**7-2 Project Two Submission**

Chris Bridges

Southern New Hampshire University

CS 320 Software Test, Automation QA

Professor Joseph Martinez

15 April 2025

During the development of the mobile application's three core features: ContactService, TaskService, and AppointmentService , I employed a comprehensive unit testing approach using JUnit to ensure each feature functioned following the software requirements. Each component was validated independently through test cases designed to assess both positive and negative behaviors.

For the ContactService, my testing strategy included verifying the creation of valid contact entries, enforcing unique IDs, and confirming that appropriate exceptions were thrown when invalid data was passed. For instance:

@Test

public void testAddDuplicateContactThrowsException() {

Contact duplicate = new Contact("12345", "Alice", "Smith", "1234567890", "123 Main St");

contactService.addContact(duplicate);

assertThrows(IllegalArgumentException.class, () -> contactService.addContact(duplicate));

}

This is aligned with the requirements that contact IDs must be unique and non-null.

For the TaskService, I tested the length and null constraints of both task names and descriptions. My tests asserted both the upper limit of the name length and rejected values that exceeded the limit:

@Test

public void testTaskNameCannotExceedMaxLength() {

assertThrows(IllegalArgumentException.class, () -> new Task("001", "ThisNameIsWayTooLongToBeValid", "Valid description"));

}

The AppointmentService required testing against date validation, especially to ensure that past dates were rejected. I used a dynamic date setup to guarantee accuracy:

@Test

public void testAppointmentDateCannotBeInPast() {

Date pastDate = getPastDate();

assertThrows(IllegalArgumentException.class, () -> new Appointment("A001", pastDate, "Routine check"));

}

My testing approach aligned with the software requirements, which required limitations on string lengths, date validation, and null checks. All of these were covered in my test cases.

The overall quality of my tests was confirmed through coverage percentages consistently above 80%. I ensured that both true and false branches of conditional statements were executed in tests. For example, I verified not just that a name over 10 characters throws an exception, but that exactly 10 characters are accepted.

Writing these JUnit tests helped me understand how to structure tests around boundaries and edge cases. I found that writing small, focused tests increased clarity and made debugging easier. I found that starting from the top function and working my way through each possibility allowed me to compile a more comprehensive set of test cases.

To ensure the code was technically sound, I structured tests using clean assertions, isolated test cases, and used helper methods. I used targeted assertions like “assertEquals("Alice", contact.getFirstName());” to ensure my code was technically sound. I created test setups, which allowed the tests to be isolated and repeated. For example:

@BeforeEach

public void setUp() {

contactService = new ContactService();

contact = new Contact("97451", "Jimmy", "Dean", "9745167890", "123 Main St");

contactService.addContact(contact);

}

Efficiency was maintained by avoiding unnecessarily repeated object creation and by using helper methods to generate reusable date instances. Rather than duplicating logic to create future dates, I created a utility method to reduce redundancy, make the logic cleaner, and more efficient.

private Date getFutureDate() {

Calendar cal = Calendar.getInstance();

cal.add(Calendar.DATE, 1);

return cal.getTime();

}

The testing techniques I used included equivalence partitioning, boundary analysis, and negative testing (Garcia, 2017). I tested acceptable lengths for names and descriptions as well as values just over the maximum. Negative testing was used to confirm that invalid inputs properly triggered exceptions.

Techniques I did not use include system and integration testing. System testing checks to see how all the components work together at the unit integration level, and integration testing is suited for evaluating interactions between components, which were not requirements of this project.

Boundary analysis ensures inputs like phone numbers, dates, or addresses are handled safely (Hambling, 2019). Equivalence partitioning helps reduce redundant testing while maintaining effectiveness. Negative testing ensures software behaves securely and predictably in failure scenarios.

As a tester, I used a creative and structured mindset to think of edge cases as well as thoroughly test each component. I assumed that errors could occur with what appears to be valid input. For example, I tested a valid-looking phone number like "123456abcd" to confirm it failed validation. I also made sure that null values were tested across every setter.

To avoid bias, I treated the code I wrote as a friend created it. You always seem to find flaws in someone else’s work much faster than you can with your own. I separated the logic of my classes from the logic of my tests. I wrote tests first and evaluated the outcomes before looking into fixing any errors in the code itself.

As a software engineer, commitment to quality means not cutting corners. Skipping tests or validations could allow critical bugs to slip into production. I plan to avoid technical debt by writing code that is modular and testable while automating regression tests to catch issues early.

A disciplined approach to testing helps maintain long-term reliability and reduces the cost of fixing bugs in production. I plan to document test cases thoroughly and regularly refactor them to ensure clarity and maintainability.

**References**

García, B. (2017). *Mastering software testing with junit 5: Comprehensive guide to develop high quality java applications*. Packt Publishing.

Hambling, B. (2019). Static testing . In *Software Testing : An ISTQB-BCS Certified Tester Foundation Guide* (4th ed., pp. 73–155). essay, BCS Learning & Development Limited.