

CS320 Summary and Reflections Report

Eric Slutz

Southern New Hampshire University

Table of Contents

Table of Contents	2
1. Summary	3
1a. Describe your unit testing approach for each of the three features.	3
1b. Describe your experience writing the JUnit tests.	5
2. Reflection	6
2a. Testing Techniques	6
2b. Mindset	7







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 Entity gave the specific requirements, specifying the fields and the requirements for each field such as field type, length, and nullability. The Contact Service also gave the specific requirements for the expected functionality of each service such as the ability to add, delete, or update an object. With these requirements it was easy to create tests for each requirement to ensure everything worked as expected. For example, the firstName field is specified as it should not be null. This can be validated through testing by passing a null value for name and seeing if the issue is caught when attempting to set the firstName value.

Defend the quality of your JUnit tests.

For the Contact Entity and Contact Service, all the specified requirements were validated by creating tests to check each requirement, including checking against edge cases. The code coverage shows that between the Contact Entity and Contact Service there was 95.1% code coverage.

Console Problems Debug Shell Coverage X				
AllContactTests (Jan 25, 2023 4:59:13 PM)				
Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
Module 3 Milestone	 79.3 %	760	198	958
src	 79.3 %	760	198	958
test	 72.0 %	470	183	653
contact	 95.1 %	290	15	305
Contact.java	 89.9 %	133	15	148
ContactService.java	 100.0 %	157	0	157







Task Entity and Task Service.

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

The Task Entity gave the specific requirements, specifying the fields and the requirements for each field such as field type, length, and nullability. The Task Service also gave the specific requirements for the expected functionality of each service such as the ability to add, delete, or update an object. With these requirements it was easy to create tests for each requirement to ensure everything worked as expected. For example, the name field is specified as it should not be null. This can be validated through testing by passing a null value for name and seeing if the issue is caught when attempting to set the name value.

Defend the quality of your JUnit tests.

For the Task Entity and Task Service, all the specified requirements were validated by creating tests to check each requirement, including checking against edge cases. The code coverage shows that between the Task Entity and Task Service there was 100% code coverage.

Console Problems Debug Shell Coverage X				
AllTaskTests (Jan 25, 2023 4:57:46 PM)				
Element	Coverage	Covered Instructions	Missed Instructions: ▾	Total Instructions
Module 4 Milestone	 89.5 %	529	62	591
src	 89.5 %	529	62	591
test	 84.4 %	336	62	398
task	 100.0 %	193	0	193
Task.java	 100.0 %	86	0	86
TaskService.java	 100.0 %	107	0	107

Appointment Entity and Appointment Service.

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

The Appointment Entity gave the specific requirements, specifying the fields and the requirements for each field such as field type, length, and nullability. The Appointment Service also gave the specific requirements for the expected functionality of each service such as the ability to add, delete, or update an object. With these requirements it was easy to create tests for

each requirement to ensure everything worked as expected. For example, the description field is specified as it should not be null. This can be validated through testing by passing a null value for description and seeing if the issue is caught when attempting to set the name value.

Defend the quality of your JUnit tests.

For the Appointment Entity and Appointment Service, all the specified requirements were validated by creating tests to check each requirement, including checking against edge cases. The code coverage shows that between the Appointment Entity and Appointment Service there was 100% code coverage.

test (1) (Feb 16, 2023 6:23:58 AM)					
Element	Coverage	Covered Instructions	Missed Instruction: ▾	Total Instructions	
Module 5 Milestone	87.2 %	421	62	483	
src	87.2 %	421	62	483	
test	81.8 %	279	62	341	
appointment	100.0 %	142	0	142	
Appointment.java	100.0 %	85	0	85	
AppointmentService.java	100.0 %	57	0	57	

1b. Describe your experience writing the JUnit tests.

How did you ensure that your code was technically sound?

In order to create technically sound code, I created test cases for a wide variety of input for the different each method within the classes and services. By trying to test for edge cases it helps to make sure the code and handle whatever could possibly be thrown at it. For example, in this validation code for the Contact class constructor you can see how it checks for all requirements.

```
if (contactId == null || contactId.isBlank()) {
    throw new IllegalArgumentException("contactId must have a value.");
} else if (contactId.length() > 10) {
    throw new IllegalArgumentException("contactId cannot be longer than 10 characters.");
}
```

```
} else {  
    this.contactId = contactId;  
}
```

How did you ensure that your test code was efficient?

In terms of writing efficient code, I worked to keep the classes and services simple. I also utilized existing methods when available for creating functionality. For example, in the Task Service, I used a HashMap for storing the tasks. This allowed me to add or delete a task in a single line each.

```
return this.tasks.putIfAbsent(newTask.getTaskId(), newTask) == null;  
return this.tasks.remove(id) != null;
```

2. Reflection

2a. Testing Techniques

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

When creating tests for this project, the black-box test techniques were employed. Specifically equivalence partitioning, boundary value analysis, decision table testing, and use case testing were used. Part of equivalence partitioning works by grouping input and testing a wide range of input that should be valid. Boundary value analysis on the other hand can be used to test input around the lower and upper end of the valid input range. This should also include testing that falls outside of the valid input range. Decision table testing includes creating a table with some business rules and the different input conditions to write tests that can be used to validate any business rules.

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

The black-box technique of state transition testing was not used for the project. This technique is used to test different outputs that are triggered by a combination of inputs. White-box testing, and experience-based techniques were also not used. White-box testing is used to test the code's internal structure and design. Experience-based testing is based on the user's experience with error guessing, exploratory testing, and checklist-based testing.

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

The testing techniques that were employed allow for thorough testing of input to ensure that any unexpected input is tested for and that only valid input is allowed to pass. The implication of equivalence partitioning is that any valid input should work correctly with the program. The implication of boundary value analysis is that any edge cases do not cause any issues with the program. The implication of decision tree testing is that all business requirements have been met. For example, in the appointment class milestone, there was a requirement that the appointment date could not be in the past, which is a business requirement. This testing technique can be used to ensure that the requirement is being met.

2b. Mindset**Assess the mindset you adopted working on this project.**

When developing tests for this project it is important to be cautious and keep in mind the complexity and interrelationships of the code being tested. For example, if you are not aware of the relationships between parts of the program, it could be possible to miss a branch of code for testing. It could even result in missing an entire test case for the program.

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

When reviewing code, it is important to attempt to limit any bias. Having written the code, you know what the expected input is and the expected output. This can make it difficult to think of the unexpected input that may be entered when a user with not prior experience attempts to use the program. This can then result in failures because of missed opportunities in testing.

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

It is important to not cut corners when writing or testing code. You never know what unintended results there could be from your code failing. For example, by cutting corners and not thoroughly testing the code to get the program finished faster, it increases the likelihood of users running into problems when running the program or causing some sort of catastrophic error. Technical debt can be avoided by adding or updating tests as you make changes to the program.