**Summary and Reflections Report**

**Grand Strand Systems**

Ronnie Corpuz

CS 320 Software Test, Automation QA 2025

Professor Kalysa Wilson

February 18, 2025

**Summary**

**Unit Testing Approach for Each Feature**

For this project I used a step-by-step approach to unit testing to make sure each feature worked correctly. Each feature had its own set of rules, so I wrote tests to check that everything followed those rules. Here is how I tested each service:

* **Contact Service**. I wrote tests to make sure contacts could be added updated and deleted. I also checked that invalid contacts were rejected such as those with missing names, phone numbers that were not exactly ten digits or addresses that were too long.
* **Task Service**. I focused on making sure tasks could be added updated and deleted while keeping the task ID unique and unchangeable. I tested different scenarios including valid and invalid names and descriptions to make sure all rules were followed.
* **Appointment Service**. Since appointments require future dates, I tested to confirm that past dates were not allowed. I also verified that appointment IDs were unique and that descriptions followed the length limit.

I tested each service separately to make sure they worked on their own before worrying about how they might work together.

**Alignment to Software Requirements**

The tests matched the software requirements closely. I made sure that

* Each contact task and appointment had a unique ID.
* Required fields like names, descriptions, phone numbers and dates were properly validated.
* Bad inputs such as missing names or past dates were rejected.

For example, in the Contact Service I tested that adding a duplicate ID would cause an error

java

CopyEdit

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

This test made sure the system followed the rule that no two contacts could have the same ID.

**Quality of JUnit Tests**

The effectiveness of the tests was measured using code coverage. This showed how much of the code was actually tested. The final results were:

* **Contact Service**. 89.1% coverage
* **Task Service**. 87.8% coverage
* **Appointment Service**. 87.2% coverage

These high coverage numbers show that most of the code was tested. While this does not guarantee that every possible issue was caught it does mean that the main logic of each service was thoroughly checked.

**Experience Writing JUnit Tests**

**Ensuring Code Quality**

I made sure my tests were well written by

* Using assertions like assertEquals assertThrows and assertNotNull to compare expected and actual results
* Testing edge cases to catch potential problems
* Checking for invalid inputs to prevent errors later

For example, I tested that an appointment cannot have a past date

java

CopyEdit

assertThrows(IllegalArgumentException.class, () -> new Appointment("AP123", pastDate, "Old Meeting"));

This test makes sure that users cannot create appointments for dates that have already passed which helps prevent scheduling mistakes.

**Writing Efficient Code**

I kept the tests efficient by

* Grouping similar tests together to avoid writing the same kind of test multiple times
* Using reusable code to set up test data instead of repeating the same setup steps

For example, instead of writing separate tests for every possible invalid task name I combined them into one test:

java

CopyEdit

@Test

public void rejectInvalidTaskName() {

assertThrows(IllegalArgumentException.class, () -> new Task("T001", null, "Valid description"));

assertThrows(IllegalArgumentException.class, () -> new Task("T002", "ThisNameIsWayTooLongToBeValid", "Valid description"));

}

This makes the test suite easier to read and maintain.

**Reflection**

**Testing Techniques Used**

1. **Boundary Testing**
   * I tested values that were right at the limit such as task IDs that were exactly ten characters long and those that were one character too long to make sure the system rejected invalid inputs.
2. **Equivalence Partitioning**
   * I divided possible inputs into groups and tested only a few examples from each group. This way I did not need to write a test for every possible invalid phone number just one too short one too long and one that was valid.
3. **Exception Testing**
   * I made sure that invalid operations caused errors instead of allowing bad data to get into the system.

**Testing Techniques Not Used**

1. **Integration Testing**
   * I did not test how different services worked together because I was only responsible for unit testing each one separately.
2. **Performance Testing**
   * I did not measure how fast the system ran since that was not part of the requirements.
3. **Security Testing**
   * I did not check for security issues like hacking attempts because this project did not include a database or online access.

**Practical Uses of These Techniques**

* Boundary testing is useful for **checking user input in forms**.
* Equivalence partitioning helps make **automated testing faster** by reducing the number of test cases needed.
* Exception testing is important for **making sure a system does not crash when something unexpected happens**.

**Mindset**

**Caution in Software Testing**

I took a careful approach to testing because even small mistakes can cause big problems. I focused on

* Finding edge cases like empty strings or very long inputs
* Making sure errors were caught early before they could cause bigger issues

For example, I made sure that phone numbers had to be exactly ten digits

java

CopyEdit

assertThrows(IllegalArgumentException.class, () -> contact.setPhone("12345"));

If this rule was not enforced users could enter incomplete phone numbers which would cause problems later when trying to contact them.

**Avoiding Bias in Testing**

Bias can be a problem if a developer assumes their code works without testing enough different cases. I avoided bias by

* Testing both successful and failed cases
* Not assuming that inputs would always be correct

For example, I originally thought that descriptions would be trimmed automatically but a test failure showed they were not. This led me to improve the code by adding trimming logic.

java

CopyEdit

assertEquals("Trimmed Description", appointment.getDescription());

Without this test, users could have accidentally entered extra spaces and not realized that it might cause problems later.

**Commitment to Quality and Avoiding Technical Debt**

Cutting corners in testing leads to more problems in the future. I made sure to

* Keep test coverage above eighty percent to reduce the chance of missed bugs
* Write clear, well-structured tests that would be easy to update in the future
* Catch problems early so they do not turn into expensive fixes later

For example, refactoring my test cases reduced the number of lines of code and made them easier to manage

java

CopyEdit

assertThrows(IllegalArgumentException.class, () -> new Task("T001", null, "Valid description"));

assertThrows(IllegalArgumentException.class, () -> new Task("T002", "ValidName", null));

If I had ignored edge cases future developers might have struggled with unexpected bugs which could have required major fixes later.

**Conclusion**

Writing effective unit tests was an important step in making sure the Contact Task and Appointment services worked correctly. By using boundary testing exception testing and careful test design I was able to follow the project requirements while keeping test coverage above eighty seven percent for all services.

Sticking to a disciplined approach to testing helped make the code reliable and easy to maintain. In future projects I would consider adding integration tests to check how services work together and performance tests to see how well they handle larger amounts of data.

This project reinforced the importance of unit testing in building high quality software that meets real world needs.