Project Two - Reflections

Melissa Chessa

Southern New Hampshire University

CS-320 - Software Test, Automation QA 2025 C-4

Professor Angel Cross

08-17-2025

Project Two – Reflections

Summary:

Unit Testing Approach

For this project, I wrote tests for the Contact, Task, and Appointment classes and their service classes. I used both positive tests (making sure valid inputs worked) and negative tests (making sure bad inputs threw exceptions). For example, in *ContactTest* I confirmed a valid object was created:

```
Contact contact = new Contact("12345", "John", "Doe", "1234567890", "123 Main St");
assertEquals("John", contact.getFirstName());
```

I also tested invalid inputs to make sure the rules were enforced:

```
assertThrows(IllegalArgumentException.class, () ->
   new Contact(null, "John", "Doe", "1234567890", "123 Main St"));
```

This way I knew the code was working correctly for both good and bad inputs (JUnit User Guide, 2024).

Alignment to Requirements

The tests were written directly to match the requirements. For example, the requirement that appointments couldn't be in the past was tested like this:

```
Date pastDate = new Date(System.currentTimeMillis() - 1000);
assertThrows(IllegalArgumentException.class, () ->
   new Appointment("A1", pastDate, "Doctor visit"));
```

So the tests lined up with the requirements exactly, instead of me just testing random things (CS 320 Module Four Tutorial Using JUnit, 2025).

Quality of JUnit Tests

I know my tests were solid because the coverage report showed high percentages, and every method got tested. I didn't just test "happy path" cases; I also hit edge cases and exceptions. That's how I know the tests would catch problems if they came up.

Experience Writing JUnit Tests

The biggest challenge was just remembering to test the edge cases, like when strings are too long or values are null. For example, in TaskTest I made sure updates worked:

```
task.setTaskDescription("Updated description here");
assertEquals("Updated description here", task.getTaskDescription());
```

It made me think about what could go wrong, not just what should work.

Technically Sound Code

I made the code sound by checking the important rules directly. For example, in ContactServiceTest I confirmed duplicate IDs weren't allowed:

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

This showed the service logic worked consistently (JUnit Tutorial With Examples, 2024).

Efficient Code

I kept the tests efficient by not writing the same test repeatedly. Instead, I grouped invalid cases when I could, using assertThrows to cover multiple paths in one run. That made the tests clear and less repetitive.

<u>Reflection:</u>

The main techniques I used were unit testing and boundary testing. Unit tests focused on individual methods, and boundary tests checked edge cases like string length limits.

Techniques I didn't use were integration testing and system testing. Those are useful when multiple classes or the whole system work together, but this project only needs class-level testing. In real projects, integration and system testing would be important for databases, APIs, or user interfaces.

I worked with caution because small mistakes could cause big problems. For example, I had to be careful with appointment dates since time-based bugs can be tricky. Testing helped me respect how each rule connects to the others.

I also tried not to assume anything. Even if I thought something "should" work, I tested it anyway. For example, I still tested that setDescription checked the character limit, even though I thought it was obvious. That helped keep me from being biased.

Bias is always a risk if you're testing your own code. It's easy to assume your logic is fine. That's why I think testers have to be strict about checking everything.

Being disciplined is huge here. If you skip testing or cut corners, you'll just run into bigger problems later. For example, if I didn't test exceptions, the program could crash when someone entered bad input. My plan moving forward is to try to always write both positive and negative tests so nothing slips through.

References

Baeldung, & Baeldung. (2023, November 17). *A guide to JUnit 5 | Baeldung*. Baeldung. https://www.baeldung.com/junit-5

Basili, V. R., & Selby, R. W. (2006). Comparing the effectiveness of software testing strategies. *IEEE transactions on software engineering*, (12), 1278-1296. https://ieeexplore.ieee.org/abstract/document/1702179

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing : An istqb-bcs certified tester foundation guide - 4th edition*. BCS Learning & Development Limited. https://ebookcentral.proquest.com/lib/snhu-ebooks/detail.action?docID=5837074#

Jakubiak, N. (2025, May 2). *JUNit tutorial with examples: Setting up, writing, and running Java unit tests*. Parasoft. https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/

JUnit 5 User Guide. (2024). *Assertions and assumptions*. https://junit.org/junit5/docs/current/user-guide

JUnit Tutorial With Examples: *Setting Up, Writing, and Running Java Unit Tests*. (2024). Tutorials Point.

CS 320 Module Four *Tutorial Using JUnit*. (2025). Southern New Hampshire University.

Luo, L. (2001). Software testing techniques. *Institute for software research international Carnegie mellon university Pittsburgh, PA*, *15232*(1-19), 19. https://www.cs.cmu.edu/~luluo/Courses/17939Report.pdf