of equivalence partitioning would be a partition which consists of contact names that are all less than 10 characters in length and contain only alphabetic characters. Boundary value analysis is the other type of black box testing technique that wasn’t used. This type of testing technique is, as its name implies, testing operating boundaries of input variables. These variables can be of any data type, but most often coincide with numerical boundaries such as integers and floating-point data types. In a nutshell, boundary value analysis is an important testing technique to ensure the program behavior acts appropriately when a value entered in the program’s input is very close to the minimum or maximum allowable boundary.

**Uses and Implications of Techniques**

Equivalence partitioning has a practical use in programs where a very large amount of input data needs to be tested but would otherwise be infeasible to individually test every value. An example program where equivalence partitioning would be useful could be one that requires a user to create a new username that must be between 3-20 alphabetical characters in length. One equivalence case would include a group of usernames that fits into this category. Another equivalence case could be a group of usernames that have a non-alphabetical character in the name. Another equivalence group may have usernames that exceed the maximum length requirement. Since so many usernames can be created, equivalence partitioning would be a great testing technique for this scenario.

Boundary value analysis has many practical uses in software testing that require checking that a variable is within a specified range. An example of this may include testing a date entry field in a software program. There are many different cases where a boundary exists for dates such as the month (1-12), or the day of the month which can vary depending on the month. Boundary value analysis is also important when checking an input value that could potentially be out of range for the expected data type, such as an integer. If the program is expecting a valid integer to be entered between the range of –32,767 and 32,767, it is important that the boundaries of this range are tested near the extreme minimum and maximum values allowed as well as the program’s behavior when these boundaries are exceeded.

**Caution**

Testing the code with caution was important because it was easy to assume that certain parts of the code worked correctly and didn’t need to be tested. For example, the constructor method for each class seemed relatively straightforward in my mind, but that didn’t mean I could just assume it would work correctly under every circumstance. Test cases were created to ensure correct operation of the constructors when valid and invalid data was passed to the constructor methods.

**Bias**

Limiting bias when testing code that was also written by the same person can be especially difficult due to our own personal biases. Operating under the mindset that my code is well written and contains no bugs must be avoided to catch any potential bugs that may be hidden in the code. I had to treat my code like it was written by someone else and test every one of the methods using JUnit tests. This is a good reason why many software development teams have dedicated testers instead of developers testing their own code.

**Discipline**

It is important to keep a high level of discipline as a software developer or tester, especially when the risk involved with the project is at a greater level. According to (Morgan et al., 2019, p. 12), “Greater risk implies more and better testing”. An example of this risk and discipline that must be maintained could include designing and testing critical safety system software for commercial airplanes. It would be of the utmost priority to make sure the software that goes into the avionics of the plane is free of any bugs because there is an extreme amount of risk – people's lives and safety.

References

Jakubiak, N. (2022, December 6). *JUnit Tutorial: Setting Up, Writing, and Running Java Unit Tests*. PARASOFT. Retrieved August 1, 2024, from <https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/>

Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). Software testing : An istqb-bcs certified tester foundation guide - 4th edition. BCS Learning & Development Limited.

Garg, K. (2023, November 15). *White Box Testing*. Scaler. Retrieved August 16, 2024, from https://www.scaler.com/topics/software-testing/white-box-testing/