CS320 Summary and Reflections Report

David Faulkner

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. 6](#_Toc121853843)

[2. Reflection 7](#_Toc121853844)

[2a. Testing Techniques 7](#_Toc121853845)

[2b. Mindset 9](#_Toc121853846)

[References 11](#_Toc121853847)

CS320 Summary and Reflections Report

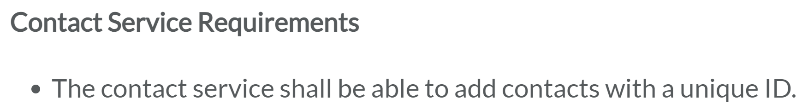
# 1. Summary

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

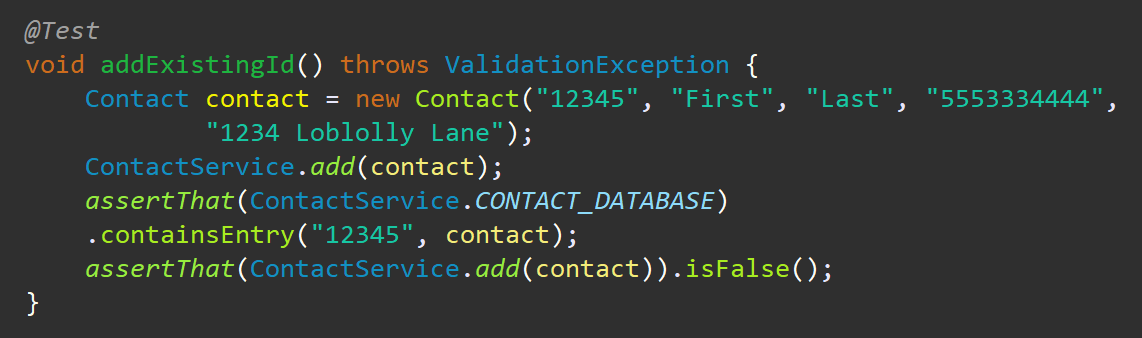
### Contact Entity and Contact Service.

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

100% percent of software requirements were tested to ensure quality. For example, in addition to the success path of each contact having a unique ID as required in the Software Requirements (Figure 1), the failure path was tested in case a contact was entered with an ID that already existed (Figure 2).



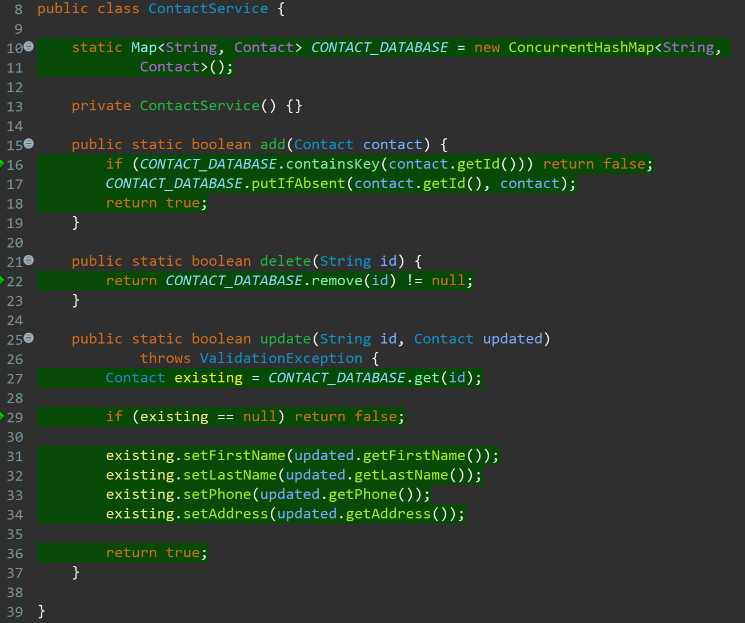
*Figure 1 Contact Service Requirement Number One*



*Figure 2 Contact Service test ensuring unique IDs are added*

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

100% code coverage for the ContactService class was provided as shown in Figure 3 and Figure 4:



*Figure 3 ContactService class with 100% code coverage*

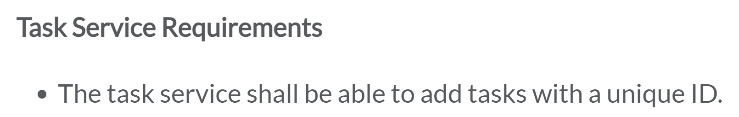
**

*Figure 4 ContactService class code coverage as a percentage*

### Task Entity and Task Service.

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

100% percent of software requirements were tested to ensure quality. For example, in addition to the success path of each task having a unique ID as required in the Software Requirements (Figure 5), the failure path was tested in case a task was entered with an ID that already existed (Figure 6).



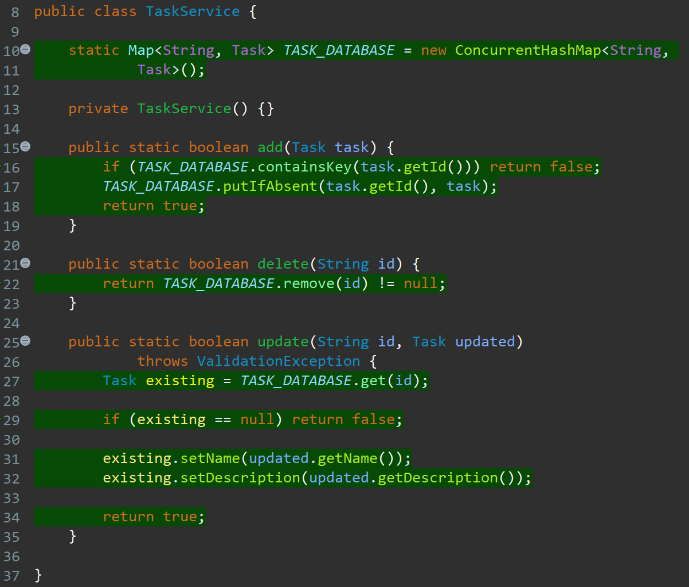
*Figure 5 Task Service Requirement Number One*



*Figure 6 Task Service test ensuring unique IDs are added*

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

100% code coverage for the TaskService class was provided as shown in Figure 7 and Figure 8:



*Figure 7 TaskService class with 100% code coverage*

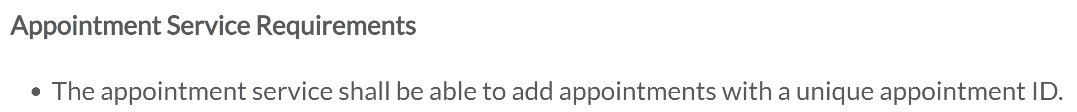


*Figure 8 TaskService class code coverage as a percentage*

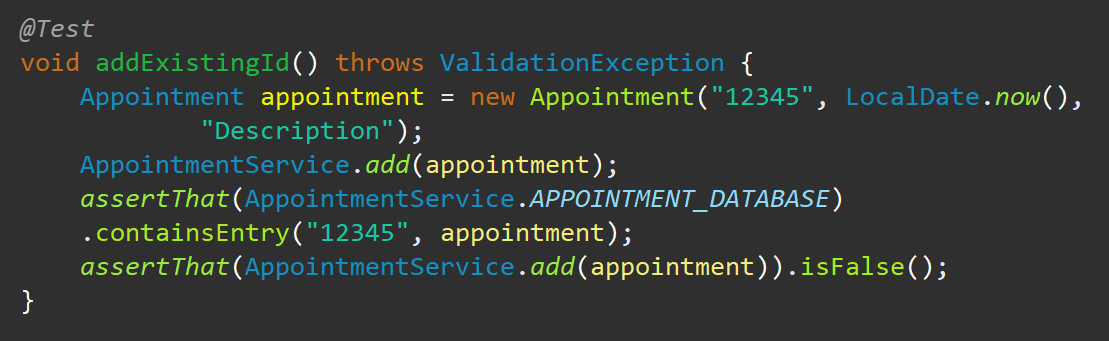
### Appointment Entity and Appointment Service.

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

100% percent of software requirements were tested to ensure quality. For example, in addition to the success path of each appointment having a unique ID as required in the Software Requirements (Figure 9), the failure path was tested in case an appointment was entered with an ID that already existed (Figure 10).



*Figure 9 Appointment Service Requirement Number One*

**

*Figure 10 Appointment Service test ensuring unique IDs are added*

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

100% code coverage for the AppointmentService class was provided as shown in Figure 11 and Figure 12:



*Figure 11 AppointmentService class with 100% code coverage*

**

*Figure 12 AppointmentService class code coverage as a percentage*

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

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

I ensured that my code was technically sound, from a testing standpoint, by verifying that every validation requested in the software requirements was performed in the validation() methods of the Contact, Task, and Appointment classes, lines 26, 22, and 24 respectively. I also made sure that every execution path was tested in the ContactService, TaskService, and AppointmentService classes by using the coverage tools in Eclipse, as shown above in Figures 3, 4, 7, 8, 11, and 12, respectively.

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

I ensured that my code was efficient by using Maps instead of arrays, Lists, Trees, or other less-efficient data structures in the ContactService, TaskService, and AppointmentService classes. I also used a Validation utility class since the same code could be used to validate each entity. Lastly, I parameterized all tests, where possible, to ensure that only the data was duplicated (e.g.: Line 34 in the AppointmentTest class).

# 2. Reflection

## 2a. Testing Techniques

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

The specific techniques which I employed in this project were equivalence partitioning, boundary value analysis, use case testing, statement testing and coverage, and decision testing and coverage.

Equivalence partitioning was used when validating the length of the fields in the Contact, Task, and Appointment classes. When validating the length of those fields, it was assumed that one character above the length limit was equivalent to any number of characters above the length limit, so only one test was needed. Also, since only one test was needed, that test was done at a boundary value of one character above the length limit since most errors tend to happen at the boundary values (Samaroo et al., 2019).

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

Techniques that were not needed in this project, and which I did not use, were decision table testing and state transition testing.

Decision table testing is used when one or more methods or fields are dependent upon other methods or fields based upon the business rules given in the requirements. Since all objects and methods operate independently in this project, decision table testing was not needed.

State transition testing is needed when an application changes states or modes thus affecting system behavior (Samaroo et al., 2019). Since the scope of this project only relies on a single state of the application, state transition testing was unnecessary.

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

Equivalence partitioning and boundary value analysis almost always go hand-in-hand and apply to most projects since most projects will need to store and validate data input in one way or another (Samaroo et al., 2019).

Decision table testing applies to projects where the data or execution on one part of the application depends on the data or execution of another part of the application (Samaroo et al., 2019).

State transition testing applies to projects where the application has different states or modes that affect system behavior and those transitions need to be tested to ensure the system is in the correct state after a state or mode change (Samaroo et al., 2019).

Use case testing applies to almost every project since both data and methods will need to be tested for both success and failure use cases (Samaroo et al., 2019).

Statement and decision testing and coverage will apply to most projects since almost all projects will have fields and methods which will need their output tested and compared to expected output (statement testing) and will also have conditional statements which will need all execution paths tested (decision tested). All projects will need to be tested to a minimum of eighty percent coverage (Samaroo et al., 2019).

## 2b. Mindset

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

In acting as a software tester, specifically a beginner software tester, I employed the *maximum* possible caution in this project. I made sure that *every* class had 100% code coverage, as demonstrated above in Figures 3, 4, 7, 8, 11, and 12.

It was important to understand the complexity of the code, specifically when it came to ensuring code efficiency, so that parameterized testing could be employed to simplify the code while maintaining 100% code coverage.

The interrelationships of the Contact, Task, and Appointment classes were important to understand as that understanding allowed me to use a Validation utility class to deduplicate related code in those classes.

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

When testing code, it’s important to limit bias since there is a natural, healthy tendency to take pride in your work, and trying to break something you’ve built can seem counter-intuitive.

One way to avoid bias is to constantly refer back to the software specifications when designing your tests to ensure that you’re testing every requirement. Also, using a third-party tool such as the code coverage tools found in many IDEs can help ensure that you aren’t forgetting any code branches in your testing that you might have “forgotten” due to bias.

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

It’s important to not cut corners when it comes to testing in order to avoid technical debt in your projects. Regardless of how critical your code may be, it will always cost more to fix it than it will to do it right the first time (Samaroo et al., 2019).

One way I intend to avoid technical debt as a software engineer is to employ Test Driven Development as often as allowed so that the most possible amount of my code has been tested before delivery.

**References**

Samaroo, A., Thompson, G. G., & Hambling, B. (2019). *Software Testing: An ISTQB-BCS Certified Tester Foundation Guide 4th Ed* (4th ed.). BCS Learning & Development Ltd.