Ariel Foor

Professor Rob Tuft

CS 320

August 18, 2024

7.2 Project Two

For the unit testing of the service portions of the assignments, I made sure that I tested that it would update the contact or task and that it would also throw an exception if they did not work. The code had a loop to check for the item so that it was unique and then would throw an exception if not. This occurred on all portions of the assignments including when it was updating a task or checking the date on an appointment.

A computer code with black text

Description automatically generated

While some unit tests may have been excessive, all the test cases made it to 100% coverage. The code was technically sound by ensuring that there were no errors present and that there were also no warnings on any of the text. As well it provides errors during running if there is an issue. The code is efficient because it tests one section at a time and with precision to make sure that there is not gaps, but they run quickly to get through all the items.

A screenshot of a computer

Description automatically generated

The tests I used in the milestones were unit tests and functional tests. The unit tests included testing if things were too long, null, or before the current date. I also used exceptions for items outside the rules' bounds. For things that were too long we would create tests where if an id had too many characters, it would throw an exception. This was also the case for null, where we traded the value for null and tested each component in each class. Lastly, we used "before" for the date by asking the test to get the date I had assigned and checking it against the day's date. Since it would be in the past, it failed the before but passed the false test. The functional tests also checked if specific items were added to the list. In the addNewAppointment test we checked over the list to see if there was a specific id, from getId, added by doing a Boolean statement to return true if the id was on the list.

I also did not use integration and end-to-end tests in the testing portion of the milestones. Since the portions we had would require user input, we could not use integration to ensure that the components were working together. Also, with end-to-end tests, since this is an unfinished project, we don't know how it will look to the end user and, as such, could not test it as an end user. As well I had not used security testing, user acceptance testing or usability testing. Since there is no end user to test the programs, there is less security risk as well and the usability and user acceptance testing as these require the final finished project to be completed.

Unit tests allow us to test specific pieces of code, especially ones that have boundary statements, such as less than or not null. Those are great for when user input will be used. This works well with lists and games as input is needed to build around the project. However it can also be used to make sure that decision making is done correctly and follows a correct path. Unit tests would likely be useful on most projects. Functional tests are always necessary unless there are no requirements. However, since most projects have a list of requirements, making sure that it is meeting those requirements is an important and necessary step. Integration testing is valuable to projects with multiple parts. The integration will make sure that each component works together. Lastly end-to-end, is before it is released to end users to make sure that the project works together and to see how the end user would view it. This is vital to video games and making sure that menus load correctly and that specific options can be selected.

Working on this project helped me to exercise caution because we want to make sure that we are testing the right things. Since a lot of the test could be repeated with old code, it was necessary to ensure that all pieces of the old code were changed to reflect that which it was inputted. The code is complex and should be appreciated since we can have the code do so much. Such as when created an object or deleting an object is calls another class to find and delete it from an array.

A screen shot of a computer code

Description automatically generated

Being biased about ones code is normal but needs to be recognized to ensure that the code is still of great quality. Our bias can make it difficult to see how our code could be different or think there is nothing wrong with it. I had to fight my own bias since I couldn’t figure out why a specific test wasn’t working and that was because I was biased thinking my original code was perfect. I had to step away and review it discover I hadn’t changed the quantity for the number of characters for the address in the contact service. It taught me a lesson to be open minded and that it most likely is an easy fix.

We must stay vigilant and disciplined about our code as code quality and efficiency as this is the standard. We must hold high standards to keep quality high. Poorly running code is frustrating to users, buyers, and shareholders. As developers we should always be testing the code and ensuring that is meets requirements. There should be plans in place to ensure efficiency without cutting out necessary steps. By ensuring our code quality is high and that we are careful to meet requirements we will reduce our technical debt exponentially. When we are not stuck refactoring codes or chasing bugs we can spend more time on improving the code or adding additional features in the same amount of time. There was a lot of time spent making sure that each test worked, checking coverage and making sure that it met the marks for what it called for. I would run each test to make sure that it ran well and was efficient, then I would add in another test, especially in the appointment class. As soon as I had figured out the one part of the date I would move onto the next such as checking that the branch for the past went through and passed the exception.