CS320 Summary and Reflections Report

Jonathan H. Boeglin

Southern New Hampshire University

Table of Contents

[Table of Contents 2](#_Toc117338013)

[1. Summary 3](#_Toc117338014)

[1a. Describe your unit testing approach for each of the three features. 3](#_Toc117338015)

[1b. Describe your experience writing the JUnit tests. 4](#_Toc117338016)

[2. Reflection 5](#_Toc117338017)

[2a. Testing Techniques 5](#_Toc117338018)

[2b. Mindset 7](#_Toc117338019)

[References 10](#_Toc117338020)

CS320 Summary and Reflections Report

# 1. Summary

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

### Contact Service.

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

The testing performed on the contact service was in line with the software requirements because it covered not only all the inputs in the contact entity and made sure they were not left blank or null, that non-existent entries could not be deleted, and a new entry could not be created that had the same ID as another one. In addition, the phone entry was created and tested to have exactly ten digits, since every number has an area code and seven other digits.

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

My JUnit tests were effective, because coverage of the Contact entity and service classes totaled 100.0%, meaning that every line of code in the two classes was read by the tests and acted upon in some manner. The only thing that needed improvement were the parameterized tests I set up to fail, as I used the wrong version of assert to make them.

### Task Service.

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

The task service testing was probably the easiest, aa there were no special fields for the task entity. This made conforming to the software requirements easier, as all that needed to be done was limit the number of characters allowed in each field and make sure none were left blank. These tests, like the other non-specialized ones, were done with parameterized tests that made sure the boundaries of the acceptable character limit were enforced.

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

These classes also received 100.0% coverage and had the same issues as the other two services, as I mostly copy-pasted the code to get them to be more uniform, with exceptions for this being the date and phone number entries. Either way, when the coverage test ran, the results were green for most of the lines.

### Appointment Service.

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

The appointment and AppointmentService test classes aligned with the software requirements because all the inputs in both were tested unit by unit with varied types of tests. An example of this is with the Appointment and AppointmentTest classes, where the required ID, date, and description of the appointment could be entered, and tests were run to make sure that the entries were not left blank, and that the correct limit to the number of characters was allowed.

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

The tests on these two classes also came to 100.0% coverage, but as with the others, the parameterized tests I set up to fail intentionally used the wrong assert type. However, the tests still gave feedback that I expected, and read every line of the classes. The one tough test to create was the correct date test, as until the announcement was posted, I kept receiving warnings that the Date class was outdated and to use Calendar instead. After the post, however, it made a lot more sense.

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

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

I believe the code is technically sound because it is logical, concise, and modular. It is logical because it completes all the requirements of the project in as simple of a way as possible. It is modular, because each test runs in the order that the entry into the entity would occur. For example, in the Task class the order of entry is ID, name, and then description, and in the TaskTest class the test order is to make sure the task isn’t empty, and then the length is correct of the ID, name, and description, in order. Finally, the code is concise because of the coverage of testing and the level of efficiency achieved.

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

The code I created was efficient because I was able to employ parameterized testing and the record class type in the project. Parameterized testing allowed me to run a single test several times with different inputs, cutting down on the number of tests total and having to repeat tests for different values. An example of this would be the ContactTest class, which had parameterized tests for each entry in the contact entity. Without these tests, the number of tests written would jump from 6 to 22. The record class type allows major shearing of the bulk of code needed to set up an entity, with the ability to just enter the contents of the entity and parameters to help shape it, instead of all the getters, setters, and constructor you would normally need. An example of my use of this is the Contact class, where the information and constraints are easily read and laid out in a few lines. This was extra helpful with this entity, since it had the most entries needed of the three.

# 2. Reflection

## 2a. Testing Techniques

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

The software techniques used in testing the code included static and dynamic testing types, and were white box, grey box, and unit testing. Static testing refers to checking the program manually for syntax and other somewhat obvious errors, and dynamic testing refers to running the code to see if it compiles and runs without throwing errors. A general example of static testing that I completed was looking over the code to catch any syntax errors or other errors that were caught by Eclipse. A general example of dynamic testing done to my code is when the code was compiled and run by JUnit in Eclipse to see if the tests I created completed successfully. White box testing is the act of creating tests to run while knowing how the code itself functions, and grey box testing is a combination of knowing and not knowing how the code works and writing tests from that perspective. Examples of white box testing are some of the tests I wrote for each of the services and entities, since I also wrote the code and understood how they were supposed to function. An example of grey box testing is implementing tests for the functions using the HashMap. This is because while I vaguely understand how a hash map works off the top of my head, I do not know how the specific code is set up in this instance. Finally, unit testing is the act of creating tests by breaking the code being tested into smaller chunks, and an example of this would be writing tests for each method in a service, rather than a single test for the entire class.

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

Other software techniques not used include black box, automated, regression, exploratory, functional, and useability testing. Black box testing is testing done from a completely outside perspective. Unlike white and grey box tests, black box tests are run without any knowledge of the inner workings of the program. Automated testing uses a third party’s software to test the code. Regression testing involves making sure new code added to a program still allows old tests to pass, and exploratory testing is the act of creating tests on the fly, instead of planning out beforehand the structure of the testing system. Functional and usability tests involve making sure the program can accomplish the goals of the software from the user’s perspective, and that the interface for doing so isn’t obtrusive and hard to use.

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

Each of these tests have practical uses outside of this project, as do the ones not utilized. Static testing in general is good for cleaning up code and making it readable, and dynamic testing makes sure code functions. White, black, and grey box testing helps with making sure the software is stable, meets user needs, and can be deciphered for further testing if necessary. Automated testing is good for large companies that cannot add a dedicated QA team to their roster, and regression testing for companies that release constant updates and patches to their already live software. Unit testing helps catch and pinpoint errors in code and can allow the rest of the program that is not dependent on it to function normally. Exploratory testing allows fresh eyes to actively look through software and see if they can find errors otherwise missed, like with people participating in open betas. Functional testing is always needed, as with most software the client will not have access to anything but the user-side, so its goals should be easily attainable from there. Finally, usability is especially important in videogames, as a bad user interface can make the game unplayable or at least less fun to play.

## 2b. Mindset

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

I would say that I employed caution fairly regularly, especially once I started creating the tests themselves. While creating the code, I would write large chunks and then go back and make sure that at least statically, it looked okay. After starting to create tests, and not really understanding at fist how to do so, I became more cautious and started testing with JUnit much more often. It is important to appreciate the code’s complexity for the same reason that caution is advised, which is that one small misstep in writing out a test or even base code and the entire program can stop functioning correctly. Healthy appreciation of how wrong things can go with just an accidental addition or deletion of something is necessary to help discover errors before they become too much of a problem. Good examples of this are with the creation of the different tests. For ease, I would copy and paste a test and just change the name and what service it referenced. If I hadn’t been cautious and checked to make sure everything was renamed correctly and duplicates didn’t exist, I could have had two tests in different classes testing the same thing, while completely skipping over the area I thought I was testing.

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

Bias can be a large part of improper testing, for various reasons. If you are testing your own code, you might skip testing things you believe don’t need to be tested, whether it is because you believe you could not cause an error in creating the specific code or feel the code is too simple to fail. Another reason is believing someone would never choose to enter something that could cause an error. An example for me would be with the entities. I would not have thought to create a separate test to test the entity itself before, as I figured just testing the service would be coverage enough. Another example is testing to make sure a date is entered that exists in the future. Most people would know to enter a future date, but accidents happen, so therefore testing to make sure it blocks this from happening is actually necessary.

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

It is important to not cut corners while writing and testing code, for the same reasons that having a bias in testing is a detriment to proper test coverage. Skipping over anything in the testing process can result in major issues later in the program’s life. Personally, I will try to avoid technical debt by using automated tests that are trusted to have decent testing coverage, and when I must create tests myself, make them as uniform as possible to be useable more than once, and break it into units so that it is easier to parse. This will help in cases of needing to rush because I will have a portfolio of basic tests I can pull from in the future that will help save time in creation for things that need specific care.

References

Boog, J. (2019, September 5). *9 types of software testing in software engineering*. The QA Lead. Retrieved November 27, 2022, from https://theqalead.com/test-management/types-of-software-testing/

ProductPlan. (2021, September 2). Technical debt. ProductPlan. Retrieved December 11, 2022, from https://www.productplan.com/glossary/technical-debt/