CS320 Summary and Reflections Report

Jonathan M. Kleven

Southern New Hampshire University

Table of Contents

[Table of Contents 2](#_Toc127729299)

[1. Summary 3](#_Toc127729300)

[1a. Describe your unit testing approach for each of the three features. 3](#_Toc127729301)

[1b. Describe your experience writing the JUnit tests. 11](#_Toc127729302)

[2. Reflection 13](#_Toc127729303)

[2a. Testing Techniques 13](#_Toc127729304)

[2b. Mindset 14](#_Toc127729305)

[References 17](#_Toc127729306)

CS320 Summary and Reflections Report

# 1. Summary

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

### Contact Service.

The software requirements were relatively simple and pairing the Contact with Jakarta validation constraints helped ensure the Contact object was readable; additionally, the annotations helped reduce unnecessary logic. For example, “@NotBlank” requires neither a null nor blank parameter, one annotation that eliminates two branches of logic. The annotation “@Size” was used to limit the length of each string to a maximum value. Additionally, the use of regex and the annotation of “@Pattern” for the phone number allowed a pattern to determine the format and string length to align with the requirements. The simplicity of annotations enhanced the code readability of each parameter and restricted each of the required entities and their parameters to the format and length prescribed in the software requirements.

Text

Description automatically generated

The Contact Service uses an in-memory data structure. To satisfy that, I chose to use a concurrent hash map. This hash map uses the entity's id to map the Contact's location. Since each ID is meant to be unique, this will safeguard against collisions if another task or Contact is created that attempts to use an existing location. Therefore, the unique ID was used as the key and the entire Contact entity as the stored value.

Text

Description automatically generated

The requirements of the Contact Service are to create Contacts with a unique ID, delete Contacts using the ID, and update Contact fields using the ID – note: only the first name, last name, phone number, and address of each Contact can be updated.

Text

Description automatically generated

When a new Contact is created, that Contact entity is first passed into an object validator to ensure the entire Contact entity is not null. Then the Contact entity is reviewed against the current hash map for IDs that match the one attempting to be created. If a duplicate is found, an error message is given. Otherwise, the Contact will be added to the HashMap under the Contact ID as the key. The same is true for Contact entities to be deleted; the Contact ID is searched in the hash map before ensuring it exists in the data structure. Once the ID is matched, that key-value pair is removed from the hash map.

Text

Description automatically generated

Finally, the Contact Service allows updating each of the existing Contact entities. Like the delete method, the hash map is reviewed for an existing ID that matches the one given. A message is passed to the user with that information if it is not found. If the ID matches a key in the data structure, that value is replaced with the new Contact entity value that was passed.

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

Understanding the possible inputs a user can pass is essential to determine which tests to run. Using the parameterized test feature of JUnit allows for tests to repeat the code and run with different versions of possible inputs. This allows for more concise coding and readability. For example, I tested the Contact entity and the Contact Service. The entity has specific parameters and requirements to test, whereas the Service tests the methods of creating, deleting, and updating to test. Compartmentalizing the tests is more concise if features or requirements are changed later. Then the corresponding tests can be updated or additional tests added.

The Contact entity and its Services requirements were used as a baseline to test from. For example, the Contact has a unique ID that is less than 10 characters and cannot be null or updateable. To ensure this, the ID parameter provides an error message should the user attempt an invalid input. Tests using various inputs that are valid and invalid (i.e., more than 10 characters or an ID that already exists, etc.) were employed for the variety of tests. This method is called Equivalence partitioning.

Overall, 60 tests were performed on the Contact entity with a coverage of 54.5%. However, it should be noted that one test failed. This test (testPhoneNumber\_Blank) proved to be challenging, as I attempted to pass multiple versions of blank strings, which I assumed the test counted the spaces as characters which would have changed the error message from “Phone Number is a required field” to “Phone Number must be exactly 10 digits.” Therefore, I chose to have the variation of this test where multiple blank spaces were entered be the focus of this test and achieved 2 out of 3 successful tests performed.

Graphical user interface, text, application, chat or text message

Description automatically generatedChart, bar chart

Description automatically generated

The Contact Service test had 39 tests run without failures or errors. The Contact Service also had 81.8% coverage. The test runs on the Contact Service employed multiple tests on each method. These tests covered valid and invalid inputs. Additionally, the tests on the Contact Service used valid data introduced into an existing in-memory data structure to test delete and update Contact entities.

Graphical user interface, text, application, chat or text message

Description automatically generatedA screenshot of a computer

Description automatically generated with medium confidence

### Task Service.

The Task entity and Service were simpler to execute than the Contact entity and Service. The requirements of the parameters for the task entity are strings of a max character length that are not null or blank. Like the Contact entity, I used annotations from Jakarta validation constraints to make the entity more readable. The parameters for the Task entity are string values of a unique ID, a Task name, and a Task description.

Graphical user interface, text, application

Description automatically generated

The Task Service uses three methods to create Tasks, delete Tasks, and update Tasks – where the Task name and description are the only updateable fields. Like the Contact entity and Service, the Task Service uses a hash map with the key as the unique ID, and the value is the entire Task entity. When a new Task is created, that Task entity is passed into a validator to ensure the whole Task entity is not null. Then the Task entity is compared with the current hash map for matching IDs. If a duplicate is found, an error message is given. Otherwise, the Task will be added to the hash map under the Task ID as the key. Similarly, the Task entities are deleted only if the Task id is found in the hash map. Then that key-value pair is removed from the hash map.

Finally, the Task Service can update existing Contact entities. Like the delete method, the hash map is searched for an existing ID. If the ID matches a key in the data structure, that value is updated with the new Task entity value passed.

The JUnit tests for the Task entity and Service looked at valid and invalid inputs passed by each parameter and Service method. Using lessons learned from the Contact entity and Service tests, I duplicated the efforts and revised the tests to ensure they were testing the appropriate Task entity parameters. The parameters were tested to ensure that no null or blank input would be accepted and valid and invalid input for each parameter to test for string length. With these tests, I began using a function to print or output the hash map to ensure the data being tested was reflected in the data structure. This allowed me to better understand how the tests interacted with the Task entity and Service to ensure they aligned with the requirements.

The tests of the Task entity employed 30 tests with no failures or errors while covering 57% of the Task entity. Whereas the tests of the Task service ran 22 tests with a coverage percentage of 77.9%

TaskTest Coverage

Chart, bar chart

Description automatically generated

TaskService Coverage

Chart, bar chart

Description automatically generated

### Appointment Service.

The Appointment entity and Service used the same method employed in the Contact and Task entity and Service. The requirements of the parameters for the Appointment entity are strings of a max character length that are not null or blank, as well as a date that is not null or blank but also not a past date. Like the Contact and Task entity, using Jakarta validation constraints, annotations simplify much of the code while ensuring that the parameters align with the software requirements. The specific parameters are Appointment ID, Date, and description.

Text

Description automatically generated

The Appointment Service uses two methods to create and delete Appointments. Again like the Contact and Task entity and Service, the Appointment Service uses a hash map to ensure only unique IDs are added. When a new Appointment is created, that Appointment entity passes through a validator to ensure the entire Appointment entity is not null. Then the Appointment entity is evaluated with the current hash map for IDs. If a duplicate is found, an error message is given. Otherwise, the Appointment will be added to the hash map under the Appointment ID as the key. Likewise, the Appointment entities are deleted or removed from the in-memory data structure. Again, if the Appointment id is found, that key-value pair is removed from the hash map.

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

The JUnit tests for the Appointment entity and Service use Equivalence partitioning. This type of testing splits up the testing work into two categories – valid and invalid. The parameters were tested to ensure that null and blank input would not be accepted and valid and invalid input for each parameter to test for string length. The date parameter was challenging because I was unfamiliar with the Java LocalDateTime module. I had to learn how to pass the information and format the input correctly. Once I understood it, I could test for various dates and boundary tests against the software’s requirement that the date could not be created in the past. In my review of my implementation, this posed another hurdle: I found that LocalDateTime.now() function calls the time on the local date and time from the machine and then runs the test. As such, the time from the call to running the test meant that the call was already in the past, even by a few milliseconds. Further research and testing showed that dot stringing an additional function, “.plusSecond(1)”, allowed for enough time between the call and the test to allow for testing a present or left-most boundary.

The Appointment entity employed 25 tests with no failures or errors while covering 57.1% of the Appointment entity. Whereas the tests of the Appointment service ran 16 tests with a coverage percentage of 100%.

AppointmentTest Coverage

Chart, bar chart

Description automatically generated

AppointmentService Coverage

Chart

Description automatically generated

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

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

In the beginning, or for the first milestone, to ensure that my JUnit tests were technically sound, I relied on the tutorial's guidance to help break ground and to start seeing how the tutorial could be revised to meet the software requirements. I would write each test one at a time and run them. Then, depending on how far along I was, I would run just the one test that I was working on to compare the result with the expectation. I began to include parameterized tests for two reasons 1) I could run the same test on multiple inputs, and 2) it required the format of the test to be slightly different, and I liked being exposed to different approaches.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text

Description automatically generated

Many Contact, Task, and Appointment entities had string parameters with a maximum limit of strings. Boundary testing was used to test valid and invalid inputs, i.e., characters at the limit, below the limit, and blank or null. Each string parameter required the input of a string - this made it hard to attempt to pass an integer, despite some effort. The other challenges were testing the unique ID and the Contact phone number entity.

The phone number entity utilized regular expressions to establish a pattern of strings that were required to be digit characters of a length of ten. Again, Boundary tests attempted to pass non-digit characters and lengths that were not equal to ten. I struggled with testing a variety of blank strings of varying lengths of spaces (zero, one, and three). In retrospect, I may have overcomplicated this test and added more than was necessary. Additionally, had the pattern been more complex, it would have required additional tests for a more complex pattern. Finally, ensuring only a unique ID was saved in memory with the use of a hash map, being collision resistant against more than one key-value pair with a matching key.

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

Ensuring that the test code was efficient was difficult. There were several instances when parameterized tests were used and didn’t need to be, such as in the Contact phone number example above. In other cases where parameterized tests were used, they tested a similar length string but with different characters. I suppose then that parameterized tests of the entities would be better served if there was more logic within the unit test and a boundary test set up to group valid, invalid, and/or out-of-bounds parameters.

# 2. Reflection

## 2a. Testing Techniques

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

In my tests, I chose to split up each test, testing one aspect at a time. I opted to follow the professor's example code as I saw that implementation was more complex than I was used to in past Java projects. Consequently, annotations in each entity appeared more concise than using setters and getters and implementing the appropriate logic for each. It made a more straightforward entity, and the restrictions placed by the annotations looked much cleaner and saved several lines of code. With that said, the interaction with the entities and services with the tests may have had the same outcome had setters and getters been employed but troubleshooting the entities seemed easier using the annotations.

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

One technique that could have been implemented was grouping. This technique could have set all the inputs to be tested under one test; for example, all of the services have a create and delete method, and all of the various inputs could be grouped into one create and delete test. Troubleshooting a grouped test would not be complicated; however, it may be tedious to try and identify which line of the assertions within that group had failed and what aspect of that assertion was incorrectly set up.

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

Understanding the requirements for each entity and how to test them for each milestone, one can see how they would apply to other software development. No matter what the project, there needs to be some data that needs to be validated and verified. Having just the entity and Service provide verification and validation is essential, but ensuring that the entity and Service are doing it properly is the purpose of tests.

Based on past coursework and personal projects, I spent a lot of time writing and checking the application to ensure what I added didn’t break or did what it was meant to. These tests within the milestones show a much quicker method to do the same thing. It also builds confidence in the entities and services that will perform as I intended once they are run. Additionally, the practical aspects of ensuring the code align with the software parameters and runs as intended allow for the growth of the software to improve the design and add additional features later on.

## 2b. Mindset

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

As I noted previously, I employed the parameterized test, which may not have been necessary for the individual test. However, testing the same data type multiple times despite its configuration or format leads to excessive and ineffective coding. In addition, it reduces the overall efficiency of the tests by expending unnecessary resources “… concerning the accuracy and completeness with which users achieve goals.” (Boni Garcia. 2017).

Using a hash map in each Service allowed the interaction of the Contact, Task, and Appointment entities to work with their respective Service to address the unique ID requirement. In addition, this method avoided the need for additional logic in either the entity or the Service.

Where I did not employ caution was not continuing to improve my Contact entity testPhoneNumber\_Blank test. While I tested various ways to make a parameterized test succeed, I left the test as is. It was poor judgment on my part and perhaps elicited some “code smell” or “undesirable symptoms within the source code.” (Boni Garcia. 2017).

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

I tried to keep to the requirements as best as possible. However, I was often critical (especially in the first milestone) of how this would interact with a user interface. So much of what we see is the interface and not the backend. The update method in each of the Services was one implementation that took me a while to reconcile. My bias was that there should have been an update method for each of the parameters of the corresponding entity. That way, only one parameter for each method would be updated rather than replacing the entire entity with one or all parameters changed. With my update method, additional logic or code would need to be added to ensure that data is not lost for the parameters that were not meant to be updated. Although, much of that could be handled by the interface where all of the fields of the existing entity are presented, and only the fields that were changed are saved.

Text

Description automatically generated

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

In my previous example relating to the Contact entity testing an invalid input of multiple space characters, I thought I was diligent in covering more than one option of blank inputs. Where I lacked discipline, however, was leaving this test alone and not finding a solution. This deficiency cost me points but is a technical debt that could have been remedied before submission. The takeaway from this is that the importance of testing the requirements is not merely a check in the box of the software satisfying the minimum requirements. Simply revising the test to have one blank input would have ensured the tests success but would not have identified an issue with inputting multiple space characters. Despite that, understanding what the JUnit test report is reporting and taking action on that report is what a sound practitioner who doesn’t cut corners should do. The Contact entity testPhoneNumber\_Blank test makes apparent that the entities parameter required further review and revision to account for multiple spaces.

In a cold run of the JUnit test, meaning the first run after loading the project, you can see below that all tests failed.

Graphical user interface, text, application

Description automatically generated

Subsequently, the very next run of the test produced different results. Which again tells of an underlying issue with the test or entity itself. It should be noted that an error message was passed. However, it was not from the @Blank annotation, which was the expected message written into the test but from the @Pattern notation.

Graphical user interface, text, application

Description automatically generated

In general, errors like the one above that are identified and reported to the software engineer should be dealt with as soon as there is a finding. If not, the project can be set back by additional features or integration with other aspects of the program that need this parameter to respond in a certain way. Furthermore, not testing the software requirements fully and to completion, causes rework and will hinder a timely release. This reduces the programmer's effectiveness and the team they are working with.

References

Boni Garcia. (2017). *Mastering Software Testing with JUnit 5 : A Comprehensive, Hands-on Guide on Unit Testing Framework for Java Programming Language*. Packt Publishing.