CS320 Summary and Reflections Report

Mitchel Dauk

Southern New Hampshire University

Table of Contents

[Table of Contents 2](#_Toc121853840)

[1. Summary 3](#_Toc121853841)

[1a. Describe your unit testing approach for each of the three features. 3](#_Toc121853842)

[1b. Describe your experience writing the JUnit tests. 7](#_Toc121853843)

[2. Reflection 9](#_Toc121853844)

[2a. Testing Techniques 9](#_Toc121853845)

[2b. Mindset 10](#_Toc121853846)

[References 13](#_Toc121853847)

CS320 Summary and Reflections Report

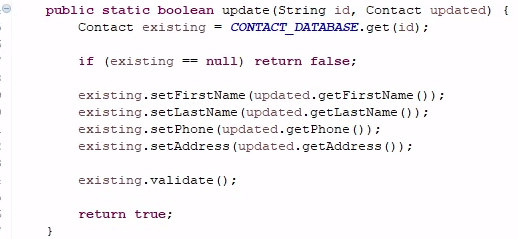
# 1. Summary

## 1a. Describe your unit testing approach for each of the three features.

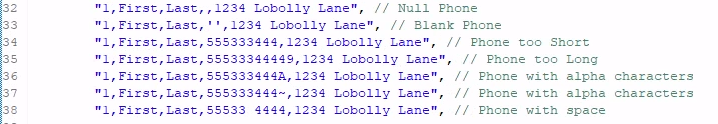
### Contact Entity and Contact Service.

#### To what extent was your approach aligned with the software requirements?

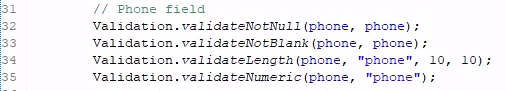
The Contact Service had requirements that required the id, first name, and last name. They couldn’t be longer than 10 characters or be null. The phone string required the number should be exactly 10 digits and not blank or null. The address required the field did not exceed 30 characters, or be blank or null. First name, last name, phone number, and address should have the option to be updated. The id should be able to add and delete contacts, but not update. This option makes the id immutable and it cannot be changed, as seen below.



Testing the fields pushes the tester to come up with scenarios where the program could have problems. For example, the phone number had to be 10 digits, could not be blank, and could not be null. All fields will pass the test, except the phone number will be blank. Another test will allow the phone number to be null. When it comes to 10 digits for a phone number, I as the tester, had to think logically. Could a phone number have 9 digits? How about 11? Can one of the digits be a character? I needed to write a test that put all of those parameters into place.



After the names inside the class were set, the parameters were then put in place to show that a phone number wasn’t null or blank, the length was correct, and that all fields were numerical digits.



#### Defend the quality of your JUnit tests.

The JUnit tests were effective after testing for coverage. After testing, I reviewed the code and saw that green meant the tests were covered, yellow meant there was a partial test, and red meant that there wasn’t a proper test. This feature is an easy way to find what isn’t covered or can be made better. When validating JUnit tests and coverage, we are making sure that the tests are covered for the program being made. Without this, the user will have free reign and will leave the system susceptible to errors and security breaches.



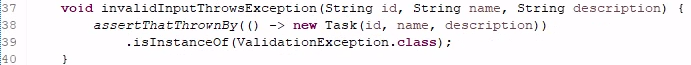


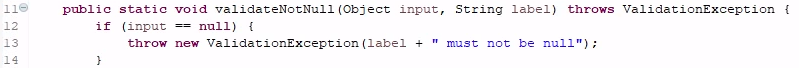
### Task Entity and Task Service.

#### To what extent was your approach aligned with the software requirements?

The Task Service requirements were to make sure the id, name, and description were not null or blank. The id couldn’t exceed 10 characters, the name couldn’t exceed 20 characters, and the description couldn’t exceed 50 characters. These tasks were a bit easier because they didn’t have a set amount of characters, nor a limit to only numbers. The id, again, needed to be immutable and not able to update, unlike the name and description. The entities were to be able to be stored and deleted.

Each of the fields were tested by throwing exceptions. A separate exception class was made to allow the tests in the main class. For example, the fields must not be null. This validation suggests that as long as we set a parameter for not being null, the test will pass.







#### Defend the quality of your JUnit tests.

The JUnit tests were effective after testing for coverage. After testing, I reviewed the code and saw that green meant the tests were covered, yellow meant there was a partial test, and red meant that there wasn’t a proper test. This feature is an easy way to find what isn’t covered or can be made better. When validating JUnit tests and coverage, we are making sure that the tests are covered for the program being made. Without this, the user will have free reign and will leave the system susceptible to errors and security breaches.



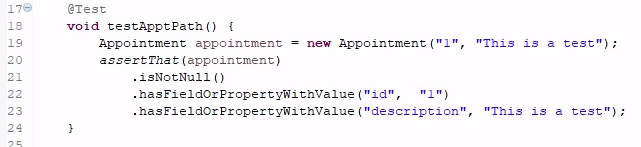


### Appointment Entity and Appointment Service.

#### To what extent was your approach aligned with the software requirements?

The Appointment Service requirements were to make sure the id, date, and description were not blank or null. The id couldn’t have more than 10 characters, the date could not be in the past, and the description could not be more than 50 characters. The id was again immutable and not updatable. The fields must also be able to be stored and deleted.

Since I had trouble with the date, I allowed the date to be in a separate class and the id, along with the description, to be in the main class.



Since the fields were nearly identical to the other tasks, they followed the same protocols as the Contact and Task services.





#### Defend the quality of your JUnit tests.

The JUnit tests were effective after testing for coverage. After testing, I reviewed the code and saw that green meant the tests were covered, yellow meant there was a partial test, and red meant that there wasn’t a proper test. This feature is an easy way to find what isn’t covered or can be made better. When validating JUnit tests and coverage, we are making sure that the tests are covered for the program being made. Without this, the user will have free reign and will leave the system susceptible to errors and security breaches.

This test did not include the date, as I had trouble coding the proper functions. If I had left the date broken, the tests would not have run properly and the coverage would have fallen under 80%. When coding with JUnit tests, it is important to include all areas if you are specifying them in the original test class.

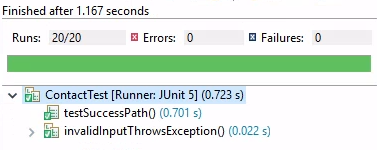




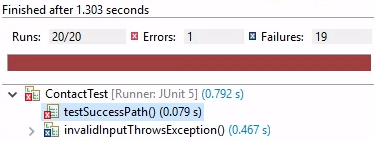
## 1b. Describe your experience writing the JUnit tests.

### How did you ensure that your code was technically sound?

In order to find out if the tests are successful, a JUnit test is implemented into the class. Below, the Contact class is tested to find if the tests are covered in the main class.



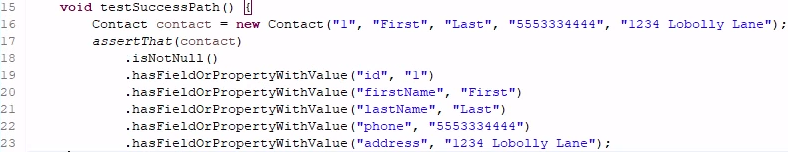
If the JUnit test fails, there will show an error and what is wrong.



Each test had to be specific to the requirements. The statement below states that the phone number should be 10 characters or digits, and 10 only.



The “assertThat” statement tests assumptions in the code. The fields are named so they can be properly tested.



What if the number had a space, or an alpha character?



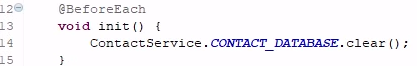
The reasons for the tests allowed independence. Above, the phone number could only be 10 digits.

### How did you ensure that your test code was efficient?

The JUnit tests show the efficiency and amount of errors within the code.



The JUnit’s timeout feature was not used, but @BeforeEach and @ParameterizedTest was used to disallow duplicating code.







A separate test allows a certain section of code to be tested, through JUnit. Redundant code is shown on the coverage test as not needed.

# 2. Reflection

## 2a. Testing Techniques

### What were the software testing techniques that you employed in this project?

The software testing techniques that were used were white-box testing. White-box testing is a structure-based technique that test the components and how they interact with each other (Hambling, 2019). Within the tests, there were parameterized tests and validation exceptions, that were completed through static tests. Parameterized tests were used to test the requirements that the customer had. For example, the ID should not be blank or null. A specific class would have to show the ID being blank, so that it could be fixed with a specific test. Originally, the information can be tested by another class, but is best used through a validation exception.







### What are the other software testing techniques that you did not use for this project?

Other software techniques that were not used were black-box tests and experience-based tests. Black-box tests are behavioral-based techniques and focus on what the components do (Hambling, 2019). The reason black-box testing was not used is because the main purpose of a Junit test is to fulfill the customer’s requirements by checking all functions. Black-box testing focuses on behavior and no other aspects.

Experience-based techniques are tests that don’t have a specification but use experience to find the most important parts to test (Hambling, 2019). For example, instead of using Junit tests, the tester may use a checklist of common faults and run through it before executing the program. They won’t write a specific code, but use their instincts and experience.

### For each technique you discussed, explain their practical uses and implications for different software projects and situations.

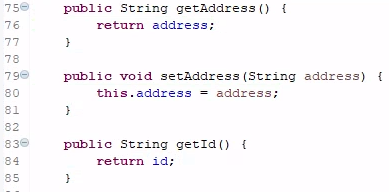
White-box testing is used for testing the whole structure, while black-box testing is used to test specific behaviors. For example, I may be writing a program for a clock. The white-box testing will focus on how the code is written and making sure the components match how it should be made. Black-box testing will make sure that the clock reads the correct time. Experience-based testing will only test parts that the tester thinks may be important or flawed, from previous experience.

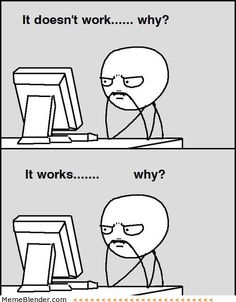
## 2b. Mindset

### Assess the mindset you adopted working on this project.

It is important, as a software tester, to employ caution. There are various levels and types of caution to be aware of when it comes to programming and testing. After each small section of code, or even each statement, it is important to stop and test. Not before long, the program can become saturated with errors and issues. Some issues might have been partially fixed with the next line of code, which makes it harder to identify certain problems.

The requirements should be taken seriously and brought to the attention at the beginning. By not allowing an id to be changed early on, it stops future headaches.





Each part of the code should be taken into consideration and treated as its own. Writing or copying code that you don’t understand, only leads to problems. A programmer should question when code works and when it does not work.

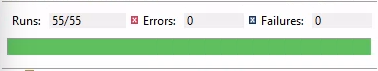
### Assess the ways you tried to limit bias in your review of the code.

Limiting bias allows reviewing code to be easier. Bias can refer to sharing knowledge, focusing on important tasks and not nit-picking small tasks, clear descriptions and communication, and thorough documentation (Brunfeldt, 2021). Bias could be favoring a certain person or a program entirely.

### Finally, evaluate the importance of being disciplined in your commitment to quality as a software engineering professional.

It is important to thoroughly investigate code because of not only the time it took to put together, but the time it takes to fix it. Imagine driving halfway to work, only to remember you forgot your lunch. You don’t spend just extra time to get your lunch, you spend the time it took to get halfway to work and then you retrack your steps. Often, this is the best-case scenario because backtracking a far amount can lead to more issues that were not taken care of properly.

To avoid technical debt, you must make sure that you are testing code and have an understanding on what you are coding. For example, if you don’t understand the code, but it works, you may have an error that is not easily traceable. JUnit tests are a great way to test the code and make sure there is proper coverage, without errors.





References

Morgan, P., Samaroo, A., Thompson, G., & Williams, P. Hambling, B. (2019). *Software testing: An istqb-bcs certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.

Hudgens, J. (n.d.). Top 5 programming memes. Brunfeldt, K. (2021a, October 12). *A complete guide to code reviews*. Swarmiacom RSS. <https://www.swarmia.com/blog/a-complete-guide-to-code-reviews/>

Hudgens, J. (n.d.). Top 5 programming memes. https://devcamp.com/site\_blogs/top-5-programming-memes