Michael Abell

Project 2- 10/20/2024

CS-320-16226-M01 Software Test, Automation QA 2024

Instructor Parul Hirpara

I approached all three features in the same manner using the basic features of JUNIT testing. I used assertions of true or false. I accounted for each of the request of the requirements and tested that it would take appropriate arguments as well as reject and throw exception for invalid arguments.

As I have detailed in previous responses, there is no excuse or justification for not following requirements. They are integral to the SDLC. The best analogy that occurs to me is for someone to ask me to build a structure. I spend my time and effort creating and testing and perfecting a tree house. I present it to the requestor, and they are mortified. They needed a bridge to cross a river. With requirements I would not have built a treehouse, nor would I try and rework the treehouse to just make it work. I’ve failed and my reputation and finances will suffer. Additionally, no one will be crossing that river. My requirements were straight forward and unambiguous for this assignment. There were three services that had exact functions requested and so the service was built to these as well as the testing was set to ensure they worked appropriately. One example of requirements was: “The task object shall have a required unique task ID String that cannot be longer than 10 characters. The task ID shall not be null and shall not be updatable.” In my code I followed these requirements as represented by the following:

private final String taskId;

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

throw new IllegalArgumentException("Invalid ID");}

The final variable structure ensures it cannot be updated, while the if statement takes care of the taskID not being null or a length greater than 10 characters.

There is not a lot of room to debate the effectiveness. The Junit test outputs the exact percentage of coverage based on the code you input. I’m nota world class coder yet, so there is some trust to have in the test, but I do know that if I missed testing a feature, the coverage would let me know and highlight that call. My personal attestation is that the coverage testing gets the job done.

Ensuring technically sound code was accomplished by utilizing the features of error detecting that comes with the Eclipse IDE. If classes or variables are not properly set up there is a helpful message that appears in the form of a red underline with suggestions. This was particularly helpful when having to incorporate the java.util.Date for the appointment service. To ensure that my Junit test would go through without extransous disruption I added in “@SuppressWarnings("deprecation")” which made sure I didn’t get warnings or errors associated with having to use an old library.

There are times when being verbose can be beneficial, to add clarity to programmers that come after you who have to maintain the code. Then there are times where code needs to be simple and concise so that constructors or setters or getters aren’t eating up dozens of lines of code. The following code could have been expounded to include the class and take up double the lines, or we can just say “this” to let us know it’s part of the Task class we are in:

this.taskId = taskId;

this.Name = Name;

this.Description = Description;

Some of the techniques that I employed were positive and negative testing. I knew what the code needed to do in a “happy path” sense, so I utilized Junit testing and asserting all the correct values. In a sense this is white box testing since I made the code and know what to expect. I then add on to the assertions by intentionally creating incorrect values to assert, and I want those to equate to false, to prove that the code actually fails when it should and throws an error. These are reasonable baseline test.

@Test

void testAddTask() {

Task task = new Task("1234567890","Send Email","Send email at 0900");

Task task2 = new Task("1234567891","Send Fax","Send fax at 0900");

Task task3 = new Task("1234567892","Send PO","Send PO at 0900");

TaskService service = new TaskService();

assertEquals(true, service.addTask(task));

assertEquals(true, service.addTask(task2));

// this is set to false to prove you can't add the same ID again

assertEquals(false, service.addTask(task2));

assertEquals(true, service.addTask(task3));

The assert trues are examples of positive testing. The assert false is checking for failures.

There was a resource that shows us how to integrate the testing within Maven. Maven testing is still new to me so I have shied away from using this method, but that would add an additional layer of dependency checks for outdated API’s or drivers. This increase in testing would only serve to make the application a more secure product and eliminate down time from finding out about depreciated assets later on in the SDLC.

The practical use of Junit testing and assertions would be to set the process up on an automated script. You keep running these test as new classes or extensions are added to the project to ensure that the baseline functionality is not altered. As soon you get failures on the Junit testing you audit the newest change, roll it back and refactor the code so it does not fail your testing.

Keeping in mind, caution, bias and discipline, the code was built from the ground up. This was one of the first coding assignments that wasn’t just augmenting existing code to debug and correct it that I’ve worked on. I was purposeful in following the requirements while trying to be mindful of limits or exploits that bad variable construction can create. I was probably a little biased towards the code just working and assuming that trusted data would be input. There was not much input validation or protections against SQL injection, so if I was doing this assignment real world, then I would be more interested in securing the code.