CS-320 Project Two

Rob Marlatt

CS 320: Software Test Automation and Quality Assurance

Dr. Kalysa Wilson

Southern New Hampshire University

June 18, 2023

While setting up my code for Project One, I wrote the code based on the rubric, making sure that each point was followed. Once I was writing the testing for the code, I wrote the testing code to ensure that each section of the testing requirements was tested so that it conformed to the rubric as well. For each of the Class requirements, the classes needed to have a unique identifier no longer than 10 digits and not null. The contact class also needed to have firstName, lastName, phone and address fields that were of a certain length and not null as well. The task class needed to have name and description fields that were of a certain length and not null. And the appointment class needed to have a date in the future and a description of a certain length and not null. The services needed to be able to generate the unique identifiers, allow them to be deleted, and allow the class information to be updated. For example, for the contact class, I initialized my variables with the specific length needed for that field using a static final field. This was then used to ensure that the field, once information was written to it, met the requirements for that field. I also included exceptions in the functions that write the information in the fields to throw an error if the field is too long or null, or in the case of phoneNumber, was anything other than numbers. (Module Three Guidelines and Rubric, n.d.) (Module Four Guidelines and Rubric, n.d.) (Module Five Guidelines and Rubric, n.d.).

I tested my classes for complete coverage using the JUnit tests with the testing requirements that would ensure that the test would pass with the correct information, and also set up false testing for a field too large, and a null field. The services tests were set up to ensure that the service would add information, delete the information, and be able to update the information with changes. By testing all these situations, I was able to ensure complete coverage of the requirements listed in the rubric. This ensured that I met the minimum of 80 percent coverage for my testing.

I was able to ensure that my code was technically sound by writing functions that used assertions to check that the field met the requirements of the rubric. For example. when writing the JUnit test code for the contact service, I would have the function create a new instance, then assert that the code met the requirements by adding correct and then incorrect information to the contact fields to test whether the exceptions in the service class would be thrown. See below for an example of code that checks to ensure that the new contactId would initialize to the INITIAL setup:

assertAll(

"service",

()

-> assertNotNull(service.getContactList().get(0).getContactId()),

()

-> assertEquals("INITIAL",

I was able to ensure that the code was efficient by setting up a template for how the code should be written, including with the JUnit tests, and then followed the format throughout the code. I also wrote the code so that the tests for each field were contained in just one function, instead of multiple functions testing the same field. See below for an example of code that I wrote to test the contact service for the firstName field:

service.getContactList().get(0).getAddress()));

service.newContact(firstNameTest);

assertAll(

"service",

()

-> assertNotNull(service.getContactList().get(1).getContactId()),

()

-> assertEquals(firstNameTest,

service.getContactList().get(1).getFirstName()),

()

-> assertEquals("INITIAL",

service.getContactList().get(1).getLastName()),

()

-> assertEquals("1234567890",

service.getContactList().get(1).getPhoneNumber()),

()

-> assertEquals("INITIAL",

For each of the project's milestones, I tested using white box testing methods. White box testing requires that the tester be familiar with the software's operation. This was the best option because I created the program. For instance, I checked the appointment ID to make sure it couldn't be altered and wasn't null or more than 10 characters. I ran tests on the Date field to make sure the date was present and not in the past, correctly formatted, and not null. I also verified that the description was valid and was no more than 50 characters. To ensure that I could add and cancel appointments with the correct appointment ID, I tested the service class. I utilized unit testing while I was creating the code to make sure every part was functional before adding it to the program. There are a few additional methods I didn't employ. Black box testing, where the tester does not know what the software does before testing, Integration testing because we do not have the entire program to test it with, Acceptance testing, and Grey box testing because we do not have a finished project, and Manual and Static testing because they were not appropriate given that Manual testing requires manually entering an input and Static testing does not execute any code.

I made sure to use caution when I was writing my test code by looking at how each class was set up. I checked how the unique ID for each class related to the fields contained in the class, and how each field could possibly be affected by any other field. When I wrote the tests, I made sure to test each field individually, then tested them as a whole, to make sure that there was no interference once all the fields were combined. If the interrelationship of each field in the class is not examined, an error in one field could cause the entire class to fail, instead of just the one field failing. This is how you ensure that your testing has complete coverage for any issue, not just for surface issues.

I tried to limit bias in my testing by writing the test code to specifically test the standards set out in the rubric. I could look at my code and know that the code followed the rubric, but since I wrote it, I have a comfort level with the code that could cause me to overlook an error that the test could easily find. Sometimes the error is obvious to see to someone else, but we cannot see it because we are too familiar with it.

As a developer, it is of great importance to be disciplined when coding and writing your test code. If you are lax on your quality, your work and errors could cause an entire system to crash and cause multiple problems. Without quality testing, a small error, such as with our contact ID, could cause multiple contacts to be overwritten with the same ID, losing information or even leaking information to someone that should not have access to it. A data breach could cause serious repercussions, if it is personal information, or it is something that is secret or confidential, such as government work or HIPAA information. Writing efficient code that is well documented is also critical. If someone else has to modify or update your code, they could have a hard time doing so if the code is not clearly written.

**References**

(n.d.-a). Retrieved June 18, 2023, from <https://learn.snhu.edu/d2l/le/content/1311909/viewContent/24289747/View>

(n.d.-b). Retrieved June 18, 2023, from <https://learn.snhu.edu/d2l/le/content/1311909/viewContent/24289750/View>

(n.d.-c). Retrieved June 18, 2023, from <https://learn.snhu.edu/d2l/le/content/1311909/viewContent/24289755/View>