Marysha Brown

CS-320 Software Test Automation & QA

12/7/2022

Project 2

# Summary

The software requirements for the Task Class wanted the string task ID to be unique, so I created a test in TaskServiceTest.java that would test if the program would accept a new task with the same task ID as a task already in the system. The Task Class requirements wanted a task object with a task ID of fewer than 11 characters and not null; this was tested in the TaskTest.java file to check if the program would accept a task ID over ten characters or a null ID. The Task Class requirements also wanted the string task name that was not null and to be less than 21 characters. I added a test in the TaskTest.java file to test if the program would accept a task name longer than 20 characters or a null name. Finally, the Task Class requirements wanted a task object with a string description of fewer than 51 characters and not null; this was also tested in the TaskTest.java file to check if the program would accept a task description over 50 characters or a null description.

My tests met all of the requirements from the instructions. My coverage for TaskTest.java was 84.8% for the Test.java file. My coverage for the TaskServiceTest.java was 83.7% for the Test Service Class. I have written other catches and error handling in my TaskService.java code that was not required to test. All of the code that was required was tested and passed.

In the TaskService.java file, on lines 13-20, there is error handling if the user tries to input a task with the same ID as another task already in the system. This is a creative way of checking this requirement and handling the error of redundant task IDs since it will prompt the user that there is already a task with the same ID and they need to create a unique one. In the Task.java file, in the Constructor, starting on line 9, I created checks that would throw errors if the task ID, name, or description was not in line with the program's requirements.

My code is efficient because there are not many lines of extra code that do not directly relate to the program's requirements. Many of these lines were not tested, but they were important to the program's framework, such as the setter methods on lines 43-62. I also wrote one test per requirement to ensure that all of the program's tests were relevant to the requirements for the program's functionality (Brown, Module 4 Journal, 2022).

# Reflection

The requirements that the client specified were fairly simple to implement. The client wanted a mobile application to create a contact, a task, and an appointment. Each of these objects has attributes to store information about each object. The requirements specified some of the parameters for the attributes, such as the contact ID must not be longer than ten characters and cannot be null. I focused my testing efforts on performing Junit tests, a type of black-box functional testing (GeeksforGeeks, 2021). These tests were to make sure that each attribute would abide by the parameters set by the requirements.

I did not perform any manual testing on the code. The files that we were tasked with making were not intended to run, and so I did not create the main class to run the program. The current requirements did not specify that we needed to format any UI or UX design for the application, which would have prompted the creation of the main class. I expect to use manual testing more in the coming weeks when we put all of the milestones together.

Junit testing is a framework for Java that helps create automated tests. Junit testing is beneficial because you do not need to interfere with your source code to test it and break anything by accident. Junit testing also provides coverage data, so you understand what parts of our code are being tested and what is being left out. A requirement for testing may be that the coverage for tests needs to be at least 80%, so JUnit testing makes this easy to calculate. Manual testing does not use any automated tools to test the program. The test engineer tests the product manually while viewing it from the customer's perspective to create the necessary test cases and provide the developers with accurate feedback. Manual testing allows software testers to access visual elements, including layout, text, and other components. They can spot UX and UI problems as well (Brown, Module 5 Journal, 2022).

I'd say that I used caution to a significant level when developing my programs and tests. To ensure that every file had the greatest amount of coverage, I thoroughly reviewed the tests. Because methods may function well when called individually but may be invoked by other methods that result in unexpected behavior, relationships can be crucial when building tests. On line 17 of my TaskTest class, I created a test for the class constructor to make sure that when the program would try and create an instance of that class, it would initially work properly and provide a lot of test coverage. Making sure the fundamental relationships of the class methods interact as expected is crucial to building a working complex program.

All programs can be effected by bias. What we expect results to be is due to bias. Of course, testing code also has the potential to cause this. By strictly following the client's specifications, I attempted to limit this in my own testing. A bias that is written into the requirements is that the names cannot be longer than 10 characters. On lines 16-22 of the Contact class, I wrote in code to handle these exceptions. This requirement is written with the bias that the names entered into the program will only be a single first and last name with less than 10 characters. If you have a hyphenated last name or two first names, you'd have to alter your name, which is unfortunate and frustrating because you run into this often (Powers, 2018).

Any developer must maintain discipline and a commitment to code quality throughout their career. Coding shortcuts can result in lost time, lost money, and in some cases, even lost lives like we are learning in our module discussion. The approach to creating methods and objects will change to be more thorough, if you make a commitment to quality at the beginning of every project. Additionally, the hope is that with experience, you'll test the software with any conceivable unforeseen circumstances that might arise in the actual world. By systematically designing my code before writing it and testing it after it is written all the way until it is functional. I intend to avoid accumulating technical debt when producing code in my career by tyring to think ahead and staying organized since a lack of planning and inadequate documentation can lead to technical debt (Weber, 2022).

When writing my Appointment class code in the first milestone, I tried to make sure all of my logic was sound and that the code fulfilled all of the requirements. I was able to re-use the code for the following milestones to help speed up my completion times. I was also able to give everything a second look and changed a few naming issues or comments before submitting my final code project. I think it is very important to comment code and try to write simple comments above each code block to let readers quickly understand what is happening and follow my logic flow more easily.

# References

Brown, M. (2022). *Module 4 Journal* [Unpublished paper]. Southern New Hampshire University.

Brown, M. (2022). *Module 5 Journal* [Unpublished paper]. Southern New Hampshire University.

GeeksforGeeks. (2021, 03 01). *Software Testing Techniques*. Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/software-testing-techniques/

Powers, C. (2018, 06 11). *Why You Should Stop Asking for First and Last Names on Forms*. Retrieved from Software For Good: https://softwareforgood.com/why-you-should-stop-asking-for-first-and-last-names-on-forms/

Weber, J. L. (2022, 04 14). *Technical Debt: Definition, Types & Examples*. Retrieved from ProjectManager: https://www.projectmanager.com/blog/what-is-technical-debt