Bryce Cooperrider

CS-320

4/21/24

Project 2: Reflection

I believe my testing approach for project one was able to verify all features. All requirements were satisfied in accordance with the requirements for each class. I don’t want to show code snippets for each class because that would fill the document up quite fast, but one example is my Appointment class. This is the code I used for making the class:

    private String appointmentID;

    private Date appointmentDate;

    private String appointmentDescription;

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

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

            throw new IllegalArgumentException("Invalid ID");

        }

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

            throw new IllegalArgumentException("Invalid Appointment Date");

        }

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

            throw new IllegalArgumentException("Invalid Description");

        }

        appointmentID = ID;

        appointmentDate = date;

        appointmentDescription = description;

    }

For this class I wanted to test the submitted fields as they are being passed into the class function. This is a step to help ensure that the field is not null or longer than the maximum number of characters. The fields are also checked during the setters for the class as a secondary measure to ensure the proper strings are in the fields.

I believe my Junit tests to be of quality. I know they were effective based on the number of passed tests. Taking my AppointmentTest.java for example, it passed all seven tests designed to make sure the entered fields were within range. An example test is:

    @Test

    void testLongID() {

        String ID = "12345678901";

        Date date = new Date();

        String appointmentDescription = "This is a valid description";

        IllegalArgumentException exception = assertThrows(IllegalArgumentException.class, () -> {

            new Appointment(ID, date, appointmentDescription);

        });

        assertEquals("Invalid ID", exception.getMessage());

    }

This test passes because it throws an IllegalArgumentException indicating that the string “12345678901” is too long. The maximum characters allowed in the ID is 10, so this works as intended.

I made sure my code was technically sound by keeping it as simple as possible. For example, take the deleteTasks function on TaskService.java:

    public void deleteTasks(String ID) {

        if(tasks.containsKey(ID)) {

            tasks.remove(ID);

        }

    }

There are not many moving parts to removing a task, it just says if the ID is in the tasks list, remove it. Typically, the less moving parts there are, the less room for error exists.

I made sure my code was efficient by constructing each class object efficiently. I briefly touched on this earlier but checking the entered fields as it goes into the class makes sure that there isn’t invalid data in an already constructed object. This way we know that if there is a specific class object created, it should be within the requirements in all fields. My example is from the Task.java

    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);

    }

The biggest technique of test I used was unit testing. Unit testing is when you test individual components of the application to make sure they function correctly. I did this for each class member. Another type of testing used was integration testing. I made multiple files that worked with each other and testing them to see if they work together was a vital part of the project. The last type of testing I used was manual testing. That just involves manually looking over the code to spot any error in thinking or logic that might have occurred.

There are some techniques I didn’t use. One of them is non-functional testing. Non-functional testing is when you test the performance and security of the system. My application is small, so I didn’t feel this was necessary to do. Another type of testing not used was user acceptance testing. We didn’t really have a user to check in with and see if the program met their requirements, rather I made sure it fit in within the given requirements from the prompts.

There is not a single type of test that is better than the other; they all contribute to the project in different ways. Unit testing is important to make sure your individual pieces work correctly, with integration testing being important to make sure multiple pieces function as intended. For techniques I didn’t use, non-functional testing is typically very important to be testing. If the program’s performance is suffering, it makes for a poor end-user experience. No one wants to use a program that isn’t fast and snappy. Security is also very important in order to protect user information. Many lawsuits have come to light in the past from poor security. User acceptance testing is also important because they are the people that will be using the product. It is better to accept their critiques before the product is officially launched to incorporate any helpful feedback received.

The mindset that I went into this project with was that of a tester. I wanted to write code that was easy to follow and easy to test. I thought that would help me down the line to make any changes I needed and to write the test code and make sure the program was functional. It is important to exercise caution because in the real world, many different people will end up using a program. It is hard to account for everything that someone might do, but some of those actions involve unintentionally breaking the program or having a poor experience. If you employ caution, you can code with those aspects in mind therefore writing better code for the situation. It is crucial to take a step back and appreciate the complexity of what your code does so you can grow proud of it, making you want to develop to the best of your ability.

I tried my best to reduce bias by looking at my code from an outside perspective. Bias is a concern for your code, especially if you are testing it yourself. You might think that you coded a function the best way possible until someone else suggests it can be better. Having your own bias get in the way of testing limits the program because critiquing your own work is a hard thing to do. I think that once you acknowledge your bias, you can write better tests for your own code.

Writing disciplined code is something all developers should strive to do. Following standards and outlines make it easier to work with multiple people on the same project, increasing efficiency and behavior or the project as a whole. Cutting corners always comes with a cost of reliability, speed, or security, making it important to take the extra time to develop the codebase properly. It is hard to say how I plan to avoid technical debt in the field, but my strategy is to always be open minded and open to new ideas. There are multiple ways to do the same thing and if someone points out to me that I’m doing something poorly, I want to listen and incorporate that feedback to improve my behavior.