

CS 320 Summary and Reflections Report

Kennedy Uzoho

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 Junit tests.....	5
2. Reflection.....	6
2a. Testing Techniques.....	6
2b. Mindset.....	7

Summary

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

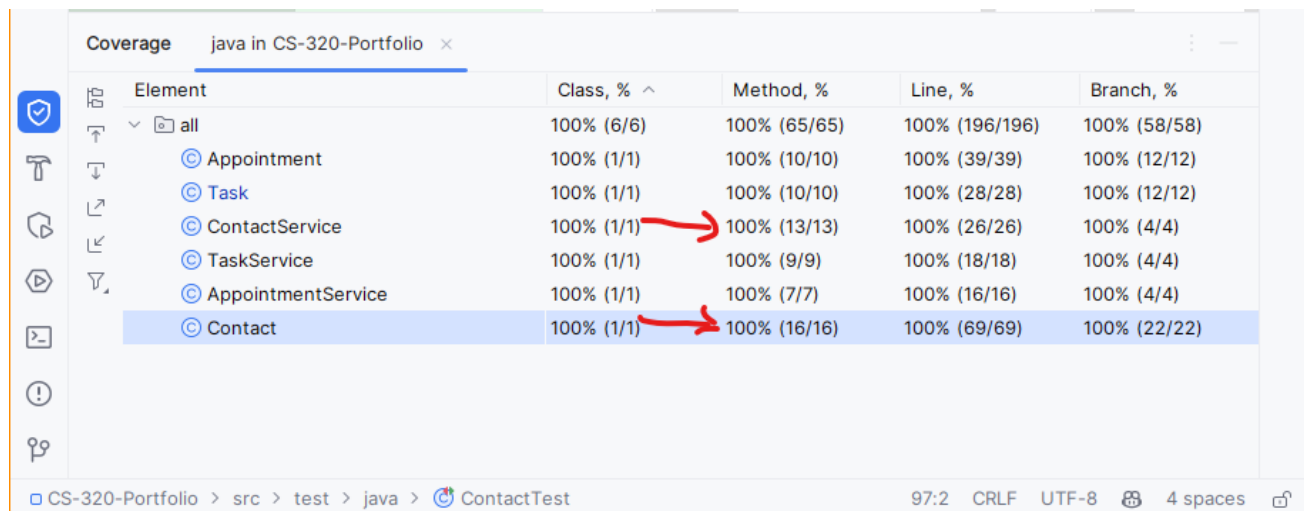
Contact Entity and Contact Service.

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

The `ContactTest` and `ContactServiceTest` were designed to thoroughly evaluate the core functionalities of the `Contact` entity and the `ContactService` class, ensuring each behaves as expected under various conditions. The `Contact` entity has specific requirements for field types, lengths, and nullability, which were addressed in the tests. Similarly, the `ContactService` has defined expectations for its operations, such as adding, deleting, or updating objects. These clear requirements facilitated the creation of targeted tests to verify compliance. For instance, the `firstName` field must not be null; this requirement was tested by passing a null value to the name field and checking if the error is properly caught.

Defend the quality of your JUnit tests.

The JUnit tests encompass a diverse range of test methods designed to evaluate the functionality of constructors, functions, and methods within a class. I made sure that every aspect of the entities and services was rigorously tested. Each constructor was tested individually to confirm that it correctly initializes objects with valid data and properly handles edge cases, such as null values or excessively long input. The detailed test coverage indicates that all 16 methods of the `Contact` entity and all 13 methods of the `Contact` service were tested comprehensively, achieving 100% coverage.



The screenshot displays the Coverage tool in IntelliJ IDEA, showing test results for the project 'CS-320-Portfolio'. The table lists the following elements and their coverage:

Element	Class, %	Method, %	Line, %	Branch, %
all	100% (6/6)	100% (65/65)	100% (196/196)	100% (58/58)
Appointment	100% (1/1)	100% (10/10)	100% (39/39)	100% (12/12)
Task	100% (1/1)	100% (10/10)	100% (28/28)	100% (12/12)
ContactService	100% (1/1)	100% (13/13)	100% (26/26)	100% (4/4)
TaskService	100% (1/1)	100% (9/9)	100% (18/18)	100% (4/4)
AppointmentService	100% (1/1)	100% (7/7)	100% (16/16)	100% (4/4)
Contact	100% (1/1)	100% (16/16)	100% (69/69)	100% (22/22)

Red arrows in the original image point to the 'ContactService' and 'Contact' rows, highlighting their 100% method coverage.

At the bottom, the breadcrumb path is: CS-320-Portfolio > src > test > java > ContactTest. The status bar shows: 97:2 CRLF UTF-8 4 spaces.

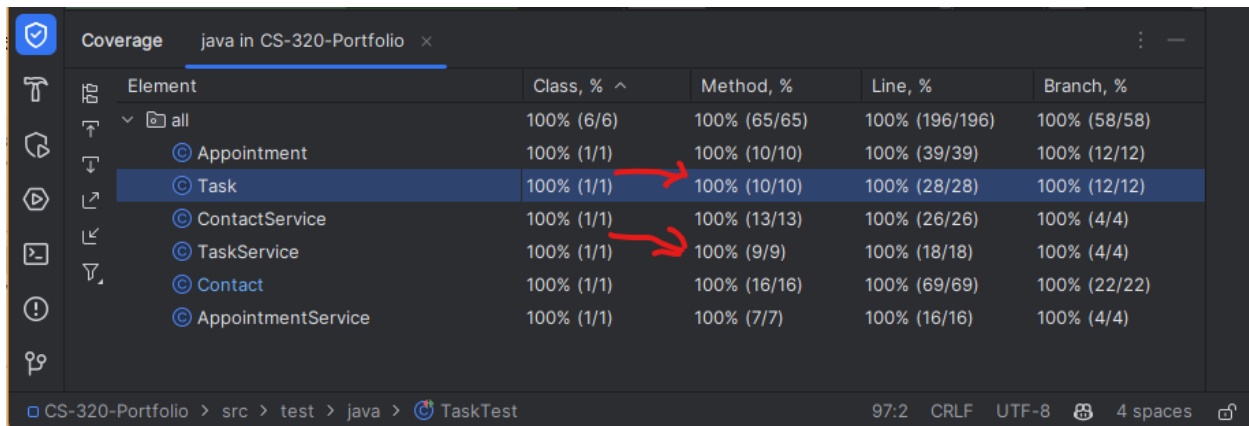
Task Entity and Task Service.

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

Like the `Contact` entity test, the `TaskTest` and `TaskServiceTest` were crafted to validate the core functionalities of the `Task` entity and `TaskService` class, ensuring that each behaves as expected in various scenarios. The `Task` entity and its service define specific requirements for field types, lengths, and nullity, which allowed for targeted testing of their functionalities, such as adding, deleting, or updating objects. For instance, the `taskId` field must not be null and should not exceed 10 characters. These constraints were tested by passing null values or exceeding lengths to verify that the system correctly handles exceptions and maintains robustness.

Defend the quality of your JUnit tests.

All methods of the `Task` entity and its service were rigorously tested. Each constructor was evaluated individually to ensure it correctly initializes objects with valid data and handles edge cases, such as null values or excessively long inputs. The comprehensive test coverage reveals that all 10 methods of the `Task` entity and all 9 methods of the `Task` service were covered 100% by the tests.



The screenshot shows the Coverage tool in IntelliJ IDEA for the project 'java in CS-320-Portfolio'. The table displays coverage metrics for various elements, with 'Task' and 'TaskService' highlighted in blue. Red arrows point to the 'Method, %' column for 'Task' and 'TaskService', both showing 100% coverage.

Element	Class, % ^	Method, %	Line, %	Branch, %
all	100% (6/6)	100% (65/65)	100% (196/196)	100% (58/58)
Appointment	100% (1/1)	100% (10/10)	100% (39/39)	100% (12/12)
Task	100% (1/1)	100% (10/10)	100% (28/28)	100% (12/12)
ContactService	100% (1/1)	100% (13/13)	100% (26/26)	100% (4/4)
TaskService	100% (1/1)	100% (9/9)	100% (18/18)	100% (4/4)
Contact	100% (1/1)	100% (16/16)	100% (69/69)	100% (22/22)
AppointmentService	100% (1/1)	100% (7/7)	100% (16/16)	100% (4/4)

CS-320-Portfolio > src > test > java > TaskTest 97:2 CRLF UTF-8 4 spaces

Appointment Entity and Appointment Service.

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

The `AppointmentTest` and `AppointmentServiceTest` were designed to validate the core functionalities of the `Appointment` entity and `AppointmentService` class, ensuring that they perform as expected. The `Appointment` entity specified requirements for its field lengths and validity through constructs, such as the `updateDate` construct was implemented to throw an `IllegalArgumentException` if the date is null or set in the past. Similarly, the

`AppointmentService` class has defined expectations for its functions, including the ability to add, delete, and update objects. By specifying requirements for fields and arguments, the entity creates opportunities for various types of tests, ensuring that all functionalities adhere to their intended requirements.

Defend the quality of your JUnit tests.

All methods of the `Appointment` entity and its service were thoroughly tested. Each constructor was evaluated individually to ensure that it correctly initializes objects with valid data and handles scenarios such as validation errors or excessively long inputs. The comprehensive test coverage demonstrates that 100% of the methods in the `Appointment` entity (10 methods) and the `Appointment` service (7 methods) were tested.



The screenshot shows the Coverage tool in IntelliJ IDEA for the project 'java in CS-320-Portfolio'. The table lists the following elements and their coverage:

Element	Class, % ^	Method, %	Line, %	Branch, %
all	100% (6/6)	100% (65/65)	100% (196/196)	100% (58/58)
Appointment	100% (1/1)	100% (10/10)	100% (39/39)	100% (12/12)
Task	100% (1/1)	100% (10/10)	100% (28/28)	100% (12/12)
ContactService	100% (1/1)	100% (13/13)	100% (26/26)	100% (4/4)
TaskService	100% (1/1)	100% (9/9)	100% (18/18)	100% (4/4)
AppointmentService	100% (1/1)	100% (7/7)	100% (16/16)	100% (4/4)
Contact	100% (1/1)	100% (16/16)	100% (69/69)	100% (22/22)

Red arrows point to the 'Appointment' and 'AppointmentService' rows, highlighting their 100% method coverage.

1b. Describe your experience writing the JUnit tests.

How did you ensure that your code was technically sound?

To ensure that code is robust and performs well under various conditions, it is essential to thoroughly test methods, functions, and services. This includes testing field validation and process handling across different scenarios. For instance, consider the `updateDate` method, which updates the `Appointment` date with a specified date.

```
public void updateDate(Date date)
{
    if (date == null)
    {
        throw new IllegalArgumentException("Appointment date cannot be null.");
    }
    else if (date.before(new Date()))
```

```

{
    throw new IllegalArgumentException(
        "Cannot make appointment in the past.");
}
else
{
    this.appointmentDate = date;
}

```

In this example, the `updateDate` method is used to safely change the `appointmentDate` field of the `Appointment` class, ensuring that the new date is neither `null` nor in the past.

The update method was tested using the JUnit test below.

```

@Test
void testUpdateDate()
{
    Appointment appt = new Appointment();
    assertThrows(IllegalArgumentException.class, () -> appt.updateDate(null));
    assertThrows(IllegalArgumentException.class,
        () -> appt.updateDate(pastDate));
    appt.updateDate(date);
    assertEquals(date, appt.getAppointmentDate());
}

```

The test creates a new `Appointment` object and verifies that passing a null date to `updateDate()` throws an `IllegalArgumentException`. It also asserts that passing a past date to `updateDate()` throws an `IllegalArgumentException`. Additionally, it updates the appointment date with a valid date and, in the end, verifies that the appointment date is updated correctly.

How did you ensure that your code was efficient?

At the unit level, code efficiency is achieved by defining attributes and fields as simply and clearly as possible. Methods, functions, and dynamic variables should be straightforward and concise to enhance readability for both humans and machines. By leveraging code portability and reusability across modules and classes, a modular approach fosters better separation of concerns. This not only simplifies maintenance but also enhances the potential for reusing methods, classes, and modules in various projects, whether on-premises or externally.

For instance, in the `Task` class, methods like `checkTaskId`, `setDescription`, and `setName` exemplify this principle. By centralizing validation logic in these methods, any necessary changes can be made in a single location, ensuring consistency, and minimizing the need

for updates across multiple constructors. This approach not only maintains the integrity of the code but also streamlines future modifications.

```
protected void setDescription(String taskDescription)
{
    if (taskDescription == null || taskDescription.length() > 50)
    {
        throw new IllegalArgumentException ( "Task description is invalid.
        Ensure it is shorter than 50 characters and not empty." );
    }
    else
    {
        this.description = taskDescription;
    }
}
// Method to get the task description
public final String getDescription() { return description; }
{
return description;
}

##### Test #####
void getDescriptionTest()
{
    Task task = new Task(id, name, description);
    Assertions.assertEquals(description, task.getDescription());
}
```

Reflection

2a. Testing Techniques

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

In software testing, a range of techniques is employed to verify that code performs as expected. Key techniques utilized in this project include Black-Box Testing, Unit Testing, Boundary Testing, Exception Testing, and Regression Testing.

- **Black-Box Testing** focused on evaluating the functionality of classes and methods from an external perspective, without delving into the internal workings of the code. For instance, various inputs were applied to methods such as `setDescription()` and `getDescription()` in the `Task` class to ensure that outputs or exceptions align with the expected results.

- **Unit Testing** was used to validate individual components or methods of the code to ensure each part functions correctly in isolation.
- **Boundary Testing** examined the edges of input ranges to ensure the code handles boundary conditions properly.
- **Exception Testing** was employed to verify that the code correctly handles and raises exceptions under erroneous conditions.
- **Regression Testing** ensured that new changes did not negatively impact existing functionality, maintaining the overall integrity of the software.

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

Most elements of White-Box testing techniques were not applied in this project. White-Box Testing focuses on evaluating the internal structure and logic of the code, requiring an understanding of the code's design and implementation. This approach enables more comprehensive testing by covering various code paths, conditions, and loops. While White-Box testing can provide in-depth insights into code functionality, it's worth noting that these techniques can be used individually or in combination with other testing methods, depending on the project's complexity and specific requirements.

For each of the techniques you discussed, explain the practical uses and implications for different software projects and situations.

All testing techniques used in this project were designed to ensure that outputs and exceptions align with expected results. Unit testing focuses on validating individual components or methods to confirm that specific sections of the code, such as functions or methods, operate correctly. These unit tests can be complemented by integration tests to verify that different parts of the application work seamlessly together. For instance, tests can be created to assess how the Task entity interacts with other classes, methods, or repository classes, ensuring that all components integrate and function as intended.

2b. Mindset

Assess the mindset that you adopted working on this project.

Upon completing various testing approaches for this project, the principles of modular design were effectively demonstrated through organized code and a clear focus on separation of concerns. Extensive regression testing underscored a commitment to continuous improvement. Challenges were systematically addressed, including resolving environmental issues, mastering Maven configuration, and optimizing code. The complexity and interrelationships of the code were carefully considered, which is crucial for ensuring comprehensive testing. Awareness of these relationships helps prevent oversight of critical code segments, reducing the risk of missing important test cases.

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

Limiting bias in code reviews is essential for maintaining objectivity and ensuring that assessments are grounded in the code's quality, functionality, and adherence to best practices, rather than personal preferences or assumptions. In this project, JUnit automated tests played a critical role in validating the code's functionality, offering objective feedback on whether the code performs as intended. Automated tests help mitigate personal bias by delivering clear, binary results (pass/fail) based on predefined criteria, thus ensuring a fair and consistent evaluation of the code.

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

Maintaining discipline in your commitment to quality as a software engineering professional is essential for several reasons. It ensures the reliability and stability of the software, meets, or exceeds user expectations, and facilitates easier maintenance and scalability. Upholding

high standards in code quality demonstrates a deep respect for the craft and a strong dedication to delivering lasting value to clients, employers, and end-users.