CS320 Summary and Reflections Report

John L. Miller

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. 4](#_Toc121853843)

[2. Reflection 4](#_Toc121853844)

[2a. Testing Techniques 4](#_Toc121853845)

[2b. Mindset 5](#_Toc121853846)

[References 6](#_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 entity and service classes tests were designed with the software requirements in mind. The contact entity required a unique ID, first name, last name, telephone, and address fields. Each field has character requirements and must be non-null. These requirements guided me in designing tests that would validate the entity if all requirements were met or test to see if they were correctly thrown. For example, testing to see if a contact object with a null ID would be correctly thrown.

The contact service class requirements were instrumental in designing tests. This led me to develop test cases to ensure the functions performed precisely as expected—for example, the code required deleting contacts using the ID as an identifier.

public void deleteById\_RemovesContact(){

This test case was used to add, remove, and confirm the deletion of a contact. While I did have an intimate knowledge of the code's structure, the test cases were written without considering it. For example, I knew the program needed to meet the delete function and designed my test case to ensure this occurred.

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

The test coverage for my contact entity and service class is 100%. Every option in the service class is successfully tested. The tests ensure that the requirements for the contact service class are fully met and functional, such as ensuring that a contact can be updated by ID.

*assertEquals*(updatedC, service.findById(id)); // Asserts updatedC has the same data as the object now in service.

*assertEquals*(updatedC.firstName(), service.findById(id).firstName()); // Asserts that updatedC has the same first name as the updated object.

*assertEquals*(updatedC.lastName(), service.findById(id).lastName()); // Asserts that updatedC has the same last name as the updated object.

*assertEquals*(updatedC.contactPhone(), service.findById(id).contactPhone()); // Asserts that updatedC has the same phone as the updated object

*assertEquals*(updatedC.contactAddress(), service.findById(id).contactAddress()); // Asserts that updatedC has the same address as the updated object.

The test uses assertEquals to verify a contact created and then, using the updateByID function, replaces an object with the matching ID in the service class. Reaching 100% coverage in the contact entity involved using @ParameterizedTest to feed an input stream to a test case, ensuring each invalid contact was correctly thrown with an appropriate error message. This method allowed me to quickly test a variety of contact fields without using repeated code. Ensuring my code met the requirements and using the tools provided allowed me to reach 100% coverage and ensure that the tests were acceptable.

### Task Entity and Task Service.

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

The task service and entity requirements were integral in designing the test cases for these classes. The task entity had specific field requirements, which guided me in developing test cases to ensure these requirements were met. For example, the task ID field is required to be non-null.

Arguments.*of*(null, "TestCase", "checking if ID is Null", "taskID is a required field"), // NULL ID

The task service class has specific functional requirements that were instrumental in designing my test cases. One example is adding a task with a unique ID. This was tested by creating a task, adding it to a service, and then attempting to add a task with a duplicate ID to verify an error is thrown in this instance.

*assertThrows*(Exception.class, () -> service.create(new Task (id, "TestCase", "Testing prevent creation")));

The assertThrows statement returns a Boolean showing true(passed) or false(failed) if the code does or does not throw an exception, respectively.

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

The test coverage for the task service and entity classes is 100%. I was sure to test every function of the service and parameter of the entity. I used a pattern similar to the contact entity in the task entity test. ParameterizedTest uses MethodSource to feed arguments and test the constraints of the task entity.

// Test Case for invalid Tasks.

*@ParameterizedTest*

*@MethodSource*("taskInfoProvider")

public void testTasks(String taskID, String taskName, String taskDescription, String expectedMessage) {

The argument stream successfully tests each constraint, proven by the 100% test coverage. The task service's 100% test coverage ensures that each option of every function is executed. The assert statements within the test cases validate the function of the executed lines. For example, the task service required the function to update the task by ID. The following assert statements ensure this action is accomplished and the updated task has the expected values.

*assertEquals*(updatedT, service.findByID(id).get());

*assertEquals*(updatedT.taskName(), service.findByID(id).get().taskName());

*assertEquals*(updatedT.taskDescription(), service.findByID(id).get().taskDescription());

### Appointment Entity and Appointment Service.

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

The appointment entity and service test cases follow a similar pattern to the contact and task objects. The appointment entity's requirements guided me in the test cases. I wanted to ensure each field constraint was appropriately accepted or thrown. This was accomplished again with a Parameterized test and a stream of arguments for both valid and invalid appointment objects.

static Stream<Arguments> InvalidAppointmentProvider() {

static Stream<Arguments> ValidAppointmentProvider() {

The appointment service class required that appointment be deleted by ID. The following assert statements ensure an object named deleted is created, matches the object created that was to be deleted, and finally that, the id of these objects is no longer present in the service.

*assertTrue*(deleted.isPresent());

*assertEquals*(a, deleted.get());

*assertThrows*(IllegalArgumentException.class, () -> service.findByID(id));

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

The test coverage for the appointment service and entity was 100%. As with the others, this ensures that every line and option of code was executed within my test cases. This indicates that the code is all reachable. The assert statements within my tests ensure that the code acts as expected. For example, the appointment entity requires that the appointment date be today or later. This was successfully tested in the AppointmentTest ValidAppointmetProvider. The stream included an appointment for today and in the future, validating that either is acceptable.

Arguments.*of*("1234567890", ***validDate***, "Testing future date"), // Test future date is correctly accepted

Arguments.*of*("1234567890", ***todaysDate***, "Testing today's date")); // Test today's date is correctly accepted

The appointment service also required each appointment to have a unique ID. To verify this, an appointment was created and added to the service. The code then attempts to add an appointment with a matching ID to verify that an exception has been thrown in this instance.

*assertThrows*(IllegalArgumentException.class, () ->

service.create(new Appointment(id, date, "Testing Prevent Creation")));

Understanding the requirements and how the code is executed to meet these requirements allows me to be confident in the test cases.

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

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

I ensured my code was technically sound by validating each function and constraint. This was done by understanding the coding patterns and requirements used to develop the code. For example, the appointment service required the ability to delete appointments by ID. To accomplish this, the service must have a way to identify appointments by ID. Thus, although not explicitly required, the service class needed a way to return appointment objects by their ID, and a find by ID function was included. This function, for coverage’s sake, must be tested as well. I developed test cases that verified an object is returned when a matching ID exists and an error is thrown when a matching ID is not found.

public void findByID\_ReturnsMatchingIDObject() {

public void findByID\_NomatchingID() {

Being thorough and considering every option in the service class allowed me to develop test cases like these and achieve 100% code coverage. The code coverage also gave me confidence that the code was technically sound, as every option was executed at least once. I also ensured a new service was created for each test case.

*@BeforeEach*

void init() {

appointmentService.*INSTANCE* = null;

service = appointmentService.*getInstance*();

}

This pattern ensures a “clean slate” before each test; therefore, there can be no residual side effects from the completed test.

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

# I ensured my test code was efficient by using best coding practices when developing my test cases. I eliminated much redundant code by utilizing parameterized tests in the entity classes. For example, the contact entity had many fields (ID, first name, last name, phone, and address), each with unique constraints. Instead of writing a test case for each combination, I used a parameterized test that leveraged the source of the method. This allowed me to have a stream of arguments that were tested individually. This way, I could test each constraint separately and efficiently.

*@MethodSource*("contactInfoProvider")

public void testContacts(String id, String firstName, String lastName, String contactPhone, String contactAddress, String expectedMessage) {

The fields of the function testContacts were assigned, and then I had test cases to ensure each argument stream threw the expected error message.

final ContactInfo testCase = new ContactInfo(id, firstName, lastName, contactPhone, contactAddress);

ValidationException exception = *assertThrows*(ValidationException.class, () ->

service.create(testCase));

String actualMessage = exception.getMessage();

*assertTrue*(actualMessage.contains(expectedMessage));

The assert statements verify that an error is thrown and that the error message contains the expected message. I also made sure each test case only tested one function. As established, the appointment service required the functionality to delete appointments by ID, and a separate function to identify each appointment by ID was needed to accomplish this. Although these functions work in unison, they must be tested separately to verify their functionality. Thus, I included separate tests to confirm the search and delete functions.

public void findByID\_ReturnsMatchingIDObject() {

public void delete\_ByID() {

Testing anti-patterns occurs when a test case tests multiple functionality at a time. Designing many separate test cases testing each option prevented this.

# 2. Reflection

## 2a. Testing Techniques

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

I employed various testing techniques when designing my test code. One example is equivalence partitioning. This involves “chunking” data into groups of similar inputs or expected outputs. This is demonstrated in the ParameterizedTest of the entity classes—these “chunk” inputs into either valid or invalid arguments for the various entities (Morgan et al., 2019).  
static Stream<Arguments> InvalidAppointmentProvider() {

static Stream<Arguments> ValidAppointmentProvider() {

The above is an example of gathered arguments that should be thrown or accepted. The parameterized test also uses boundary value testing, or testing values near acceptable boundaries and confirming they are appropriately accepted or denied (Morgan et al., 2019). This is evident in my test cases for the contact entity. The constraints dictate that the ID field cannot be null or longer than ten characters.

Arguments.*of*("01234567890", "John", "Miller", "1234567890", "123 Fake St.", "contactID cannot be longer than 10 characters"), // id is too long

The argument provides an ID of 11 characters, all other fields as valid, and the expected error message. State Transitioning testing is checking to ensure combinations of input conditions produce various actions or state changes (Morgan et al., 2019). An example of this in my test code is ensuring an error is thrown when attempting to add a contact to the service when the ID already exists.

*assertThrows*(IllegalArgumentException.class, () -> service.create(new ContactInfo(id,"John", "Miller", "0123456789", "123 Fake St.")));

This ensures a state change occurs in the form of an error thrown.

As discussed, the test code achieves 100% code coverage. This was accomplished by implementing statement testing and decision testing. Statement testing exercises executable statements in the code. An example of statement testing is found in the task entity test (Morgan et al., 2019). The task entity had constraints, such as the fact that the ID could not be null. I developed my test case to verify this specific constraint.

Arguments.*of*(null, "TestCase", "checking if ID is Null", "taskID is a required field"), // NULL ID

Decision testing exercises the decisions in the code (Morgan et al., 2019). An example of decision testing can be found in the contact services tests. The contact service required contacts be updatable by ID. The code dictates a contact with a matching ID replaces the old one or throws an error if no match is found. Thus, both instances are tested in the test code.

public void update\_PreventsUpdateWithoutMatchingID() {

public void update\_UpdatesMutableFieldsOfContactInfo(){

### 

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

There are several testing techniques I did not use for this project. One is use-case testing. Use-case testing captures the interaction between “actors” and the system (Morgan et al., 2019). This would have been impractical for this project as no user interface was designed for this program, and the code was only executed through my test cases. Thus, no user can interact with this system, making use-case testing impractical. Decision table testing involves using tables to list all input conditions and actions that can arise from them (Morgan et al., 2019). This may have been practical for designing the parameterized tests. However, I did not use this technique as the parameterized tests were either thrown or accepted, making the implementation of the parameterized test straightforward and the need for a decision table redundant.

### 

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

Equivalence Partitioning allows developers to test various sets of inputs efficiently. By chunking like actions/inputs together, the tester may see if they lead to similar results.

Boundary Value Testing—Testing near boundaries is hugely beneficial. Suppose the tests accept bounds that are near but within limits and do not accept values near but outside the bounds. In that case, the tester may assume all acceptable values within bounds are correctly accepted and values outside of the bounds are correctly rejected. This is critical in programs where precise input ranges are crucial.

State Transition Testing - This is useful in systems where the output depends on the sequence of previous inputs, not just the current ones. For example, in a user authentication system, state transition testing would ensure that after several failed login attempts, the account state transitions to ‘locked.’

Use-Case Testing—This is practical for projects with user interfaces or where user interaction is a significant component. It covers real-world usage scenarios to help ensure the system behaves as expected from the user’s perspective.

Decision Table Testing—This is useful in projects with complex business logic where multiple conditions and their combinations must be tested. It provides a systematic way to capture all possible combinations and their expected outcomes.

Statement and decision Testing are useful in scenarios where code coverage is required for the testing process. This style of testing occurs when a tester has intimate knowledge of the code structure and wishes to execute specific statements or decisions within their test cases.

## 2b. Mindset

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

It is crucial to employ thoroughness and caution when acting as a software tester. The code developed was highly interrelated. The AppointmentService class could not function if the EntityValidator failed. The service class depended on the validator to confirm the object before allowing it to be added. If the validator failed, the service class would also fail. Also, the constraints on the appointment class were only meaningful if added to the service. That is when the program would compare the inputs to the constraints and either accept or deny them. Each function must be tested individually and in unison, or a defect may escape the testing process.

### 

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

A developer must be vigilant in limiting their bias when testing the code they have developed. It is not uncommon for people to have a blind spot to their errors. To combat this, I attempted to build my test cases solely based on the requirements. For example, in the ContactService class, I knew intimately how the delete function was implemented. However, I designed my test case to ensure the delete function accomplished the requirement that a contact may be deleted by ID. This attempt at objectiveness adds to the test’s validity.

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

Being disciplined in my commitment to quality as a software professional is crucial. Cutting corners in testing can lead to vast technical debt. To avoid this, one should develop code and test modularly. Unit tests such as these ensure the current modules being developed are functioning. The mods should also undergo integration testing as they are completed. For example, in this project, achieving 100% test coverage and ensuring each test case was efficient and effective demonstrated a commitment to quality. Each test was developed as the entity and service class was complete. This way, I can be confident that they will behave as expected as I integrate the various classes into a package. Regression testing is also critical to ensure any previously passing test was not affected by the changes made in the program (Morgan et al., 2019).

References

Morgan, Peter, et al. *Software Testing : An ISTQB-BCS Certified Tester Foundation Guide - 4th Edition*, edited by Brian Hambling, BCS Learning & Development Limited, 2019.*ProQuest Ebook Central*, https://ebookcentral-proquest-com.ezproxy.snhu.edu/lib/snhu-ebooks/detail.action?docID=5837074.