Dale Ayers

4/20/2022

SNHU

[dale.ayers@snhu.edu](mailto:dale.ayers@snhu.edu)

**Project 2**

When it comes to the three features, Contact, Task, and Appointment, we are chiefly concerned with the software requirements. All three have similar requirements that need to be taken into consideration. For example, all three require a unique ID. All three require Strings with a certain number of characters, etc. To achieve the proper requirements, I started at the very beginning. For example, I wrote an if statement with the unique ID first to check if the ID was both not null and under ten characters.

public boolean validateID(String taskID){

if(taskID.length() <= 10 && taskID != null){

return true;

}else{

return false;

}

}

This ensures that when the ID is entered, it aligns with our software requirements that it's both not null and less than or equal to ten characters. But this is only half of the story; now I have to write a Junit test that makes sure that this logic is correct and it does what it says that it's going to.

@Test

void testGetTaskID() {

Task test = new Task("UserID", "Name", "Description");

if(test.getTaskID().length() <= 10 && test.getTaskID() != null) {

}else {

fail("Test Failed");

}

}

The Junit test I wrote for this software requirement looks very similar to the if statement I wrote in the Task class. But this time, I'm going to the end of the process to make sure nothing unexpected happened. Specifically, I'm testing the UserID. If our UserID is anything unexpected, then the test will fail. As simple as the test seems, it is a great concept to live by when testing software; take nothing for granted. If this software were much more complex, there could be many more lines of code that interact with the UserID; those lines of code could affect our UserID in unexpected ways.

Each of my tests was at or above the 80% mark laid out in the software requirements. This means that the tests covered about 80% of the code, of which that code passed all of the required tests. From the 80% test coverage, the passing of all my tests, and the logic of my tests, I can say that my code meets or exceeds the software requirements.

To make sure that my test code was technically sound, it took some thinking. For example, our add and delete functions. How does someone check to see if the methods are functioning correctly? I came up with testing the size of the array list that we are adding and deleting using the Junit function assertEquals().

@Test

void testRemoveTask() {

//create instance of Task and TaskService

Task test = new Task("UserdID", "Name", "Description");

TaskService test1 = new TaskService();

//add task to the array

test1.addTask(test);

//remove that task

test1.removeTask(test);

//assert the array contains 0 variables and check the array size

assertEquals(0, test1.tasks.size());

}

}

If the ArrayList size is 0, we know the removeTask function is working correctly; if it is anything other than 0, we know we have a problem. To check the add function, I did almost the same thing instead of the size being zero; the size would be one. In the end, technical soundness comes down to does this function does what I want it to do, and in this case, it does. When thinking about efficiency, I think about using less code to do more. Using the Junit functions assert can save us a lot of code that we otherwise would have to build out with if statements. One simple line of code can save us from having to write five or six lines of code. This might not be much now, but efficiency like this matters very much in a more enormous, more complex program.

In almost every test case that I wrote in this project, I used a form of Boundary Value Analysis. The basic idea is that if the value of a variable is within x and y, then that variable is valid. It is not valid if it is outside of that boundary value. Boundary Value Analysis is illustrated nicely in my testGetDescription test from my Task project.

@Test

void testGetDescription() {

Task test = new Task("UserID", "Name", "Description");

if(test.getDescription().length() <= 50 && test.getDescription() != null) {

}else {

fail("Test Failed");

}

}

}

The test is simple; if our Description is between 0(null) and 50, then the test is valid; if not, it is invalid. The other testing technique that I used was called Error Guessing. Essentially I use common sense to guess where the error might occur. I used Errog Guessing when I tested the add function in my Task class.

@Test

void testAddTask() {

//create instance of Task and TaskService

Task test = new Task("UserdID", "Name", "Description");

TaskService test1 = new TaskService();

//add task to the array

test1.addTask(test);

//assert 1, check the size of the array

assertEquals(1, test1.tasks.size());

}

I'm guessing that there will be an error when I try to add the task to the ArrayList. If there is an error, then the list's size will not be one. For the practical implications of both of these testing techniques, they do not test everything. With both, there could be problems that both tests do not cover. For example, we could guess the wrong kind of error with error testing and have the test pass, but an error still is present. So with both, a professional would have to think more broadly and do more extensive testing that doesn't just include Junit testing.

The mindset that I employed when doing this project was that of a pessimist. When I was writing each test case, I thought to myself, "there must be something wrong with this." I think this is the mindset that one ought to adopt when testing code. I was cautious when my code passed the Junit tests because even though the test has passed doesn't mean that the code is doing what I want it to. It is essential to respect the complexity of code because it can be affected in ways that you don't expect. For example, if I assume that my unique ID is added to the ArrayList correctly, I might miss that some complexity elsewhere caused the ID to be changed in some unexpected way. Perhaps causing it to move or change in some way. This is why it is vital of testers to understand the critical points of the code itself before they begin testing it.

I'm not sure if there is any way that I can limit my bias here, I'm too close to the code with not enough expertise. Ideally, the test code would be written without all the working knowledge of the internal code. This way the tester is not writing simple, easy test cases that align perfectly with the code that mine does here.

Discipline is essential in any field. For example, in my job now as a forklift driver, if I don't perform the job with discipline, I will get behind during the day and leave "technical debt" for the second shift crew. In software testing, it is the same way; lack of discipline will result in technical debt that others will have to pay for me. I will have to have discipline throughout my work to prevent technical debt. It starts from the beginning, using best practices when writing code, writing full notes that others can understand, and not cutting corners when I run into problems. At the end of the day, it comes down to this, if you're being paid to be a professional, you should act like a professional