Paul Kenaga

Professor Luo

CS 320

12 October 2022

7-2 Project Two Submission

The unit testing approach that I implemented for each of the three mobile application features utilized requirements elicitation data provided by the customer and the JUnit framework as the basis for test plans. By using this approach I was able to capture all of the needs of the customer and develop individual, repeatable tests that confirmed my code aligned with the software requirements. Through JUnit tests, I tested small pieces of code to verify that my final application functioned properly and failed as expected when invalid arguments are passed. I defined valid arguments, invalid arguments, and expected outcomes for when these arguments are passed. The Contact Service, Task Service, and Appointment Service each contain objects with individual attributes that are required to be specific data types, within a specific length of characters, and not null. JUnit tests were developed to check that each of these requirements were met. For instance, a contact object, task object, and appointment object are all required to have a unique ID attribute that cannot be longer than 10 characters or null. Specific evidence to support that the implemented approach aligned with software requirements are the JUnit tests within the ContactTest test case that use the testCreateContactIdTooLong and testCreateContactNullId test methods to assert that an Illegal Argument Exception will be thrown for contact objects that are instantiated with a null ID or ID with a length greater than 10 characters. Another JUnit test within the ContactServiceTest test case asserts that a duplicate contactId cannot be used to instantiate a contact. The testAddDuplicateIdFail test method asserts that the isSuccess Boolean returned from the ContactService class addContact method is false when a duplicate contact ID is passed as an argument.

JUnit tests were created to exercise all branches of code within the Contact, Contact Service, Task, Task Service, Appointment, and Appointment Service classes. The overall quality of the tests is high because all conditions are tested. This is evident by full 100 percent coverage of the classes when the JUnit tests I created are ran through a compiler. The coverage percentage demonstrates how effective the JUnit tests are and creates confidence in the code. To ensure that my code was technically sound, I used hash maps to store contact, task, and appointment objects as key-value pairs. This data structure allowed me to take advantage of the inbuilt methods it simplify the test writing experience. To illustrate, a hash map is created to store task objects using their task ID as a key when a single instance of a TaskService Class is instantiated. The deleteTask method defined in the TaskService class takes a task ID as an argument and uses an inbuilt remove method of the Hash Map class to delete the associated value which is a task object. If the key is not present, the remove method returns null and causes the deleteTask method to return false. The test method testGetTaskAndDeleteSuccess asserts that the deleteTask method will return true when a valid task ID is passed as an argument and the testDeleteInvalidTaskFail asserts that the deleteTask method will return false when an invalid task ID is passed as an argument. Unlike other data structures, I do not need to create a loop to iterate through task objects while using a hash map. In addition to reinforcing that my code is technically sound, the use of hash maps also improves the efficiency of my code by reducing the number of lines necessary to meet customer requirements. Another way that I ensured my code was technically sound was by developing JUnit tests to exercise both successful scenarios and failures. This allowed my code to become more robust as it gracefully handled a variety of inputs. Specific examples from the code are the testUpdateFirstNameSuccess and testGetTaskAndUpdateFirstNameNull test methods from the ContactServiceTest test case. The testUpdateFirstNameSuccess method illustrates how the application accepts valid input and the testGetTaskAndUpdateFirstNameNull method illustrates how the application rejects invalid input. During the test writing experience, this strategy gave me more confidence that my code would align with the customer’s requirements.

To ensure that my code was efficient, I sought to keep it as simple and organized as possible. A strategy that aided my effort during the test writing experience was the use of the Arrange/Act/Assert (AAA) pattern. I implemented this pattern in each of my JUnit tests to improve their readability and to distinguish between a test's setup, operations, and results. The test method testGetTaskAndUpdateNameSuccess from the TaskServiceTest test case is a specific example from my code that illustrates the implementation of the AAA pattern. The first line of code in the test method performs the setup required for the test by instantiating a task object. The next two lines of code perform that actions required for the test. These actions are to add the newly instantiated task object to the collection of tasks within a hash map and then update the name of the task. The fourth line of code in the test method asserts that the name of the task has been updated accurately. The organizational aspect of this approach efficiently structured my tests and made them easier to maintain. Another way that I ensured my code was efficient was by using descriptive display names and fail messages to identify tests and locate failures. To illustrate, the display name for the testGetTaskAndUpdateNameSuccess test method is "Test getting and updating a task name successfully" and the corresponding fail message for the method reads "Update Task Name Fail: 0 < String Length < 20". If another programmer were to take ownership over the application, they could easily understand the purpose of the test and how to maintain or enhance it.

Through the JUnit framework I was able to implement unit and regression testing using a variety of black box and white box techniques. The black box techniques that employed in this project were equivalence partitioning and boundary value analysis. The white box techniques that I employed were static code analysis, statement coverage and branch coverage. Equivalence partitioning reduced the number of tests necessary to prove that my application ran properly by dividing valid and invalid test data into partitions. I tested each partition through a single JUnit test instead of creating a test for every possible input. For example, the testGetTaskAndUpdateDescriptionTooLong test method from the TaskServiceTest test case examines how a description variable will fail to be updated when its length exceeds 50 characters. The string that is being passed during this test is an invalid argument and contains 66 characters. The result of test reflects results for any value of the task description variable that is over 50 characters. Boundary value analysis was used to determine whether a certain range of values would be accepted by the application. The testAddMultipleAppointmentSuccess test method from the AppointmentServiceTest test class demonstrates the implementation of boundary value analysis. It exercises four valid string lengths for the appointment description variable. Since the appointment description cannot exceed 50 characters, I tested adding appointments with a description character length of 0, 1, 49, and 50. This testing technique showed that no errors were produced at the edges of my equivalence partition which is a common area for code defects (Garcia, 2017). Through an Eclipse plugin I was able to perform coverage testing. This technique assured that my tests exercised every statement and branch within my code, strengthening their quality. Static code analysis was employed through the built-in static analysis tool that Eclipse offers. This tool warned me of syntax errors that could cause bugs without running my code.

Performance testing is a software testing technique that I did not use for this project. It checks whether software meets speed, scalability, and stability requirements. To use this technique it would be essential to understand the performance success criteria based on customer expectations. Then these values would be compared to metrics from the software being tested to determine how well it performs. This form of testing could be used to show how fast the functions I developed in the mobile application to add, update, and delete objects perform against customer goals. The practical use of performance testing for other software development projects and situations is determine whether an application meets a level of performance metrics to make it competitive on the market with similar products (Hamilton, 2022). Another testing technique that I did not use for this project is acceptance testing. Acceptance testing evaluates software to makes sure that it meets customer specifications and works as it is intended to before being released to the market. It provides confidence in the final product to developers and customers (What Is Acceptance Testing, 2022). Before the release of the mobile application, acceptance testing could be used as a final means of checking that the product aligns with customer requirements and an end to project. The implication of acceptance testing for other software development projects is that a product is ready to be released and used in a live environment.

The practical use of equivalence partitioning and boundary analysis is to make Test Driven Development easier and quicker. Developers that apply this style of programming can improve the design of their tests to be more efficient by using equivalence partitioning and boundary analysis because they will decrease the number of tests they need to write without compromising quality or jeopardizing functionality. During situations with time constraints, these techniques help speed up the development process. The implications of coverage testing are that our code and tests are less redundant. During different software development projects, this testing technique can be used to determine the efficiency of tests by seeing what impact they have on overall test coverage. If a test does not improve coverage it should be removed or modified. Additionally, it can be measured how much any one test improves coverage when it is added to a test plan.

In acting as a software tester, I employed caution throughout the project, but especially while developing my JUnit tests. My goal was to develop tests with specific purpose and to focus on how the software will be used. Though multiple assertions in a single test is not always best practice, I did employ this process in tests like testGetAppointmentAndDeleteSuccess. For this example test method, I made three assertions; the first asserted that a specific test appointment object was added to a hash map, the second asserted that the specific appointment object was deleted, and the third asserted that the deleted object could not be retrieved. The first and third assertions act as safeguards to further prove that the test is effective and that the mobile application has the functionality to delete an existing appointment. Handwriting an outline of my test plan for the project is another example of how I employed caution when testing code. Rather than keeping track of all the tests I developed for the mobile application in my head, I organized them in a notebook and then began to code them. This artifact helped me visualize my code before I wrote it. The importance of employing caution this way is to mitigate errors during later stages of development. It was important to appreciate the complexity and interrelationships of the code I was testing so that I did not overlook any aspect.

The ways I tried to limit bias in my review of the code was to adopt the mindset of the customer and apply feedback from my instructor toward future iterations of the project. By adopting the mindset of the customer I was able to create test cases that reflect real situations the mobile application may endure. As the developer, I would not enter invalid arguments because I know what the requirements are. However, as a customer, I may try to pass invalid arguments because I am not as familiar with the requirements. It was important to limit my bias so that these scenarios could be considered during the development of my code. Limiting bias was also important so that I could enhance my code with the feedback I received from my instructor. When developers are responsible for testing their own code, they may not see the improvements that could be made to make their code better whether logically, structurally, or performance-wise. Seeking peer review, especially by someone with more experience, can resolve biases and strengthen ones technical skills. From the feedback I received from my instructor, I was able to enhance my code by using hash maps to store objects instead of an array list and adopting the AAA pattern to organize my JUnit tests better.

After this project, I understand that being disciplined is foundational to my commitment to quality as a software engineering professional. When it comes to writing or testing code it is important not to cut corners because it can lead to bugs within the source code or cause software requirements to be overlooked. In essence, a lack of discipline opens the possibility of technical debt and extends the time necessary to complete a project. I plan to avoid technical debt as a practitioner in the field by being more methodical with my approach, researching and adopting best practices, and establishing sensible deadlines. Establishing a general set of procedures that I can apply toward any project will help me stay coordinated and continuously produce quality work. Best practices can be incorporated within these procedures to make sure that certain things are being done the best way possible, also improving quality. Establishing sensible deadlines will allocate adequate time to writing and testing code which helps negate the need to cut corners.

References

Garcia, B. (2017). *Mastering Software Testing with JUnit 5 : A Comprehensive, Hands-on Guide*

*on Unit Testing Framework for Java Programming Language*. Packt Publishing.

Hamilton, T. (2022, September 16). *Performance Testing Tutorial – Types (Example)*. Guru99.

<https://www.guru99.com/performance-testing.html>

*What Is Acceptance Testing*. (2022, September 24) Software Testing Help.

https://www.softwaretestinghelp.com/what-is-acceptance-testing/