# CS 320 Project Two

Jeba Singh Emmanuel

[jeba.emmanuel@snhu.edu](mailto:jeba.emmanuel@snhu.edu)

Southern New Hampshire University

February 25, 2024

Summary

1. **Describe your unit testing approach for each of the three features.**
   1. To what extent was your approach **aligned to the software requirements**? Support your claims with specific evidence.  
      My approach was very much aligned to the software requirements. I wrote the code in the classes first and then began writing JUnit code for the tests. When writing thet tests, I used the requirements as a reference to ensure each requirement had a corresponding test. For example, the ContactTest class has a test case for every requirement of which there are 5. Even in these 5 there are sub requirements which I have separate tests for. For example, I tested that the firstName string in a contact cannot be longer than 10 chars or null.
   2. Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were **effective** based on the coverage percentage?  
      My coverage % is >85 for all classes and in some cases >95%. This shows that for almost every line of code there is a corresponding test case that verifies the code functions as expected. The code coverage ensures that the tests exercise most lines of code.
2. **Describe your experience writing the JUnit tests.**
   1. How did you ensure that your code was **technically sound**? Cite specific lines of code from your tests to illustrate.  
      I used code coverage to ensure my tests were exercising lines of code. I also tested failure conditions such as throwing exceptions to ensure I wasn’t just testing happy paths.  
      Eg. *assertThrows*(Exception.**class**, () -> {defaultContact.setFirstName("Very Long Name");});

I also used real values for strings and for failure conditions to simulate bad input.  
Eg. *assertThrows*(Exception.**class**, () -> {

**new** Contact(**null**, "Jeba", "Emmanuel", "1234567890", "Location");

});  
  
 This gave me confidence that my tests were technically sound.

* 1. How did you ensure that your code was efficient? Cite specific lines of code from your tests to illustrate.  
     I grouped similar tests in a single method. Instead of writing different methods for each test scenario, I grouped related ones together. For example anything related to first name in the contact class was in a testSetFirstName method. This speeds up test run since setup and tear down doesn’t have to be done unnecessarily.

@Test

**void** testSetFirstName() **throws** Exception {

defaultContact.setFirstName("New Name");

*assertEquals*(defaultContact.getFirstName(), "New Name");

*assertThrows*(Exception.**class**, () -> {defaultContact.setFirstName("Very Long Name");});

*assertThrows*(Exception.**class**, () -> {defaultContact.setFirstName("");});

*assertThrows*(Exception.**class**, () -> {defaultContact.setFirstName(**null**);});

}

Reflection

1. **Testing Techniques**
   1. What were the **software testing techniques** that you employed in this project? Describe their characteristics using specific details.

I used statement and decision testing – whitebox testing in this project. More specifically I used unit testing. This kind of testing exercises statements in code. Tests are designed to force the program to execute certain statements by modifying input and verifying if predicted behavior/output is seen.

* 1. What are the **other software testing techniques** that you did not use for this project? Describe their characteristics using specific details.

I did not use Experience based techniques such as Error guessing, exploratory testing or checklist-based testing. Since the software is relatively simple, I did not need to but for more complex projects I would sit down and think of experience-based testing I could do.

I also did not use blackbox testing such as equivalence partitioning etc since the requirements were quite clear. For more complex projects I would try to predict possible values and test them.

* 1. For each of the techniques you discussed, explain the **practical uses and implications** for different software development projects and situations.
     1. Unit Tests: They are used to verify behavior of code you have built and helps with regression testing as you make future changes.
     2. Experience based Tests – This is useful for more complex projects where the interconnectedness and logic cannot all be documented. Experienced testers use common sense and experience to map out test scenarios.
     3. Black box testing – This is useful when you either do not know how the implementation is done or the output range is too large.

1. **Mindset**
   1. Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ **caution**? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.

I adopted a very methodical mindset on this project. I reviewed the requirements, then wrote code. I then reviewed the requirements and then wrote tests. I was cautious when writing tests as I paused and thought about how I may have missed some requirements. I spent more time reading requirements during testing than coding. For example, I realized when I was working on code coverage that I hadn’t tested null inputs even though it was in the requirements. I tested empty strings which weren’t in the requirement. It is therefore important to realize that even simple code can have bugs and testing needs to be done carefully.

* 1. Assess the ways you tried to limit **bias** in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims.

I could try to hide bugs if there are incentives to my code not having bugs. For example, if developer raises are tied to quality of code, developers would argue about every bug. I could also think having bugs was a sign of weakness and my mind would hide bugs from me. It is also possible to just not think beyond what your code does since the mind thinks it understands the problem. One way around that is to take a break between coding and writing the tests.

* 1. Finally, evaluate the importance of being **disciplined** in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.

Quality is important because the earlier you can find problems, the cheaper it is to fix. Less people are involved, less effort is needed to make changes. Quality also shows how good a craftsman you are. Cutting corners therefore inevitably leads to more work than doing things right away.