Justin Lakey

SNHU CS 320

Prof. Antoun

October 19, 2025

**7-2 Final Project**

**Summary: Alignment to Requirements**

The approach for the mobile application was driven by the customer requirements. Each customer requirement was implemented in the code. For example, there were requirements in the Contact class to have a firstName and lastName field. Each of those fields were implemented in the Contact class. Each of the name fields could not be longer than 10 characters or null, based off the customer requirements. In order to meet this requirement, we developed methods to validate the length of the name fields and that they are not null.

**Summary: Effective Tests**

The customer had requirements for certain fields. From each of those requirements, we developed JUnit tests to ensure we met the requirement. It is impossible to know if your tests expose every possible error in the code, but a good indication of effective tests is total test coverage. Coverage means how much of the code was actually ran by the JUnit tests. For this project, my JUnit tests had 100% coverage of the source code and 89.2% coverage overall (including all test code).

**Summary: Technically Sound Code**

To verify my code was technically sound, I ran all my JUnit tests multiple times. The first time some of the tests would fail. Sometimes this was because the source code was not actually meeting the customer requirements or had an error. Other times, the JUnit test itself had an error. For example, when testing the phone number I wrote a test to see if the phone number was too short (the number should be exactly 10 characters according to customer requirements). I copied the test code from a previous test but forgot to change the test String. Here is the example code with the incorrect test String (this String is 10 characters long and would not throw an exception):

*@DisplayName*("Test setPhone with too short phone")

*@Test*

public void should\_ThrowIllegalArgumentException\_When\_GivenTooShortPhone() {

String testPhone = "1234567890";

Assertions.*assertThrows*(IllegalArgumentException.class, () -> {

testContact.setPhone(testPhone);;

});

}

Once all the JUnit tests passed I looked through the passing results and the coverage to ensure they were actually testing the code.

**Summary: Efficient Code**

Efficient JUnit tests can be difficult to implement. Also, accuracy should never be sacrificed for efficient tests. That being said, to make sure my tests were as simple as possible while testing all the requirements I wrote one test for each requirement. In the below code I copied the same test and changed the test String for each requirement. Here are excerpts from my tests that show the different String used:

public void should\_ReturnPhone\_When\_GivenGoodPhone() {

String testPhone = "1234567891";

public void should\_ThrowIllegalArgumentException\_When\_GivenTooLongPhone() {

String testPhone = "12345678901";

public void should\_ThrowIllegalArgumentException\_When\_GivenTooShortPhone() {

String testPhone = "123456789";

public void should\_ThrowIllegalArgumentException\_When\_GivenNullPhone() {

String testPhone = null;

**Reflection: Techniques Employed**

We created objects and services and unit tests for them. Unit tests are ideal for testing individual objects, classes, and methods. For the object classes I wrote unit tests for the constructor and each of the getters and setters for each class variable. For the constructor there were unit tests to ensure the constructor worked when supplied with good parameters. The constructor was also tested to ensure it would throw exceptions if the parameters were invalid. In order to test the parameter validity I created a method to validate each parameter. Each of those validate methods also had a unit test.

**Reflection: Other Techniques**

Some other techniques include system testing and integration testing. Integration testing is used to ensure that all the pieces integrate together correctly. Integration testing is especially useful for APIs when different systems communicate together. Another aspect of integration testing is integrating software onto the hardware and checking for correct inputs and outputs. System testing is when all the pieces are brought together into a complete system. This could include integrating various functions such as GUIs, web applications, databases, network or cloud functions, and hardware.

**Reflection: Uses and Implications of Techniques**

Testing should be done at every level, hence there are various testing techniques. Unit tests are the smallest tests. The unit tests are designed to test individual classes and methods. Unit tests should not need any information or data from any other source. They are self-contained tests. Integration tests are the next higher level test above unit tests. Integration tests are performed when two or more modules are connected together. Each of the modules should have already passed their respective unit tests. System tests are the highest level of test. System tests are performed when the entire system is complete. System tests can integrate all of the hardware components such as user interfaces, networks, and databases as well as hardware components and even integrate with external systems. All of the components that make up the system should have been tested in unit tests and integration tests between units.

**Reflection: Caution**

Caution should be used when testing code to ensure that all possible errors are checked. It is important to check both that code works correctly and to verify when the code should throw exceptions. For example, when testing a constructor it is important to make sure the constructor works, but it is also cautious to ensure that if an invalid parameter is passed to the constructor that it correctly throws an error, even though the parameters are validated in separate methods. In this example the constructor is passed an invalid first name:

*@DisplayName*("Test Contact constructor with bad firstName")

*@Test*

public void should\_ThrowIllegalArgumentException\_When\_IncorrectFirstNameInContactConstructor() {

String contactId = "1";

String firstName = "Justin Lakey";

String lastName = "Lakey";

String phone = "1234567890";

String address = "101 Fake Street, Capitol City";

Assertions.*assertThrows*(IllegalArgumentException.class, () -> {

new Contact(contactId, firstName, lastName, phone, address);

});

}

**Reflection: Bias**

Bias is difficult to avoid when developing and testing your own code. That is why it is imperative to have a separate team member review and test your code. Some techniques to reduce bias in your own testing is to systematically check each field, method, class, and interface. Each check should ensure that it works as expected, the edge cases, and any known exceptions. Unknown bias can still occur when a developer becomes unaware of the potential faults in their code. This can happen because you assume the code is working as expected, but you do not know what you do not know. For example, when I copied code from the firstName field to test the lastName field, I forgot to change all the variable names. I did not notice the last variable testFirstName in the assertThrows method as shown below:

*@DisplayName*("Test setLastName with too long lastName")

*@Test*

public void should\_ThrowIllegalArgumentException\_When\_GivenTooLongLastName() {

String testLastName = "Justin Lakey";

Assertions.*assertThrows*(IllegalArgumentException.class, () -> {

testContact.setLastName(testFirstName);;

});

}

**Reflection: Discipline**

Discipline is important as a software engineer because even though team work is common, it is normal for a software engineer to develop and test their own code first. Depending on your development team, you may not always have others to thoroughly review all your code. It is up to you to consider customer requirements, both explicit and implied. For example, at my work we have lots of requirements that may not specify all the conditions. There recently was a requirement to login to the app within certain time frame. The system made this time requirement, but not when switching between users. Even though there was not a requirement for this specific case, engineering discipline required that we followed up with the customer to check the requirement for this particular case.

Another area where discipline comes to play is in your own code repositories. It is important to spend the time and effort when first developing code to make it as user friendly as possible. Well organized and clearly commented code reduces the technical debt for the entire team if the code has to be revisited in the future.