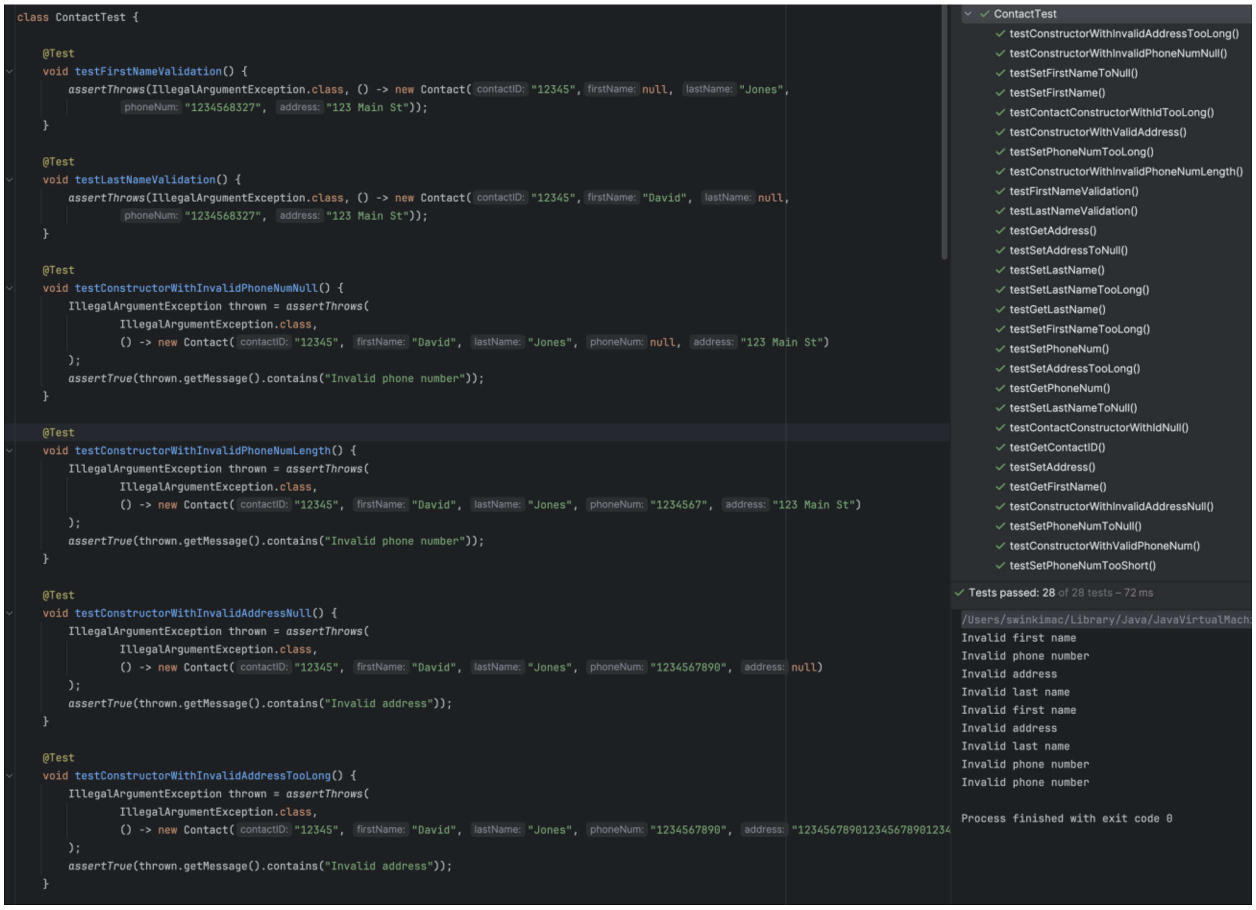
Summary & Reflections Report

Derricko Swink

Summary

For the ***Contact*** feature, I focused on testing the creation, update, and deletion functionalities. My tests ensured that a contact object could not be created with invalid inputs, such as null values or improperly formatted phone numbers. I validated that each contact had a unique ID and all fields were required for successful creation.

Contact Test Examples:



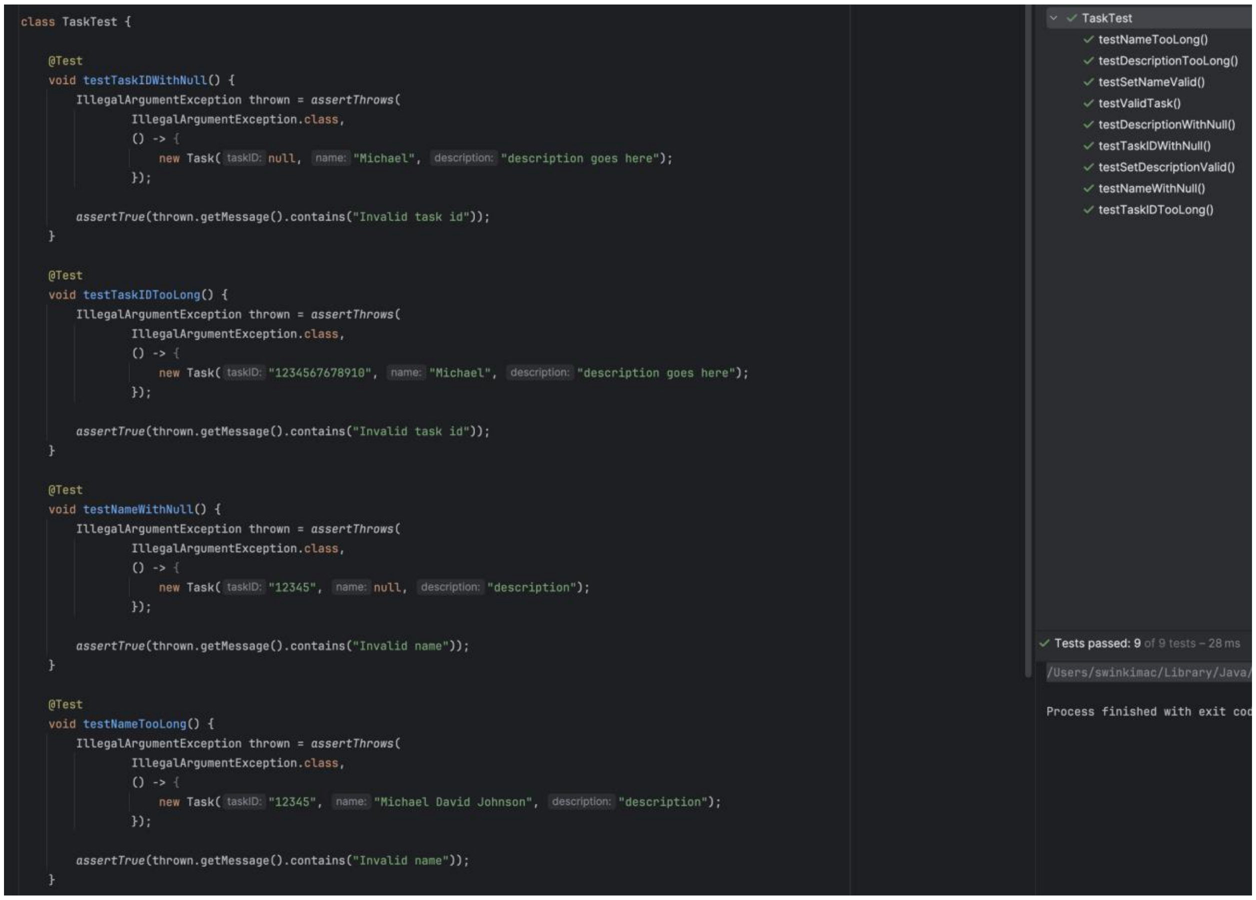
```
class ContactTest {

    @Test
    void testFirstNameValidation() {
        assertThrows(IllegalArgumentException.class, () -> new Contact( contactID: "12345", firstName: null,  lastName: "Jones",
                phoneNum: "1234568327",  address: "123 Main St"));
    }

    @Test
    void testLastNameValidation() {
        assertThrows(IllegalArgumentException.class, () -> new Contact( contactID: "12345", firstName: "David",  lastName: null,
                phoneNum: "1234568327",  address: "123 Main St"));
    }

    @Test
    void testConstructorWithInvalidPhoneNumNull() {
        IllegalArgumentException thrown = assertThrows(
                IllegalArgumentException.class,
                () -> new Contact( contactID: "12345",  firstName: "David",  lastName: "Jones",  phoneNum: null,  address: "123 Main St")
        );
        assertTrue(thrown.getMessage().contains("Invalid phone number"));
    }

    @Test
    void testConstructorWithInvalidPhoneNumLength() {
        IllegalArgumentException thrown = assertThrows(
                IllegalArgumentException.class,
                () -> new Contact( contactID: "12345",  firstName: "David",  lastName: "Jones",  phoneNum: "1234567",  address: "123 Main St")
        );
        assertTrue(thrown.getMessage().contains("Invalid phone number"));
    }

    @Test
    void testConstructorWithInvalidAddressNull() {
        IllegalArgumentException thrown = assertThrows(
                IllegalArgumentException.class,
                () -> new Contact( contactID: "12345",  firstName: "David",  lastName: "Jones",  phoneNum: "1234567890",  address: null)
        );
        assertTrue(thrown.getMessage().contains("Invalid address"));
    }

    @Test
    void testConstructorWithInvalidAddressTooLong() {
        IllegalArgumentException thrown = assertThrows(
                IllegalArgumentException.class,
                () -> new Contact( contactID: "12345",  firstName: "David",  lastName: "Jones",  phoneNum: "1234567890",  address: "123456789012345678901234
        );
        assertTrue(thrown.getMessage().contains("Invalid address"));
    }
```

```
✓ ContactTest
    ✓ testConstructorWithInvalidAddressTooLong()
    ✓ testConstructorWithInvalidPhoneNumNull()
    ✓ testSetFirstNameToNull()
    ✓ testSetFirstName()
    ✓ testContactConstructorWithIdTooLong()
    ✓ testConstructorWithValidAddress()
    ✓ testSetPhoneNumTooLong()
    ✓ testConstructorWithInvalidPhoneNumLength()
    ✓ testFirstNameValidation()
    ✓ testLastNameValidation()
    ✓ testGetAddress()
    ✓ testSetAddressToNull()
    ✓ testSetLastName()
    ✓ testSetLastNameTooLong()
    ✓ testGetLastName()
    ✓ testSetFirstNameTooLong()
    ✓ testSetPhoneNum()
    ✓ testSetAddressTooLong()
    ✓ testGetPhoneNum()
    ✓ testSetLastNameToNull()
    ✓ testContactConstructorWithIdNull()
    ✓ testGetContactID()
    ✓ testSetAddress()
    ✓ testGetFirstName()
    ✓ testConstructorWithInvalidAddressNull()
    ✓ testSetPhoneNumToNull()
    ✓ testConstructorWithValidPhoneNum()
    ✓ testSetPhoneNumTooShort()
✓ Tests passed: 28 of 28 tests – 72 ms
```

```
/Users/swinkimac/Library/Java/JavaVirtualMach
Invalid first name
Invalid phone number
Invalid address
Invalid last name
Invalid first name
Invalid address
Invalid last name
Invalid phone number
Invalid phone number

Process finished with exit code 0
```

For the ***Task*** feature, I ensured that task objects adhered to constraints like deadlines and unique task IDs. I tested scenarios where invalid inputs, such as past due dates or empty task names, should throw exceptions.

Task Test Examples:

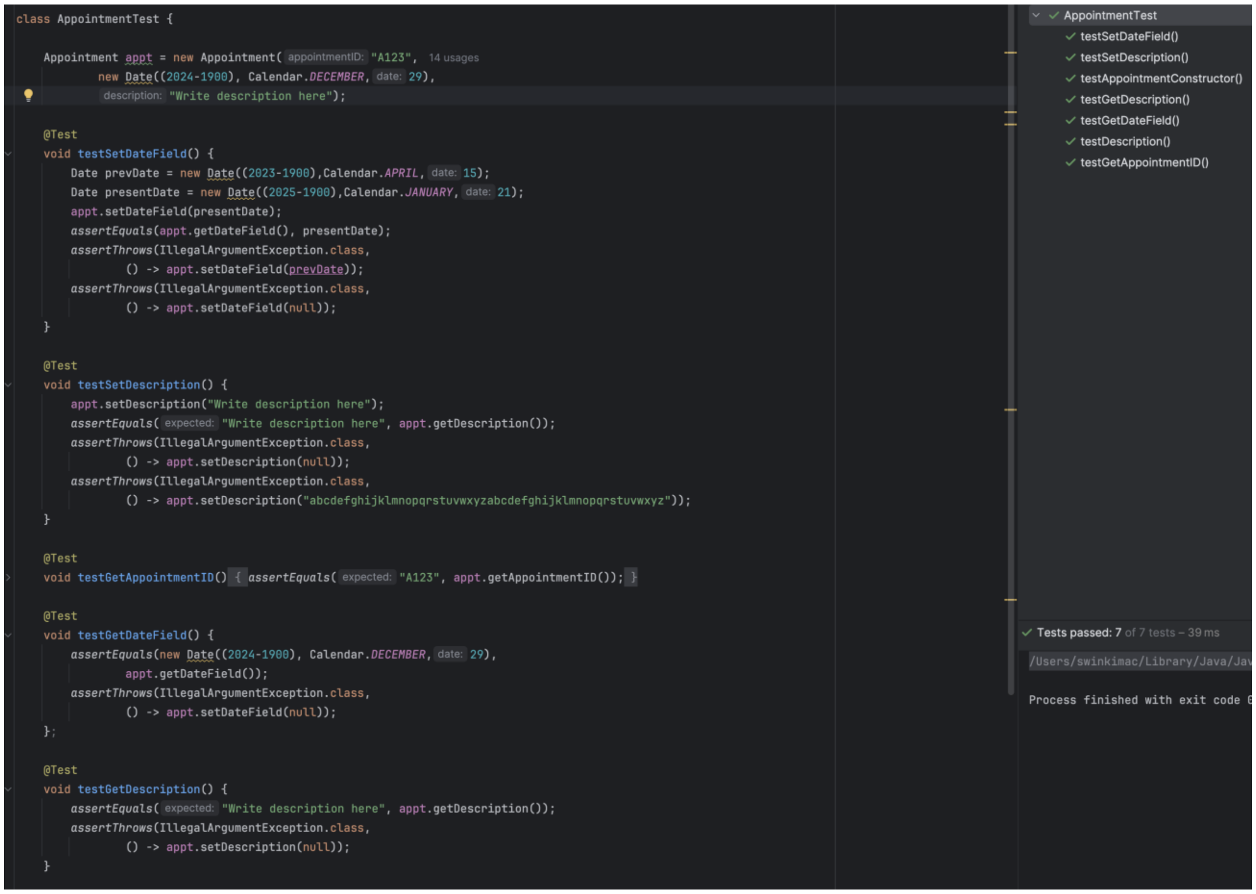


```java
class TaskTest {

    @Test
    void testTaskIDWithNull() {
        IllegalArgumentException thrown = assertThrows(
                IllegalArgumentException.class,
                () -> {
                    new Task( taskID: null,  name: "Michael",  description: "description goes here");
                });

        assertTrue(thrown.getMessage().contains("Invalid task id"));
    }

    @Test
    void testTaskIDTooLong() {
        IllegalArgumentException thrown = assertThrows(
                IllegalArgumentException.class,
                () -> {
                    new Task( taskID: "1234567678910",  name: "Michael",  description: "description goes here");
                });

        assertTrue(thrown.getMessage().contains("Invalid task id"));
    }

    @Test
    void testNameWithNull() {
        IllegalArgumentException thrown = assertThrows(
                IllegalArgumentException.class,
                () -> {
                    new Task( taskID: "12345",  name: null,  description: "description");
                });

        assertTrue(thrown.getMessage().contains("Invalid name"));
    }

    @Test
    void testNameTooLong() {
        IllegalArgumentException thrown = assertThrows(
                IllegalArgumentException.class,
                () -> {
                    new Task( taskID: "12345",  name: "Michael David Johnson",  description: "description");
                });

        assertTrue(thrown.getMessage().contains("Invalid name"));
    }
```

```
✓ TaskTest
    ✓ testNameTooLong()
    ✓ testDescriptionTooLong()
    ✓ testSetNameValid()
    ✓ testValidTask()
    ✓ testDescriptionWithNull()
    ✓ testTaskIDWithNull()
    ✓ testSetDescriptionValid()
    ✓ testNameWithNull()
    ✓ testTaskIDTooLong()

✓ Tests passed: 9 of 9 tests – 28 ms

/Users/swinkimac/Library/Java/

Process finished with exit cod
```

For the ***Appointment*** feature, I tested scheduling logic. My focus was to prevent overlapping appointments, validate null inputs, and check for invalid date formats. These tests aligned directly with the software requirements, ensuring that appointments adhered to constraints and business logic.

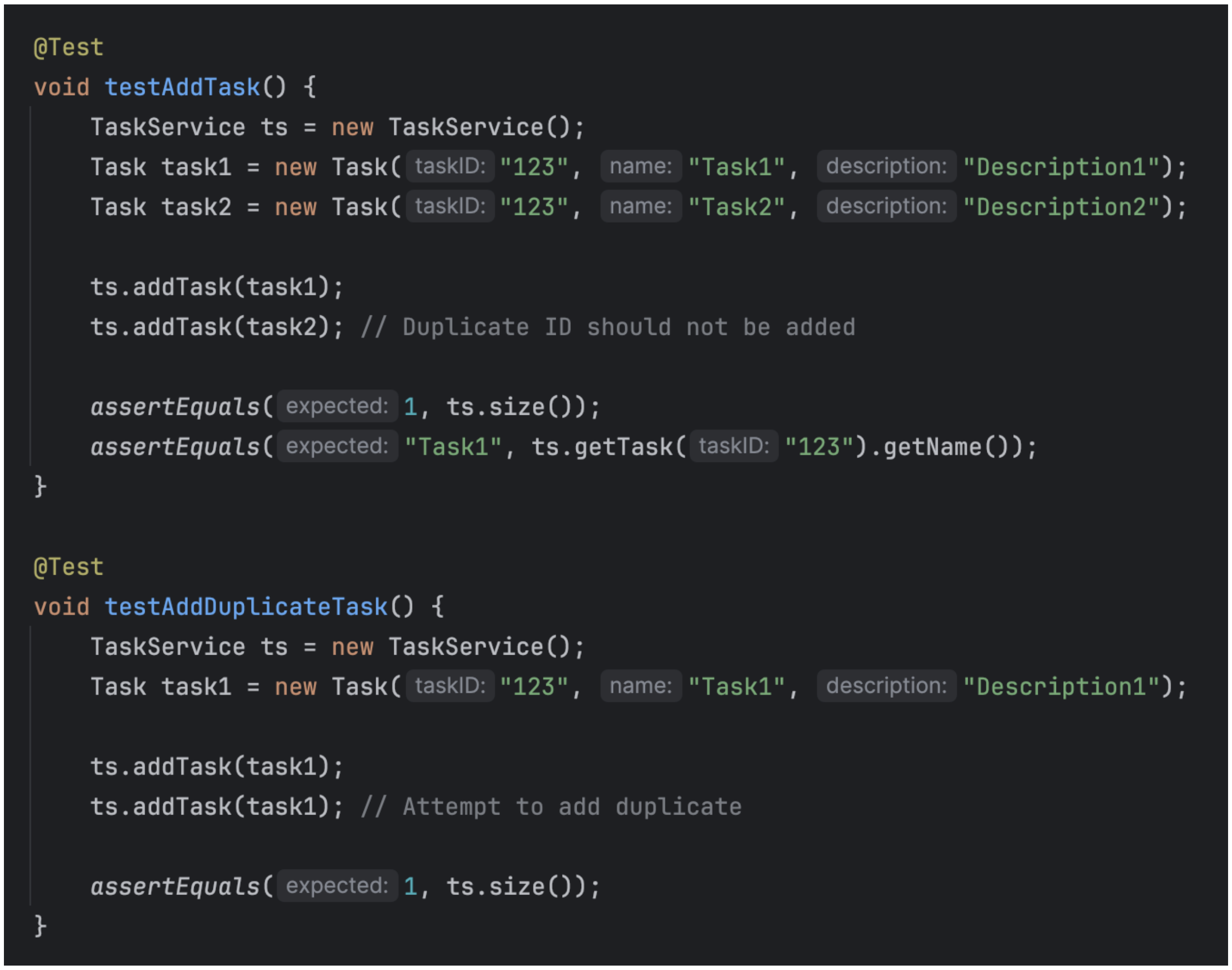
Appointment Test Examples:



```
class AppointmentTest {

    Appointment appt = new Appointment( appointmentID: "A123",  14 usages
            new Date((2024-1900), Calendar.DECEMBER, date: 29),
            description: "Write description here");

    @Test
    void testSetDateField() {
        Date prevDate = new Date((2023-1900),Calendar.APRIL, date: 15);
        Date presentDate = new Date((2025-1900),Calendar.JANUARY, date: 21);
        appt.setDateField(presentDate);
        assertEquals(appt.getDateField(), presentDate);
        assertThrows(IllegalArgumentException.class,
                () -> appt.setDateField(prevDate));
        assertThrows(IllegalArgumentException.class,
                () -> appt.setDateField(null));
    }

    @Test
    void testSetDescription() {
        appt.setDescription("Write description here");
        assertEquals( expected: "Write description here", appt.getDescription());
        assertThrows(IllegalArgumentException.class,
                () -> appt.setDescription(null));
        assertThrows(IllegalArgumentException.class,
                () -> appt.setDescription("abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz"));
    }

    @Test
    void testGetAppointmentID() { assertEquals( expected: "A123", appt.getAppointmentID()); }

    @Test
    void testGetDateField() {
        assertEquals(new Date((2024-1900), Calendar.DECEMBER, date: 29),
                appt.getDateField());
        assertThrows(IllegalArgumentException.class,
                () -> appt.setDateField(null));
    };

    @Test
    void testGetDescription() {
        assertEquals( expected: "Write description here", appt.getDescription());
        assertThrows(IllegalArgumentException.class,
                () -> appt.setDescription(null));
    }
```

```
✓ AppointmentTest
    ✓ testSetDateField()
    ✓ testSetDescription()
    ✓ testAppointmentConstructor()
    ✓ testGetDescription()
    ✓ testGetDateField()
    ✓ testDescription()
    ✓ testGetAppointmentID()

✓ Tests passed: 7 of 7 tests – 39 ms
/Users/swinkimac/Library/Java/Jav

Process finished with exit code
```

My testing approach aligned closely with the functional requirements for each feature. For example, the requirements stated that tasks must have unique IDs and appointments could not overlap. I wrote tests that directly validated these behaviors.

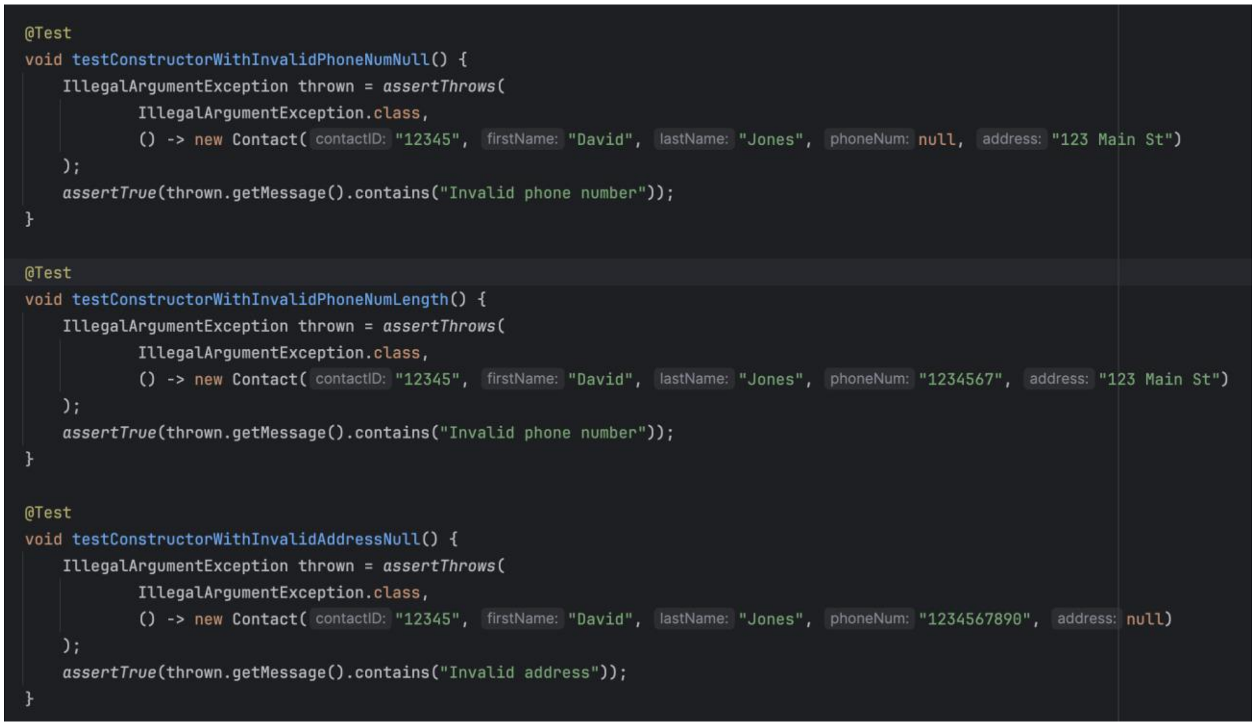


```java
@Test
void testAddTask() {
    TaskService ts = new TaskService();
    Task task1 = new Task( taskID: "123",  name: "Task1",  description: "Description1");
    Task task2 = new Task( taskID: "123",  name: "Task2",  description: "Description2");

    ts.addTask(task1);
    ts.addTask(task2); // Duplicate ID should not be added

    assertEquals( expected: 1, ts.size());
    assertEquals( expected: "Task1", ts.getTask( taskID: "123").getName());
}

@Test
void testAddDuplicateTask() {
    TaskService ts = new TaskService();
    Task task1 = new Task( taskID: "123",  name: "Task1",  description: "Description1");

    ts.addTask(task1);
    ts.addTask(task1); // Attempt to add duplicate

    assertEquals( expected: 1, ts.size());
}
```

By carefully designing tests that addressed edge cases and error conditions, I was able to confirm that my approach met the project's requirements.

As for the quality of my JUnit Tests, I measured the overall quality by evaluating the code coverage percentage. I aimed for a coverage level above 80%, ensuring that critical methods and edge cases were thoroughly tested. By focusing on input validation, exception handling, and boundary testing, I was able to write effective and reliable tests. This level of coverage gave me confidence that my code would behave correctly in various scenarios.

Writing the JUnit tests was a valuable experience that reinforced the importance of both technical soundness and efficiency, especially as a new programmer. To ensure technical soundness, I wrote tests to validate every method's functionality while including null checks and exception testing,



```java
@Test
void testConstructorWithInvalidPhoneNumNull() {
    IllegalArgumentException thrown = assertThrows(
            IllegalArgumentException.class,
            () -> new Contact( contactID: "12345",  firstName: "David",  lastName: "Jones",  phoneNum: null,  address: "123 Main St")
    );
    assertTrue(thrown.getMessage().contains("Invalid phone number"));
}

@Test
void testConstructorWithInvalidPhoneNumLength() {
    IllegalArgumentException thrown = assertThrows(
            IllegalArgumentException.class,
            () -> new Contact( contactID: "12345",  firstName: "David",  lastName: "Jones",  phoneNum: "1234567",  address: "123 Main St")
    );
    assertTrue(thrown.getMessage().contains("Invalid phone number"));
}

@Test
void testConstructorWithInvalidAddressNull() {
    IllegalArgumentException thrown = assertThrows(
            IllegalArgumentException.class,
            () -> new Contact( contactID: "12345",  firstName: "David",  lastName: "Jones",  phoneNum: "1234567890",  address: null)
    );
    assertTrue(thrown.getMessage().contains("Invalid address"));
}
```

Efficiency also played a role as a key priority. This allowed me to be able to ensure that my tests were concise and reusable. For instance, I utilized shared test methods to avoid redundancy when testing multiple cases. This approach improved test maintainability and readability.

Reflection

I primarily employed unit testing, boundary testing, and exception testing in this project. Unit testing allowed me to validate individual methods in isolation. For example, I ensured that the *Task* class correctly handled invalid dates and null inputs. Boundary testing helped verify edge cases like minimum and maximum input limits, while exception testing ensured that proper exceptions were thrown for invalid inputs.

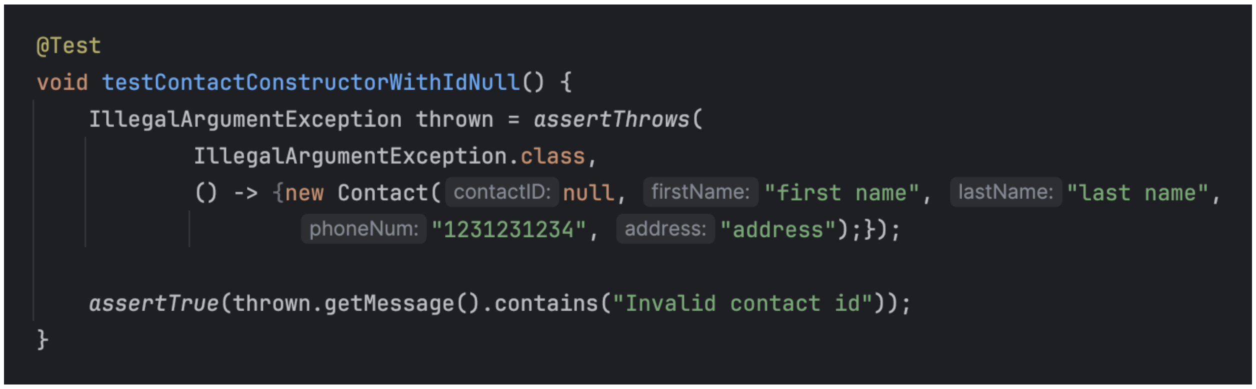
There were some testing techniques I did not use in this project such as integration testing. This testing is designed to verify the interaction between multiple components, but it was outside the scope of this unit-focused project. Similarly, system testing evaluates the complete application, which wasn’t necessary since I was only testing individual features.

Each technique has its own practical applications. While conducting research I found out that unit testing works well for small projects or individual components, whereas integration and system testing are crucial (and mainly) for complex applications with multiple interconnected modules.

Mindset

Throughout this project, I adopted a cautious mindset as a software tester. I approached each test with the assumption that errors could (and most likely) will exist, even in simple methods. For example, when testing the Appointment class, I realized the complexity of ensuring no overlapping schedules. This made me appreciate the interdependencies in the code and highlighted the importance of testing edge cases, such as invalid dates or null values.

To limit bias, I reviewed the code objectively and wrote tests to target potential failure points. And though writing tests for my own code was challenging, I can see how easy it is to assume everything works correctly. For example, I initially overlooked a null input check for the Contact class, but writing the following test exposed the issue:



```java
@Test
void testContactConstructorWithIdNull() {
    IllegalArgumentException thrown = assertThrows(
            IllegalArgumentException.class,
            () -> {new Contact( contactID: null,  firstName: "first name",  lastName: "last name",
                    phoneNum: "1231231234",  address: "address");});

    assertTrue(thrown.getMessage().contains("Invalid contact id"));
}
```

Bias can be a concern when developers test their own code, which is why I tried to approach testing with a critical eye, focusing on breaking the code rather than confirming it worked.

As a software engineering student, I recognize the importance of being disciplined in my commitment to quality. Cutting corners during testing might save time in the short term, but it will ultimately lead to bugs, rework, and higher costs in the long run. For example, skipping edge case testing on input validation could allow invalid user data to enter the system, causing failures later in the development.

To avoid technical debt, I plan to write comprehensive unit tests for all new features and use tools like JaCoCo to track code coverage. Additionally, I will adopt best practices like continuous integration and peer code reviews to maintain code quality. For example, in this project, I ensured every method I wrote had at least one unit test to validate its behavior.

In conclusion, this project helped me to develop a very systematic approach to software testing. By focusing on unit testing and edge case validation, I was able to meet the software requirements and ensure the quality of my JUnit tests. Writing these tests reinforced the importance of cautious and unbiased testing. Moving forward, I plan to stay committed to maintaining high-quality code by writing thorough tests and avoiding shortcuts that lead to technical debt.

Sources

JUnit Documentation:
Oracle. (n.d.). *JUnit 5 user guide*. Retrieved June 14, 2024, from
https://junit.org/junit5/docs/current/user-guide/

Unit Testing Best Practices:
Fowler, M. (2019). *Refactoring: Improving the design of existing code* (2nd ed.). Addison-Wesley Professional.

Code Coverage & Testing Principles:
Sharma, A., & Kumar, V. (2020). *Software testing: Principles and practices*. Pearson Education.

Technical Debt:
Cunningham, W. (1992). *The WyCash portfolio management system*. Presented at OOPSLA '92.

Testing Techniques:
Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing* (3rd ed.). John Wiley & Sons.

Software Development Best Practices:
Beck, K. (2002). *Test-driven development: By example*. Addison-Wesley.