Chris Trimmer

CS320-T3331 Software Testing Automation & Quality Assurance

Project 2

2/15/2023

**1. Summary of Unit Testing**

This project is centered on writing unit tests for a mobile application that our company, Grand Strand Systems is developing. Our job was to write the code for the application which includes service classes that will enable customers to add, update, and delete contacts, tasks, and appointment objects within the application. The application is composed of three concrete classes (Contact, Task, and Appointment) and their respective service classes (ContactService, TaskService, and AppointmentService) that perform CRUD operations on containers of the objects.

*a) Unit Testing Approach*

i. Alignment with Software Requirements

The requirements for this project are highly specific. We are given specific details on how each variable for each class is to be implemented. Furthermore, the requirements for this system are functional in nature. This guided my strategy for testing. My testing approach was aligned with the requirements by creating test cases for each requirement. In the concrete classes, we are given restrictions on the input arguments for each variable. For each variable I created accessor and mutator methods. I coded the restriction requirements in each mutator method, and this is what drove my test cases. The following screenshot supports how my approach was aligned to the software requirements.

**Figure 1**

Text

Description automatically generated

This screenshot shows that the first name of a contact cannot be null, cannot be blank, and cannot have a length greater than 10 characters, which is aligned with the software requirements. I implemented each mutator in each class in a similar manner, based on requirements for each data member.

Similarly, in the service classes, we were asked to create basic CRUD operations for a list of respective concrete objects. For these classes, I built the test cases by knowing how the operations are expected to perform. For example, the delete functionality should obviously only be able to delete an object from the list if it exists in the list, and not crash the system if the object is not found in the list. The following screenshot of the delete functionality supports my approach that a contact will only be deleted if it exists in the container:

**Figure 2**

Text

Description automatically generated

ii. Effectiveness Based on Code Coverage

We can see the percentage of statements and decisions that are covered using code coverage. The benefit of using code coverage is that “it provides a quantitative assessment of the extent and quality of testing” (Morgan et al., 2019). Statement coverage measures how many statements are covered in the block under test, whereas decision coverage identifies if the true and false conditions of decision statements are covered. It is possible to have 100% statement coverage, but this does not guarantee that 100% of decision coverage is obtained. However, if we have 100% decision coverage, then we are guaranteed that the outcomes of every decision are covered (Morgan et al., 2019). This is important to know because basing test results solely on statement coverage could lead to false belief that the block of code does not have defects. Whereas we will have much more confidence that our tests are effective based on 100% decision coverage. The following are screenshots of statement coverage (figure 3) and decision coverage (figure 4) from my Contact and Task classes, which show that I achieved 100% on both statement coverage and decision coverage:

**Figure 3**

*Statement Coverage of Contact and Task Related Classes*

Table

Description automatically generated with low confidence

**Figure 4**

*Decision Coverage of Contact and Task Related Classes*

Table

Description automatically generated with low confidence

b. *JUnit testing experience*

i. Technically Sound Code

I ensured my code was technically sound by following testing techniques. As defined by Morgan et al., (2019), techniques can be divided into three categories: black-box, white-box, and experience-based testing. For the system we are designing for Grand Strand Systems, I followed black-box and white-box testing techniques. The requirements for this project are very well defined, therefore we can derive test-cases directly from the requirements. An example of this occurs in each of our classes as we have specifications for each member variable of each class. The following examples show how I coded the setFirstName() accessor function of the Contact class, and the test cases for the method. The requirement is that the first name field shall be a String datatype not longer than 10 characters, and that it shall not be null.

**Figure 5**

*Code Test of setFirstName() of Contact Class*

Text

Description automatically generated

**Figure 6**

*Implementation of setContactFirstName() Method*

Text

Description automatically generated

Based on the requirements I was able to build a test case for each specification that is part of the requirement. The setContactName() function itself mirrors what the requirement is. Since the requirement is highly specified, we can employ black box testing techniques.

White-box testing is used to test the system and component structures. In this project, we are primarily focused on the component level, which we can use statement and decision coverage to accomplish this goal. These are techniques I outlined in the preceding section. The formal way I designed my test cases includes setting up test suites for each class. In each test suite I mocked the class variables with input values. In jUnit we can use annotations to instantiate the variables before each test and tear down the variables after each test. This ensures that each test is getting variables with expected and unexpected values. The following figures are examples of how I structured the setup and tear-down variables for the Task class testing:

**Figure 7**

*Mock variables used Task Testing*

Text

Description automatically generated with medium confidence

**Figure 8**

*Setup used before each test*

Text

Description automatically generated

**Figure 9**

*Mock Tear-down after each test*

Text

Description automatically generated

The benefit of statement coverage is that it tests code that might not have been tested using other techniques, such as black-box or experience-based testing. In those testing techniques, we are not guaranteed that every statement is tested. Now that the test suite is set up, we can use the mock variables in any given test case to test every method. To accomplish 100% statement coverage, I created test cases for each method in each class. In each test case, I used the mock variables as input to the methods under test. The following figure is an example of setting the task description field in the Task class. It uses the variable ‘invalid51’ which is a string of 51 x’s.

**Figure 10**

*Example of Variable Usage*

Text

Description automatically generated

Decision coverage ensures that all paths of our decision statements are tested. Even though we may have achieved 100% statement coverage, this does not guarantee that all decision paths have been tested. Thus, it is imperative to exercise decision coverage. To achieve 100% decision coverage, we must design the test case to ensure all decisions – true and false outcomes – are covered. The following figures show the test case I built for the addTask() function of the TaskService class and that each decision path of the addTask() method was covered:

**Figure 11**

*Example of addTask() Method to accomplish Decision Coverage*

**Text

Description automatically generated**

**Figure 12**

*Example of 100% Decision Coverage in actual addTask() Method*

A screenshot of a computer

Description automatically generated with medium confidence

ii. Code Efficiency

I have ensured my code was efficient from a testing perspective as well as in my implementation methods. In my testing code, I used two techniques for efficiency: equivalence partitioning and boundary value analysis. Using equivalence partitions means that I do not have to test every possible string length or string value. Instead, we can partition the dataset in a way that groups input into sets in such a way that we only need to test one value from each set. The one value is representative of all values in that respective set. An example of this is the setTaskName() test method. In this test case, we have three partitions: when name is null, blank, and greater than 20 characters because these are situations that would cause failure. This results in only needing three test cases to represent all possible values.

**Figure 13**

*Test cases for the setTaskName()*

Text

Description automatically generated

**Figure 14**

*Coding example of setTaskName()*

A screenshot of a computer

Description automatically generated with medium confidence

Boundary value analysis is the process of ensuring any values that are at or near the boundaries of acceptable input behave correctly. In this case, I ensure that the boundaries are covered by testing against a blank string, and against a string that is twenty-one characters long. This eliminates the need to test all the values between 0 and 21, because we know those values are already valid. We only needed to write three tests, instead of more than 20.

The other area where my code was efficient was using for-each loops in my service classes. Firstly, for-each loops have built-in safety because they ensure that we do not run into IndexOutOfBoundsExceptions. When we use standard, index-based for loops, we must ensure that all three clauses of the for loop are set correctly, and that the logic within the for loop works correctly. Furthermore, I consider for-each loops much more readable and understandable than standard, index-based for loops. From the perspective of testing, readability of code is as much an important factor as is the internal structure because ultimately somebody must maintain the code after development. The more readable the code is, the easier it is to maintain it. The following is a figure of the deleteContact() method that shows the for-each loop.

**Figure 15**

*Example of For-Each Loop*

Text

Description automatically generated

2. **Reflection**

a. *Testing Techniques*

i. Software Testing Techniques

The software techniques I employed throughout each milestone are divided into three broad categories: black-box testing, white-box testing, and experience-based testing. Within each of these groups, there are a myriad of techniques used to test software. Black-box testing is primarily used to test functionality and behavior of the system, white-box testing involves testing the internal structure and logic of our code, and experience-based testing is creating test cases based on the experience of the testers. Some of the black-box testing techniques I used include equivalence partitioning and boundary value analysis. Some of the white-box testing techniques included statement coverage and decision coverage. Each of these techniques will be described in the practical uses section of this report (section 3).

ii. Other Techniques

There are many techniques in black-box, white-box, and experience-based testing. Some of the black-box techniques that I did not use include “Fuzz Testing” and model-based testing (MBT). Fuzz testing is a form of random testing “which randomly mutates well-formed inputs and tests the program on the resulting data” (Garcia, 2017). All my test cases that involved mocking user input only required equivalence partitioning and boundary value analysis, therefore there was no need to perform random testing. MBT is the process of deriving test cases “from a model that describes some (if not all) aspects of the ([system under test]) SUT” (Garcia, 2017). I did not use MBT because there was no need to model the system.

White-box techniques that I did not use include fault injection. Fault injection is the “process of injecting faults into software to determine how well (or badly) some SUT behaves” (Garcia, 2017). In this project we are only testing concrete classes and their associated service classes, not on how these classes interact with each other or with the entire system. Therefore, it wasn’t feasible to perform fault injection.

Experience-based techniques that I did not use include error guessing and exploratory testing. Error guessing “takes advantage of a tester’s skill, intuition and experience with similar applications to identify special tests that may not be easy to capture by the more formal techniques” (Morgan et al., 2019). Exploratory testing involves creating test cases based on requirements that are vague, inadequate, or just non-existent, and “there is severe time pressure” to complete the testing (Morgan et al., 2019). For this project, the requirements provided were highly specific and the system at this stage did not require the use of these techniques.

iii. Practical Uses and Implications

In the previous sections I have discussed the testing techniques I used and did not use throughout this project. I will now discuss the practical uses and implications of using them for different software projects.

Equivalence partitioning means that we do not have to test every possible value of an input. Instead, we can partition the dataset in a way that groups input into sets in such a way that we only need to test one value from each set. The one value is representative of all values in that respective set. Instead of creating hundreds of test cases, we may only need to create three test cases.

Boundary value analysis is the process of ensuring any values that are at or near the boundaries of acceptable input behave correctly. Consider iterating through an array. We want to test the upper and lower bounds of the array to ensure we do not get an IndexOutOfBoundsException when supplying what we think is valid input. Likewise, we want a test to prove that an IndexOutOfBoundsException does occur when it is expected to. The benefit of this is ensuring that edge cases are handled correctly.

Statement coverage measures how many statements are covered in the block under test, and inevitably in the system. The benefit of statement coverage is that gives us the confidence that every statement has been tested. Consider that some lines of code may not be exercised very often using other testing techniques and when software is being used by customers. Statement coverage gives us assurance that these lines of code have been tested.

Decision coverage ensures that all paths of our decision statements are tested. Even though we may have achieved 100% statement coverage, this does not guarantee that all decision paths have been tested. Thus, it is imperative to exercise decision coverage. To achieve 100% decision coverage, we must design the test case to ensure all decisions – true and false outcomes – are covered. This gives us confidence that the entire decision tree has been tested.

Fuzz testing is a way of injecting random, unexpected input to a system to see how the system behaves. A real-world use of fuzz testing is to test for security vulnerabilities in a system. The result may be that the system crashes, or that there might actually data leaked out of the system through stack traces or exceptions.

Model-based testing is creating test cases based on a model of the system derived from specifications. One benefit of MBT include finding specification and design defects before code is written (The Challenges and Benefits of Model Based Testing, 2019). Another benefit is that the models are based on the perspective on how the system is supposed to work – the expected behavior as view from the end-user. We can know upfront if there are defects or flaws during design rather than after deployment.

Fault Injection is the process of injecting faults into the system to see how it behaves. It ultimately ties into other coverage-based testing however it is more common at a higher level than unit test as performed in this project. Examples of this might include testing command line input, or input to other API’s.

Error guessing and Exploratory testing are experience-based testing techniques that can be used in many applications. It is mostly based on the experience and skillsets of testers. It is applicable when other formal techniques do not adequately apply to a test case, or when there is a known or suspected weakness or vulnerability in the system (Morgan et al., 2019). Exploratory testing is useful when we don’t have adequate specifications and there is a time limitation.

b. *Mindset*

i. Using Caution

As a software tester it is imperative that we are cautious and deliberate in testing and debugging code. We routinely see and hear of projects that fail or have defect escapements that result in tremendous financial and information loss due to not being cautious. In 1999, a NASA satellite, the Mars Climate Orbiter, failed its mission. The $125-million satellite was lost due to a rounding error between software systems (Cowen, 1999). One system produced an output in English units, while the connecting system was expecting the result to be in metric units. The main impact of the defect resulted in the loss of a probe that cost $125 million to build, as well as time (the journey took 286-days (Lloyd, 1999)) before the defect was found. This is just one example of how important it is to be cautious in our testing.

I employed caution in my testing by ensuring I followed requirements as strictly as possible. It is easy to get caught into traps such as adding additional features, and doing more than what is required to enhance a feature, but this should not come at the cost of jeopardizing our ability to adequately test the product. Likewise, we must stay focused on all possible ways that the customer might use the software, and then try to write tests that cover these different possibilities. Boundary-value analysis and equivalence partitioning are examples of techniques that I used in my project to help mitigate varying possible inputs and outputs.

It is also important to appreciate the complexity and interrelationships of the code we are testing. The example of the Mars Orbiter scenario is an example of how the miscommunication between two interrelated system was a major financial loss and ended wasting nearly a year of time. As an application becomes more complex, it becomes more difficult to test. Even a monolithic application that does not interface with other API’s can be complex due to the amount of lines of code, the myriad of function calls, and how the application is designed and implemented. Also consider how difficult it can be test an application that is built upon microservices and multiple interfaces between API’s. In both scenarios, monolithic and microservices, it is important to consider how testing will be accomplished during planning stages of the project. If a unit or system cannot be tested, then we should consider engineering a different way of implementing it. This leads us to think about the software design process from the perspective of testing first. In the long run, this will save us a lot of time and effort and ensures that the application is more maintainable.

ii. Limiting Bias

As a developer and tester we may have experiences that guide our thought processes in what a system should do, how it should perform, and of course how the software should be implemented and tested. However, this can also lead to our own internal biases misguiding us, or cause us to miss factors that we have not considered. This is normal human behavior because it is impossible to have perfect situational awareness especially as an application becomes more complex.

The way that I limited my own bias in this project was to ensure my tests specifically covered the requirements. In several cases, I had thought of different ways to implement the code, however many of my ideas would have been more difficult to test, and they were not part of the actual requirements even though they would have satisfied the requirements. For example, consider that each concrete has the requirement for a unique identifier for each object created. I could have implemented this by using a UUID. This would have required adding an additional Java library, determining how the UUID would be accessed, and how the UUID would be generated. A UUID is great way to uniquely identify a record, however it far exceeds the requirements necessary for this project. Using a UUID would have also made testing the class more complex. While it is not impossible, it was just impractical to implement UUIDs for this project.

iii. Discipline in Commitment

In many industries there is pressure to release a product as quickly as possible. However, many times this leads teams (or individuals) to sacrifice the quality of the product by using inferior methods to produce and test the product, or simply just cutting-corners to get the product into the marketplace. There other reasons a team may cut-corners, such as lack of technical expertise, lack of funding to purchase testing products, and lack of commitment on behalf of the team to value the importance of quality. Inevitably this leads to a poor product that will eventually fail or not meet the customer’s expectations. This is why we must be disciplined in our commitment to software quality.

The key is to avoid this trap of technical debt. Implementing software based on short-term gain should be avoided because in the long run the product will be inferior, as it will be more difficult to maintain, less extensible, and most likely have escaped defects. Furthermore, the product may not be re-usable for future expansions, nor be extensible to accommodate higher load and demands. To avoid technical debt, we should ensure that we value the quality of the product above how quickly it can be rushed to market. This is easier said than done, however, because most projects are tied to a schedule, and if the schedule is not met than people may be reprimanded or even relieved of duties, or the customer may not desire to use the company for future work. So maybe the plan starts at the top, with the expectation that schedule slippage is acceptable to ensure quality is maintained. Next, the team should ensure that it is armed with adequate subject matter expertise, as well as technical tools to perform quality testing within a given time frame. This may require more money to fund the project, but that is the balance – if we want faster development speed, then it will usually require higher cost, else the inevitable sacrifice of quality, which is what we should avoid. Furthermore, testers should be given adequate amount of time to carry out testing ensuring that processes and standards are followed.

**3. Conclusion**

Our project for this course centered on developing and testing a mobile application for a customer of our company, Grand Strand Systems. This included developing three concrete classes and three respective service classes, and unit tests for each class. The overall purpose of the project was to write and carry out unit testing that supported the development process. In this report I have described how I aligned software requirements with the testing process, described how the tests were effective, technically sound, and efficient. Furthermore, I described the software testing techniques that I used in the project, as well as other available testing technique that I did not use. I provided characteristics of the testing techniques, as well as their practical uses and implications.

Finally, I have provided a synopsis of my testing mindset that I employed throughout the course. This included how and why to be cautious as a software tester and how to appreciate the complexity and interrelationships of code that I was testing. I also discussed how bias is introduced in testing and my strategy for limiting it. Lastly, I discussed the importance of being disciplined in my commitment to software quality, to include avoiding the trap of going into technical debt to satisfy short-term gains.

**References**

Boni García. (2017). *Mastering software testing with JUnit 5 : comprehensive guide to develop high quality Java applications*. Packt Publishing, Birmingham (Gran Bretaña.

Cowen, R. (1999). NASA Loses Mars Climate Orbiter. *Science News*, *156*(14), 214.

EBSCOHOST

Lloyd, R. (1999, September 30). CNN - Metric mishap caused loss of NASA orbiter –

September 30, 1999. [Www.cnn.com](http://Www.cnn.com).

<http://www.cnn.com/TECH/space/9909/30/mars.metric.02/index.html>

Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *SOFTWARE TESTING : an istqb-bcs certified tester foundation guide.* (B. Hambling, Ed.). Bcs, The Chartered Instit.

*The Challenges and Benefits of Model Based Testing*. (2019, October 21). Blog.frugaltesting.com. https://blog.frugaltesting.com/blog/2019/10/21/the-challenges-and-benefits-of-model-based-testing