For each of the three classes that were built for this mobile application, unit tests were used to ensure that these classes aligned with the software requirements. The functionality that was requested, such as adding and deleting contacts or tasks, was coded and made accessible to users via service classes. Functions such as changing appointment ID numbers were purposely made private, as the software requirements dictated some variables should not be changed after being set. Unit tests were then written for each of the classes so that each of the methods could be tested individually, using the software requirements as guidance for what passing tests should look like. For example, the requirements document outlined the variables that the contact, appointment, and task classes should consist of. Unit tests verified that each of these variables were able to be set and returned correctly, demonstrating acceptable functionality of the application. Tests also confirmed that objects were able to be updated and removed from memory when appropriate. Overall, the JUnit tests written for these six classes all provide greater than 80% coverage, which is generally considered an acceptable goal for coverage percentage. Full coverage could be achieved given enough time, but the cost of this extra time can often outweigh the benefits (Pittet, n.d.).

When writing code for the application and the corresponding JUnit tests, I made sure the code was technically sound by following best practices regarding naming conventions, whitespace, and commenting in the code. In each of the test classes, private variables were used to store testing variables, and JUnit annotations were used to print a status message before testing each class and then trigger a test for each method. Logical assertions were made within each test that could result in a pass or fail condition, mostly asserting that a variable is equal to an expected value. For deletion methods, it was asserted that the container for the objects to be deleted was empty after deleting the contents. To make sure that the JUnit tests were ran efficiently, I tried to create pass or fail scenarios with as few steps as possible while still maintaining at least 80% test coverage. All tests are completed in a fraction of a second, showing a huge saving of time versus manually testing that each function of the application works as intended.

While the tests written for this application are mostly made up of simple assertions that two values are equal, elements of equivalence class partitioning can be seen in the test classes. Arbitrary values that will satisfy any input validation in place are assigned to each of the testing variables used. It is assumed then that if one valid input passes unit testing, all other valid inputs will also pass the tests. This is one of the key aspects of equivalence class partitioning, where input is partitioned into groups that are considered equivalent for the purposes of testing (Hammad, 2021). The amount of testing necessary can be significantly reduced if, instead of testing every possible input, each partition is tested instead.

Other software testing techniques that are frequently employed are boundary value analysis, decision table testing, state transition testing, and use case testing (Hambling, 2015). With boundary value analysis, the upper and lower boundaries of the partitions created using equivalent class partitions are considered and tested. Generally, the boundary value and a value just above and below the boundary are selected for testing. This particular testing technique can be very useful because errors commonly occur at or near input value boundaries, where boundary value analysis focuses its attention. When using decision table testing, a table is created, laying out possible conditions for each function of the code in the rows of the table. Different combinations of these binary conditions are added to the table in columns, each of which represents one of the company’s business rules. Decision table testing is useful when functions have many conditions, as possible test cases that may otherwise be complex to construct can be easily derived from each column of the table once the table has been created. State transition testing considers state changes within a system and the behaviors that these state changes can trigger. A state table is created, which “records all the possible events and all the possible states; for each combination of event and state it shows the outcome in terms of the new state and any outputs that are generated,” (Hambling, 2015, p. 95). These state tables can be valuable testing tools because once the time has been taken to create a sufficient table, it can be easily referenced to rapidly create more test cases. Use case testing employs use case diagrams to create test cases for each of the actors within the system. These tests can be very effective because they give testers “an opportunity to exercise a complete process flow,” (Hambling, 2015, p. 99).

When building the application for this project, caution was exercised building classes and tests that used multiple other classes to function correctly. The service classes for each of the objects all call methods from other classes, increasing the complexity of these classes and the tests associated with them. When writing the application code and test classes, I was cautious when making changes because one small change could cause errors in other parts of the code as different methods were called. To limit any bias I may have when reviewing my code, I closely followed both the software requirements and the assignment rubric as I worked on each of the service classes. Using these documents as guidelines for how the application should function, I ensured that I was creating fair and unbiased tests. However, some developers who are testing their own code might be biased in their testing methods and create tests that are not thorough enough or that hide known errors in their code.

Working as a software developer, being committed to producing quality code is an invaluable trait that can save time and headache down the line in the software development lifecycle. By striving to write code in the best way possible the first time around and to not cut corners, technical debt in a project can be kept to a minimum. The appropriate solution to a problem may initially take longer than a quick fix, but any further revisions that may be necessary later outweigh the initial timesaving. Thorough testing is one way to help ensure code is functional, both now and later.

Resources

Hambling, B. (2015). *Software Testing: An ISTAB-BCS Certified Tester Foundation Guide*. Chartered Institute for IT.

Hammad, M. (2021, February 16). *Equivalence Partitioning Method*. GeeksforGeeks. <https://www.geeksforgeeks.org/equivalence-partitioning-method/>.

Pittet, S. (n.d.). *Introduction to Code Coverage*. https://www.atlassian.com/continuous-delivery/software-testing/code-coverage.