Kyla Gray

**Project 2**

1. **Summary**

For each of the Appointment, Contact and Test features, there were several techniques used to test their functionality and compatibility. Both static and dynamic testing were utilized in order to ensure alignment with the specifications while also confirming that the behavior of the program was as expected when it was run. These techniques were implemented modularly, verifying that any new units of code that were implemented were compatible with the overall program.

This approach was closely aligned to the requirements. For instance, all three features needed to have a unique ID that was not longer than 10 characters and couldn’t be updated. Each feature contained these similar lines of code:

if(contactID == null || contactID.length() > 10) {

throw new IllegalArgumentException("Invalid ID");

}

and each ID field was created as a constant for the change prevention. In order to test that the exceptions were thrown as designed, I performed unit testing on each exception with the type of test shown below. There was also a test to verify that the ID field was not null.

@Test

void testContactIDTooLong() {

Assertions.assertThrows(IllegalArgumentException.class, () -> {

new Contact("contact12345", "Kyla", "Gray", "1213409853","303 Test Rd");

});

}

For units that were not expected to throw exceptions, the functionality was tested while ensuring that a field was not updated if it did not meet the requirements, as shown below.

@Test

void testSetFirstName() {

Contact contact2 = new Contact("contact2", "Kyla", "Gray", "1213409853","303 Test Rd");

contact2.setFirstName("Kyla2");

assertEquals("Kyla2", contact2.getFirstName());

}

Based on the coverage percentage, these tests were effective. There was an overall coverage percentage of 87.4%, with each individual feature ranging between 89% to 97% of coverage. 80% coverage is good to aim for since attempting to reach higher levels could become costly and do not automatically guarantee that a program has been completely and thoroughly tested.

To ensure my code was technically sound, I used a combination of input validation, exception handling, and assertions. All three of these techniques were shown when I described how my approach was aligned to the software requirements. For example, in the testContactIDTooLong test, input validation was used to ensure that the ID for the Contact class was less than 10 characters. Similar tests were used for the Appointment and Task classes. Exception handling was exemplified within the same test. If the ID was too long, an exception was thrown and the attempted ID update was rejected. To test the functionality, the same unit of code can be referenced, as an illegal argument exception was thrown if the ID requirements were not met. There were also several “assert all” assertions used to test constructors such as in the Appointment class shown below.

Appointment appointment = new Appointment("Test1", testDate, "Appointment 1");

assertAll("appointment",

() -> assertEquals("Test1", appointment.getAppointmentID()),

() -> assertEquals(testDate, appointment.getAppointmentDate()),

() -> assertEquals("Appointment 1", appointment.getAppointmentDescription())

);

}

In order to ensure efficiency, data structures such as hashmaps were used to store Appointment, Contact, and Task objects by their respective ID’s as illustrated below.

private final Map<String, Appointment> appointmentMap;

public AppointmentService() {

this.appointmentMap = new HashMap<>();

}

I also ensured that my code was modularizable, as each object had similar requirements and functionality.

1. **Reflection**

Both static and dynamic testing techniques were used throughout this project. My code was manually reviewed and compared against the specifications. During unit testing, more specific techniques such as test coverage and error guessing were used. Regarding test coverage, condition coverage was used. Within all three object classes, the name field was tested when it was null and not null. It was also tested for the correct and incorrect lengths. For error guessing, there were tests that used ID’s just outside the maximum length as miscounting the necessary length could be common.

Some other techniques that could have been utilized are equivalence partitioning, acceptance testing, and more in-depth integration testing. Regarding equivalence partitioning, there was not a minimum length for the ID or description fields. If this were implemented, partitions of lengths below the minimum, the correct number, and lengths above the maximum could have been tested. Acceptance testing was not done since the client or end users have not used or tested the program which is vital before and after deployment. There was integration testing done between units, such as the Contact Service class tests utilizing both the creation of a Contact object using the constructor, the adding of the contact into the map, and the updateContact function. However, there was no integration testing done between any of the three features.

Using these unit and integration techniques allow potentially critical or detrimental errors to be caught earlier and lower technical debt. This produces higher quality code. Referring to specifications often ensures that the project is on the right track. Test coverage, error guessing, and equivalence partitioning promote thorough testing and help to ensure that the program works as expected. Acceptance testing is vital because it catches bugs that developers may have missed or wouldn’t be aware of during the development process. The client and end users are the ones who would use the program the most, and the way they use it may be different than what would be expected from a developmental perspective. Integration testing is also essential to ensure that each piece of the program is compatible and does not corrupt the program.

* 1. **Mindset**

I attempted to employ more caution throughout this project than usual because I became more aware of my own biases. Even though the functionality of each object was similar, each object and their services needed to be tested adequately with test coverage reaching 80% or higher. This made me test more carefully and explore more branches than usual. Without noticing how each object and their services were interconnected, the code also wouldn’t be as modularized and I may not have treated the requirements with as much care.

Some ways I attempted to limit bias in my code was to frequently refer to the specifications and try not to assume that the basic functionality of the code worked without adequate testing. In referring to the specifications, it allowed me to build the program based on what was needed instead of personal preferences for the program. Reviewing your own code can provide false confidence in the program. I may believe that the constructors I make for each object work as expected without testing, but if I omit testing on the very core of an object, the rest of the project will have several logic errors and may not work at all.

It is essential to be disciplined and committed to producing quality code and testing it thoroughly. If even the basic functionality of a program is not tested, it could end up becoming incredibly costly for a company and the program itself. I plan to not only review my code myself, but have my code reviewed by others who may test more objectively than myself since it is not their code. Testing often will avoid more severe technical debt later on. For example, in 2012, the Knight Capital Group was under pressure to “forgo proper documentation and testing methods” for their new features in their trading software (Heijstek, 2025). When the New York Stock exchange opened, people were able to buy stocks without authorization or proper backing from the company, and subsequently ended up in a $440 million loss. If the developers were able to test their new features adequately, this could have been prevented.

**Resources**

Trova, G. (2025, April 7). *Five examples of technical debt: How software failures and productivity loss go hand-in-hand*. SIG. https://www.softwareimprovementgroup.com/technical-debt-examples-software-failure-examples/