CS 320 Project Two

# Testing Approach

To ensure alignment with the requirements, I tested each class against the listed requirements outlined in each milestone. For instance, for the task class, the task ID was not to be null or longer than 10 characters. Therefore, I used the following test cases, testIDTooLong() and taskIDNull() to verify that the program throws an exception when the ID is initiated with too many characters or is null:

*@Test*

void testTaskClass() {

Task task = new Task("1234567890", "Task name", "Task description" );

*assertEquals*("1234567890", task.getID());

*assertEquals*("Task name", task.getName());

*assertEquals*("Task description", task.getDescription());

}

*@Test*

void taskIDTooLong() {

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

new Task("12345678901", "Task name", "Task description" );

});

}

I used assertEquals to verify valid input and assertThrows to verify that invalid input, such as a null or too-long ID, results in an exception. Doing this for every listed requirement helped me ensure the effectiveness of my tests. Thus, when 100% coverage is reached, it tells me that every line of code is run and every requirement is tested.

# Experience Writing Junit Tests

To ensure that my code was technically sound, I made sure to perform checks to validate all fields to prevent invalid or null input. For example, below is the constructor for my Appointment class.

/\*\*

\* Constructor

\* **@param** id

\* **@param** appointmentDate

\* **@param** description

\*/

public Appointment(String id, Date appointmentDate, String description) {

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

throw new IllegalArgumentException("Invalid Appointment ID");

}

this.id = id;

if (appointmentDate == null || appointmentDate.before(new Date())){

throw new IllegalArgumentException("Invalid Appointment Date");

}

this.appointmentDate = appointmentDate;

if (description == null || description.length() > 50) {

throw new IllegalArgumentException("Invalid Appointment Description");

}

this.description = description;

}

Here, checks are performed to make sure that the ID is not null or too long, that the appointmentDate is not in the past, and that the description is not null or too long before the object is initiated with the given arguments. I then tested these behaviors with both valid input and invalid input, as well as with invalid input, as shown below:

*@Test*

void testAppointmentClass() {

Appointment appointment = new Appointment("0", futureDate, "Appointment description");

*assertEquals*("0", appointment.getID());

*assertEquals*(futureDate, appointment.getAppointmentDate());

*assertEquals*("Appointment description", appointment.getDescription());

}

*@Test*

void testIDTooLong() {

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

new Appointment("12345678901", futureDate, "Task description" );

});

}

*@Test*

void testIDNull() {

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

new Appointment(null, futureDate, "Task description" );

});

}

One way I facilitated efficiency is by creating reusable test data. For instance, the Appointment test required validating that the date is not in the past. Instead of creating a new date object for each test, I took the approach of creating a future date and a past date as variables accessible to each test case, thus avoiding unnecessary repeated initiations:

Date futureDate = new Date(System.*currentTimeMillis*() + 86\_400\_000);

Date pastDate = new Date(System.*currentTimeMillis*() - 86\_400\_000);

# Testing Techniques

One technique that I employed in this project was equivalence partitioning. Equivalence partitioning involves grouping inputs into classes where similar behavior is expected and only testing one value from each partition (Hambling, et al., 2019). In my tests, the most prominent example is testing for future and past dates, as referred to earlier. Given that the code should accept all dates in the future and reject all dates in the past, it was only necessary to test one value from either case. This was also utilized for testing input length, as inputs are considered either to be too long or valid and given this, it would only be necessary to test one case of valid input and one case of input that is too long. Equivalence partitioning helps reduce the number of tests needed and prevents redundant testing thus increasing efficiency.

Other black-box testing techniques are largely unused. For example, boundary value analysis involves testing input values at edges and just beyond boundaries (Hambling, et al.). It can be said that this technique is employed in a sense since my tests for valid cases largely utilized strings with the maximum number of characters (i.e., “123456790” when the max limit is 10 characters), and my tests for too-long inputs utilized strings with just 1 more character than the max limit (i.e., “1234568901” when the max limit is 10 characters). Beyond this, however, the application of this technique was minimal. In software development projects, boundary testing is useful for validating constraints and catching errors that are off by one value. Decision table testing involves mapping combinations of conditions to actions (Hambling, et al.). While this technique is unused for this project, it would be useful in situations where whether an action takes place depends on whether a specific set of rules is satisfied. Another unused technique is state transition testing, where behavior depends on the status changes and validation of the transition between states is necessary (Hambling, et al.). This was unused since this characteristic was not present in the project, but would be highly useful in projects where workflow is important and would facilitate clear state definitions.

# Mindset

Throughout the project, I had to exercise caution. When testing the program, it was important to not assume the correctness of the code and to carefully test each aspect critically. For example, when handling the deletion of an object in AppointmentService, I made sure to test for cases where the object does not exist:

*@Test*

void testDeleteNotExist() {

AppointmentService appointmentService = new AppointmentService();

appointmentService.addAppointment(futureDate1,"Description 1");

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

appointmentService.deleteAppointment("1");

});

}

It is also important to appreciate the complexity and interrelationships of the code being tested. In the project, for instance, exception handling for invalid inputs is performed in the Contact, Task, and Appointment classes, where the ContactSerivce, TaskService, and AppointmentService classes rely on those validations. If the underlying classes failed to implement these constraints, the service classes would not function as expected.

To limit the bias in my code, I made an effort to not assume that the code would work as intended. For instance, even though I felt like all entry fields were properly validated, I still wrote tests for cases with null and invalid inputs as well as ensuring that setter functions do not accept illegal inputs. As stated by Cem et al. (2002), while bias cannot be fully eliminated as it is an intrinsic part of how we think, we can limit the biases we have by recognizing and understanding them.

Being disciplined in the commitment to quality is highly critical. Cutting corners when it comes to writing or testing code may be tempting, but doing so risks complications down the development pipeline where fixes would become exponentially more expensive and time-consuming. For instance, if input validation and testing had been neglected in the Task class and it was later integrated into a larger application, any errors that went unnoticed could result in unexpected behavior or crashes. At this point, it would be much more difficult to identify the cause of the issue than if it had been caught through proper unit testing early in development. To avoid technical debt as a practitioner in the field, I would keep this to heart and stay disciplined by writing comprehensive tests to ensure that all constraints are validated and consistently reviewing code for edge cases. By maintaining this mindset, I hope to be able to contribute towards more stable and reliable software while reducing the long-term cost of maintenance.

**References:**

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.

Kaner, Cem Bach, James Pettichord, Bret. (2002). *Lessons Learned in Software Testing - A Context-Driven Approach - 11. Planning the Testing Strategy.* John Wiley & Sons https://app.knovel.com/hotlink/pdf/id:kt011N5K53/lessons-learned-in-software/planning-testing-strategy