Victor Feight

CS-320

Project Two

4/18/21

I created my initial tests based on the given requirements, for example for the Appointment class there was a specific requirement:

*The appointment object shall have a required appointment Date field. The appointmentDate field cannot be in the past. The appointmentDate field shall not be null.*

Within the constructor the Appointment class, Exceptions are thrown for invalid input before an object is able to be created with bad data:

// throw exception if appointment date is before the current day

if(appointmentDate == null || appointmentDate.before(currentDay)) {

throw new IllegalArgumentException("Invalid input");

}

this.appointmentDate = appointmentDate;

In order to accurately turn this requirement into a test case, I created a Junit test cases a Date objects together with a SimpleDateFormat objects to create and format the date, respectively.

SimpleDateFormat dateFormat = new SimpleDateFormat("MM/dd/yyyy");

Date date = null;

try {

date = dateFormat.parse("04/14/2021");

} catch (ParseException e) {

e.printStackTrace();

}

final Date dateCopy = date;

// id too long

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

Appointment appointment = new Appointment("00000000011", dateCopy, "id too long");

});

In this example, the assertThrows method of Assertions allows me to test that an IllegalArgumentException is thrown using Java lamba syntax. It deliberately passes an appointment object with too long of an id, to ensure the correct Exception is thrown. A new Appointment object is created within the test with a copy of the date object as well as a copy of my appointment object, this helped ensure my tests were technically sound. To ensure efficiency, I broke each class into separate test files, with a test for each user requirement. For example files as AppointmentServiceTest.java, AppointmentTest.java contain @Test include tests such as:

public void testExceptions() {

This naming scheme allows for testing of all the exceptions for each class, and this made testing more efficient overall.

In certain cases, as in Contact, I had quite a high test coverage percentage of tests. In other cases, such as AppointmentTest I admittedly could have included more tests on boundary cases for object creations or further date tests.

I was able to use a Test-Driven Development approach where I extracted necessary test cases from each user requirement by analyzing them for each module. After writing a list of test cases for a particular function, I would write the function and ensure all cases passed. Specific software testing techniques I used include Input validation, as most of the input for these modules had to be checked before instantiating the object (to ensure no bad values were entered).

I utilized @Test annotations for the JUnit testing, which allowed for both easier automation and some regression testing to catch any errors after changing a certain function. Certain techniques that were not used include Integration tests, system tests, or blackbox tests. All tests (JUnit) were whitebox, and the codebase was written alongside the test cases. Other blackbox techniques not used include boundary value analysis and equivalence partitioning.

According to our readings in Software Testing: An ISTQB-BCS Certified Tester Foundation, Boundary value analysis involves testing at the boundaries of input, whereas equivalence partitioning involves dividing the input set into different partitions of testable data. The book describes Integration testing as integrating individual modules/functions with other modules/functions to ensure they are compatible in the system. System testing involves validating not only the software works, but that all of the specifications are met in terms of the whole physical computer hardware and software system.

JUnit testing can be adopted in any incremental development environment (ie github or source control) and its uses involve catching bugs early by testing early. Similarly, regression testing is beneficial in catching bugs before they appear late Software Development Life Cycle, which can be exponentially more expensive to fix. Automation technique ensures these tests can be run on a regular basis to test modules. Integration testing can help ensure each of the modules is sending and inputting data correctly between each other. System testing can ensure each physical computer, OS, and browser is working as per specifications of the system. Any form of black box testing can complement white box testing to help test performance issues, or break the code at certain input boundaries without needing access to a codebase.

Mindset

*Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ caution? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.*

In working on this project, I shifted my mindset from that of a developer, frantically hoping to push out working code, to that of an overly analytical tester. I framed each test case in terms of how I could break the functionality, and test each user requirement with high attentiveness. For example, for each class I would methodically check each requirement off the list, and would try to ensure each and every Assertion was tested, as well as object creation, and method testing. Because services used functionality from the respective object classes, attention to creating object data to be used for the services was needed in the test cases.

*Assess the ways you tried to limit bias in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims.*

Yes, since I coded these services I understand that bias can play a role in testing of these functions. However, overcoming that bias by over testing is a necessary habit to achieve good bug-free software. I still struggle a bit with testing each and every case, as can be noted that some of my object’s code coverage is <80%, however I do try to test the most important targeted requirement cases and functionality. However, bias is always something to be considered and this is why test teams are necessary in larger environments.

*Finally, evaluate the importance of being disciplined in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.*

In hopes of becoming a stronger software engineering professional, I understand and appreciate the importance of not cutting corners in Testing the codebase from day 1, even during the requirements gathering stage. Examples include clearing up any ambiguity in requirements from the start by communicating with their source, be they customers or stakeholders. The reason is that catching bugs later on in the lifecycle becomes exponentially more expensive, and having a Test-Driven mindset with the discipline to test every edge case and function means having more bug-free code (and potentially a lot less work) in the end.