Jordan Dayedes

CS 320 Software Test, Automation

10/20/2024

Testing Summary and Reflection

All three programs had specific requirements that shared similarities. For example, the contact, task, and appointment objects all had the same requirements of an immutable ID that couldn’t be longer than 10 characters while also unable to be null. My tests were centered around ensuring through constructor validation as well as not providing a setter for those objects. Unit tests were then written to confirm that if the ID objects were either null, or too long making it invalid, an exception would be raised.

For Contact Service, my unit tests reached a coverage percentage of 86.5% for Contact.java, and a full 100% coverage for ContactService.java. Only my setter functions in Contact.java were missing any coverage, as each function only reached 66.7% coverage, with two of four branches missed. Full coverage could have likely been achieved had more tests been written to confirm the setters would throw an exception if Null.

Next up is Task Service. Both Task.java and TaskService.java reached a test coverage of 100%. However, the overall coverage of all files was 85.2%. This is because the test coverage of the TaskTest.java and TaskService.java were at 72.2% and 84.8% respectively. I’m under the impression that the test coverage of the Junit test files themselves aren’t necessary and can be safely ignored.

Ok, last file. Appointment Service also had two files to be tested. Both Appointment.java and AppointmentService.java achieved 100% coverage. Not much else to say here. Unit tests were written to confirm the Constructors, getters and setters all met requirements. The only tricky aspect was writing the tests to confirm the appointmentDate object met requirements correctly.

I wrote unit tests that confirmed my class objects met the specified requirements. Tests were written to confirm that all object fields met specified criteria, and to check that invalid inputs would throw exceptions and be rejected. Back to the fields that shared requirements between all three main files. Tests were written to confirm the maximum allowed length of an ID, as well as that ID unable to be set as Null. Here’s an example from the Contact Service, contact.java file.

@Test

public void testInvalidContactID() {

assertThrows(IllegalArgumentException.class, () -> {

new Contact(null, "John", "Doe", "1234567890", "123 Main St");

});

}

This test ensures that the contactID object cannot be made Null and throws an IllegalArgumentException if a Null ID is provided. A similar test was written for both Task Service and Appointment Service for each of those files ID objects as well.

I also wrote tests to confirm the setters would allow updates to an object correctly, and if invalid data was provided, an exception would be thrown. Example setter test on the appointmentDescription object from the Appointment Class.

*@Test*

void testSetDescriptionInvalid() {

Date futureDate = createDate(1); // Initially create with a valid future date

Appointment appointment = new Appointment("12345", futureDate, "Initial Description");

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

appointment.setDescription(null); // Invalid case

});

}

This test provides a valid set of field object data, then invokes the setDescription setter function with invalid data to test the setter method will throw an exception when invalid data is provided correctly.

I worked to ensure efficient code by focusing tests on specific edge cases. Instead of writing individual tests for every potential case, I wrote tests against specific cases that the requirements would be known to fail. For example, in the case where an appointment description has the requirement of being neither null nor longer than 50 characters. I would first write one small test to confirm that the function meets the requirements cannot be null.

// Test invalid description (null)

*@Test*

void testInvalidDescriptionNull() {

Date futureDate = createDate(1);

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

new Appointment("12345678", futureDate, null);

});

}

I would also write one more test to ensure that it cannot be over the set character limit.

// Test invalid description (too long)

*@Test*

void testInvalidDescriptionLong() {

Date futureDate = createDate(1);

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

new Appointment("12345678", futureDate, "This appointment description is going to be far too long which is what we are testing for");

});

}

This means that each requirement only needs one test to be satisfied, instead of writing tests for every possible data input. We only check that it cannot be null and cannot be longer than the set condition.

During work on this project, I wrote Unit tests to ensure that each class functioned as expected in isolation. Each test was written against specific parts of the code, or each function individually, to see if the functionality and behavior met expectations. Functional testing was done by writing legal inputs into class objects, and then attempting to update the setters with invalid inputs to ensure they would fail and throw an exception as expected.

I didn’t implement any performance testing. This could have been achieved by writing tests to track the length of time it would take to create or update large amounts of field data in any of the three classes. Such as adding new ID’s and appointment data in Appointment Service, for example. If the system were to use an outside data storage solution, system and security tests would also be needed. These would ensure that the system is safe from exposed data, and vulnerabilities like SQL injection attacks.

I tried to be mindful of all the ways in which my program could fail. To this end, I wrote both positive and negative test cases. Such as this positive test on valid inputs to the constructor here  
  
 *@Test*

void testTaskConstructor() {

// Test a valid task

Task task = new Task("12345", "Test Task", "Test task description.");

*assertEquals*("12345", task.getTaskID());

*assertEquals*("Test Task", task.getTaskName());

*assertEquals*("Test task description.", task.getTaskDescription());

}

I think it can be quite easy to fall into the trappings of looking at the coverage percentage, seeing a high number and thinking the testing is sufficient. It’s probably best to take a step back, and perhaps do at least a lightweight code review with a team member. Just because the test coverage is high, doesn’t mean that the class code is sufficient to meet the client requirements.

Testing early and testing often is a tried-and-true mantra to follow in the effort of avoiding defects and software failures. Competent testing ensures that each class and component all contribute thoroughly to meeting project requirements. My plan to avoid technical debt is to follow a test-driven development plan by writing tests before any actual implementation. I’ll also regularly perform code reviews, like synchronous reviews, and more structured formal reviews too.