Southern New Hampshire University

7-2 Project Two Submission

Jehu A. Domenech Ortiz

CS-320-T2822 Software Test Automation & QA 22EW2

12/08/2022

During the production of the JUnit tests, I focused on making sure that each of the tests for the needed requirements of each feature. Some of the requirements was to avoid the values of variables from being NULL and from exceeding a certain number of characters. To test for these requirements, I made an individual tests for each variable to test for it. For example for the Task feature I have a TaskTest function testTaskClassIDTooLong() and testTaskClassIDNull() which tests whether the ID variable meets the requirement of characters for ID and whether it is not NULL.

Coverage percentage was tested for each of the features in the application. All of the tests passed and they also showed a 100% test coverage in the application this includes the service files which have the update(), add(), and delete() functions.

For making the code technically sound I made the variables private making sure that they can only be modified by running the setter functions. The constructor in each of the feature files ask for the variables and these are then filtered through if statements to check whether they meet the requirements if these do not meet the requirements a IllegalArgumentException() is thrown.

public class Task {

    private String ID = "";

    private String name = "";

    private String description = "";

public Task(String ID, String name, String description) {

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

        throw new IllegalArgumentException("Invalid ID");

    }

    if(name == null || name.length()>20) {

        throw new IllegalArgumentException("Invalid name");

    }

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

        throw new IllegalArgumentException("Invalid description");

    }

    setId(ID);

    setName(name);

    setDescription(description);

}

For the features of this program the use of assertions makes the code efficient in helping with testing. Along with getters and setters that were incorporated in the constructor the system that was incorporated to test whether a variable is appropriate to be set is efficient and effective without being overly complicated.

void testTaskClassIDTooLong() {

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

        new Task("12345678901", "Jehu Domenech", "Domenech");

    });

}

public Appointment(String ID, Calendar date, String description) {

    Calendar currDate = Calendar.getInstance();

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

        throw new IllegalArgumentException("Invalid ID");

    }

    //modify

    if(date == null || currDate.after(date)) {

        throw new IllegalArgumentException("Invalid date");

    }

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

        throw new IllegalArgumentException("Invalid description");

    }

    setId(ID);

    setDate(date);

    setDescription(description);

}

During the development of the code for the program in the past couple of weeks I had to use dynamic testing techniques. Out of all the dynamic testing techniques I used the statement coverage and the conditional coverage techniques, because the program is simple there was no need for any of the other techniques. This technique was applied throughout the development of this all JUnit tests follow a similar structure. With the statement coverage testing technique, I made sure everything was being properly tested and executed and for the conditional coverage I checked that a InvalidArgumentException was thrown whenever an invalid value was given to a constructor.

A dynamic software testing technique that I didn’t use and believe I would frequently use is the Boundary Value Analysis (BVA) technique. This technique tests to verify whether the values that are being used are within the given range. Decision tables and decision coverage techniques would also be frequently used by me if I end up having a section where there are different decisions that can be used. Decision tables tests all the possible combinations that can be made, this is necessary while testing to check for proper result. Decision coverage checks whether all the if conditions in the code are tested.

The decision coverage technique can be applied to check whether the test covers all the statement lines in the programs code this is necessary to check coverage and whether a function or part of code is ever executed. The conditional coverage technique has a useful application when it comes to checking whether all the logical conditions are properly working. The BVA technique is used whenever you’re dealing with a range in your code, preferably you want to check that the biggest and smallest value inside the range work and that a value outside that range does not work. When you have decision trees you need to make sure that all the expected outcomes are output the expected behavior and a decision table keeps track of this. Decision coverage are applied whenever if statements are used, when one if used you need to check for coverage to make sure that they all pass the test.

During the development of this project, I wanted to make sure that I was properly testing to meet all the requirements. It was necessary to research and come up with ways that were appropriate to test each of the features specially for the service files of each of the features. During the making of the test’s files, I had to be cautious and make sure that I didn’t incorrectly set up the tests which happened multiple times but I was able to fix the issues. For the example provided below the test kept failing and I did not understand what the issue was and believed that the implementation of the function for the date was incorrectly set up. The issue was that I needed to change the date using the set() function which I had not done.

@Test

    void testAppointmentClassDateBefore() {

        currDate.set(2021, 02, 01);

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

            new Appointment("1", currDate, "Domenech");

        });

    }

To limit my bias throughout the development of the testing files I focused on testing to meet all the requirements even if I knew that they were correctly setup. For the ID variables they are automatically added into the ArrayList<> by checking for the length of the ArrayList<> and setting the ID value to that. I still decided to test the ID to check whether they would pass the test for being too long or NULL even though being NULL wouldn’t happen and there’s a low chance of it being too long.

@Test

    void testAppointmentClassIDTooLong() {

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

            new Appointment("12345678901", currDate, "Domenech");

        });

    }

    @Test

    void testAppointmentClassIDNull() {

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

            new Appointment(null, currDate, "Domenech");

        });

    }

By being disciplined as a software engineer mistake can be avoided. During the creation of the tests, I noticed that there were issues in the implementation of the code that I had written. It would have been hard to notice if I had not set up the tests and the place where it was most noticeable was during the implementation of the Appointment feature which I had to check whether the date was valid and not the same date or before that date. I had accidentally used before instead of after() making all of the dates after the current day invalid to set for the appointment date.

if(date == null || currDate.after(date)) {

    throw new IllegalArgumentException("Invalid date");

}