Summary and Reflections

Jake Sheehan

Southern New Hampshire University

Software Test Automation & QA

December 14, 2020

Software Requirements and Junit Effectiveness

The software requirements outlined how the system should function and provided the basis for my testing strategy. Most importantly, the software requirements detailed the limitations of the system. For example, in the task service project the task ID was required to be unique for each task and could not be longer than 10 characters. Many of the software requirements were similar in that provided functional limitations for the system. By analyzing these limitations, I developed a testing strategy that focused heavily on boundary testing of valid and invalid inputs. For example, I tested the ID attribute by supplying a valid input of 10 characters to ensure it was accepted and tested an invalid input of 11 characters to ensure that an exception was thrown. The same strategy applied to whether an attribute was null or not. The software requirements also detailed functions that should be available to the user: add, delete, and update. These requirements were tested simply by checking that the task service data structure contained the correct data following the appropriate function calls.

My Junit tests for the contact and task services were derived from the software requirements. Each test corresponds to a requirement, and every requirement is tested as least once. Since the requirements define the behavior of the system, by testing every requirement I tested all the functionality that the system offers. To have complete code coverage, every conditional statement must be tested so that all conditional code blocks are run during a test. For example, one of the software requirements was that the task ID should be no longer than 10 characters. To accomplish this, my validation method uses a conditional statement that returns true if the ID is 10 characters or less and returns false if the ID is more than 10 characters. For complete code coverage, I tested a valid ID of 10 characters and an invalid ID of 11 characters so that both conditions in the validation method would be tested. I used this strategy to test all conditional statements in my code for every software requirement.

Technical Soundness and Efficiency

To ensure that my code was technically sound, I focused on two key factors: errors were handled gracefully, and data structures contained the correct data. In both the contact service and the task service, errors could be introduced by user input. For example, an error occurs if a user attempts to update a task name when the task does not exist. This type of error could cause the program to crash, or even open a security vulnerability depending on the data structure used to store tasks. With this in mind, I implemented a method that would attempt to find and return a task but would throw an exception if the task ID did not exist in the data structure. When updating a task name, I called that method wrapped in a try-catch block as shown below.

try {

Task theTask = this.findTask(taskID);

theTask.setName(newName);

} catch (IllegalArgumentException e) {

e.printStackTrace();

}

That way, instead of crashing when the user attempts to update a task that does not exist, the software will handle the exception gracefully and continue code execution without a crash. To handle the error, I chose to print the stack trace to the console for simplicity. A better approach would be to log the error and notify the user in the UI, but that strategy was beyond the scope of this assignment. In the Junit tests, I used the *assertThrows* method to ensure that invalid input threw an exception.

To ensure that data structures contained correct data, I created new objects and used the *assertEquals* method and the *assertNotEquals* method to test them. For example, the TaskTest.java file tests that a task name with 10 characters is accepted accurately using the following code.

void IdWithTenCharTest() {

Task task = new Task("1234567890", "Test", "Test");

assertEquals(task.getID(), "1234567890");

}

By asserting that the task accepts and contains the correct name, a passing test proves that the code functions properly. In a similar way, I tested that all task ID’s are unique using the *assertNotEquals* method. The code was considered technically sound if the test passed, showing that no two ID numbers were the same.

Non-functional requirements were not given, so code efficiency was not explicitly tested. However, Junit does return the time that each test takes to complete, and all tests completed in a reasonable period of time. I believe it was reasonable to expect that no test took longer than one second to complete, which they did not.

Software Testing Techniques Used

For each of the milestones, I relied heavily on equivalence partitioning and boundary analysis. Equivalence partitioning is a technique that groups together similar inputs and treats them as being the same. How the groups are formed depends on the system requirements of the software. For example, it may make sense to put positive numbers and negative numbers into two separate partitions. Then, all negative numbers can be treated the same and all positive numbers can be treated the same. This technique helps to identify natural boundaries in the system so that a boundary analysis can be conducted. During a boundary analysis test, boundary conditions are tested to ensure that valid inputs are accepted, and non-valid inputs are handled properly. By testing the boundaries of equivalence partitions, we can achieve full code coverage without testing every possible input value. In the milestones, I employed these techniques to test user inputs. For example, the Appointment Service should have an ID no longer than 10 characters and it should not be null. Since only positive numbers will be used for the ID, I create two equivalence partitions: numbers between 1 and 10, inclusive, and numbers greater than 10. By creating the two partitions, I can achieve full code coverage simply by testing one number from each partition. To ensure a proper boundary analysis, I chose to test one valid boundary number and one non-valid boundary number.

Software Testing Techniques Not Used

The milestone assignments were relatively simple and did not require every type of testing strategy. For example, it was not necessary to use decision tables. Decision tables are used in situations that have complex conditional paths. They define business rules by listing all possible logical conditions and the actions that will follow. The goal of this testing strategy is to test every possible conditional combination. In doing so, the test guarantees complete code coverage. In the milestone assignments, the logical conditions were simple, so it was unnecessary to write a formal decision table. I did, however, test all possible conditional paths.

Another testing strategy that was not used during my milestones is state transition testing. This testing strategy attempts to test all possible states that the system could be in, and the possible transitions between them. Often, a transition state diagram is used for this purpose. This testing strategy may seem a bit abstract, but an example may help clarify. Consider a hand-held video game controller. At any moment, the controller could be in one of several different states. Maybe no buttons are being held down, or maybe the user if holding the joystick to the right. It is possible that the same button could perform different actions depending on the current state of the controller. In state transition testing, the goal is to cover every possible valid state, the possible transitions, and any actions that may follow from the transition, ensuring that any errors or invalid states are handled gracefully.

Practical Uses and Implications

The software testing techniques that have been discussed are widely used, but the usefulness of each technique depends on the system under test. Equivalence partitioning should be used any time that too many possible inputs exist to test them all individually. Since truly complete code coverage in that situation is impossible, the only way to achieve a reasonable testing standard is to group similar inputs together. Similarly, boundary testing should be used anytime a logical boundary is present. Boundaries are fertile grounds for testing because off-by-one errors are among the most common bugs in software engineering. Decision tables and state transition testing are useful on larger, more complex projects. As a project grows, the space of possible logical conditions and states of the system can become quite large. If the system under test is expected to be complex, decision tables and state transition diagrams should be maintained from the start. The later these tools are introduced, the more difficult it will be to ensure their accuracy.

Mindset

In acting as a software tester, I was sure to employ caution. When working on a system, the most important principle is to avoid doing harm. Small changes in one part of the code can cause large errors in other parts of the system. One important strategy to avoid this was proper annotation. By annotating the code with Javadoc syntax, the Eclipse IDE shows the expected parameters and return types of functions. This documentation not only helped to prevent errors, but it helped me follow consistent naming conventions. When a change to the source code was necessary, I was sure to update the annotations to reflect the changes. Also, I was sure run every test again, a strategy known as regression testing, to confirm that my changes did not negatively impact other parts of the system.

To limit bias in my software testing, I employed a strict division of labor. When acting as a tester, I focused solely on writing test code. When acting as a developer, I focused solely on writing classes in my source code. By doing so, I was able to avoid many pitfalls that are common when developers write their own tests. For example, we all know that code should be written to pass the test, and not vice-versa. However, without a proper division of labor, some developers will do this backwards and write tests so they will pass code that was already written. It is a bit like a teacher writing quiz questions after the student has already filled out their exam. You learn nothing about the quality of the solution. By writing my tests first, and then focusing solely on development, I maintained the integrity of my test suite.

It is important to be disciplined as a software engineer. Writing unorganized code or cutting corners to save time can lead to technical debt during a project. Technical debt, especially debt incurred early in a project, can be very costly. When working on a new feature or a new test suite, it is important that work is completed to the best of your ability before it is considered done. What constitutes “done” will be defined by the system requirements documentation. This form of discipline is especially important in an environment where the developer writes both source code and tests. For example, in a test-driven development system, the developer will first write tests based on the system requirements, then write code so that it passes the tests. A lack of discipline in such a system could lead costly mistakes because there is nobody to catch bugs that occur. If the developer writes inadequate tests to make the development job easier, the tests may not provide enough testing coverage.

Overall, there are a few principles that should be followed as a disciplined software engineer. First, a definition of “done” should be established before work on a feature begins. This definition will help to determine if more work is need, or if the feature is complete. Second, tests should be written before the feature is developed. This prevents the developer from purposely writing lackluster tests to make their development job easier. Third, technical debt should be avoided if possible. Leaving problems to be fixed later is a recipe for disaster. This was the mindset that led I successfully employed in this term’s final project.