Summary

Describe your unit testing approach for each of the three features.

My general strategy when creating unit tests for the software was to try to make it as simplistic code wise as possible, and as idiot proof as possible, so that there were less possible things that would even need tests to begin with. Letting the task, service, and contact managers deal with the input and ensuring that the base classes were only ever doing things with permission from the manager classes made this a much simpler process. First and foremost, it made it so that the base classes only needed to be tested for whether they can be created, and returned with the proper information. This is easy enough, a couple return statements and a constructor check. As for the other tests, it came down to realizing all those possibilities that still could go wrong, and lining the tests up for each of them.

To what extent was your approach aligned to the software requirements? Support your claims with specific evidence.

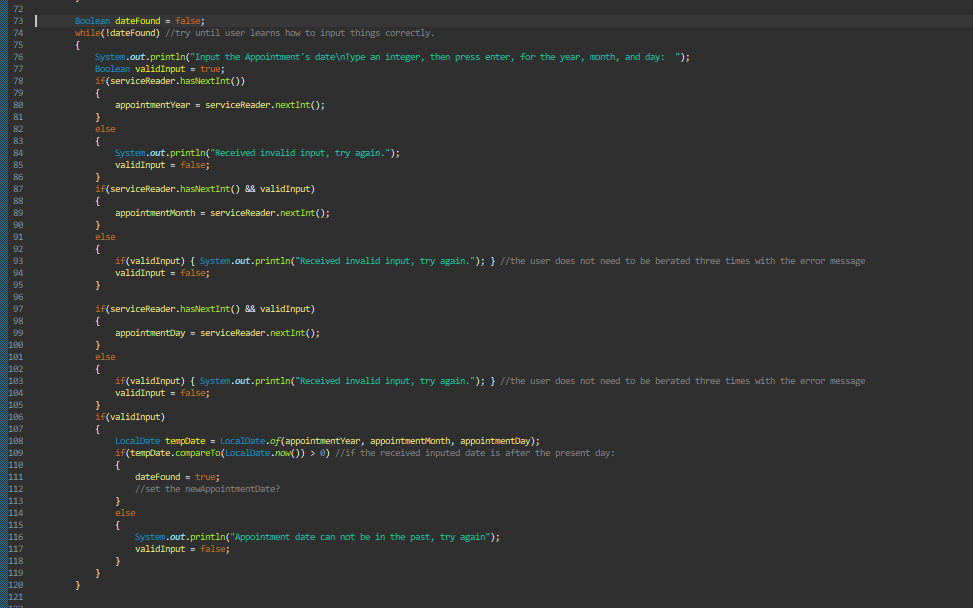
The tests that I did boiled down almost exclusively to the requirements- I had a few extras, such as testing returns, constructors, whether the objects were null, and whether the order mattered, but otherwise, every test was written with the express purpose to fulfill some aspect of the program requirements.

Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were effective based on the coverage percentage?

I thought of all sorts of different inputs- the ones that didn’t work and the ones that did. As for the bad inputs, I made it so the user couldn't really proceed or exit the program without a good one. And so the tests came pretty easily- if ever there was a bad input, you just needed the test to retry with a good one to make sure that it worked. I tried pretty much every combination that I can think of in order to ensure that they did, and as it turns out, there are only so many ways you can go way with a project this scope and size.

Describe your experience writing the JUnit tests.

At first I was a little confused, so I looked up the documentation for JUnit, and then it immediately clicked. All that I was missing was the @Test at the beginning of the test, and then afterwards it was just like the tests I’d otherwise write in-line with the code manually. That is to say, the testing process was pretty seamless from what I am used to.

How did you ensure that your code was technically sound? Cite specific lines of code from your tests to illustrate. Perhaps the most technical part of the code was the appointment service manager. Ensuring that it was receiving valid input requires quite a few different checks. And then it also has to interface with the year, month, day format of the dates for the appointment. The only way that it could have been more complex is if it somehow involved the string limitslike the other classes in this instance. Ensuring that it was technically sound, however, more or less came down to copying the previous work from previously tested classes, and building upon that solid foundation. That is to say, you work your way into it, and check each step of the way for new potential bugs, and squash them as they come, never letting them build up.

How did you ensure that your code was efficient? Cite specific lines of code from your tests to illustrate.

The same can be said for the efficiency- write the most straightforward, and most easy for the computer to understand code, avoid loops where possible, (except when the user could theoretically screw things up, because they probably will), and build upon the strong foundation. The culmination of this efficiency is again in the above image of code.

Reflection

Testing Techniques

What were the software testing techniques that you employed in this project? Describe their characteristics using specific details.

The majority of tests I did was manual end to end tests. I worked through the creation of objects, then how they interact with one another, and how they hold up when testing various combinations and varieties of inputs. I’ve done some research and see that a “smoke test,” might be the one I did the most- I wrote the majority of the code in the first weeks for the first class objects, and recycled it and reoriented the code to accommodate the new classes each week, and ran through them all over again.

What are the other software testing techniques that you did not use for this project? Describe their characteristics using specific details.

The tests I did not do are performance related- the program is small, the scope is small, there is little chance of leakage; The one thing I could have tested for was the scanner class, though I also didn’t do integration tests, to see whether the scanner even has issues with the other implemented libraries. This would be more of a research problem though, rather than testing- as the software works perfectly as intended from the looks of things.

Mindset

Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ caution? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.

I don’t think I employed “little” caution, but a lot of curiosity, and perhaps a very inquisitive mindset more so, for sure. It’s definitely a learning experience- though I think that for this project, the tests might have been more difficult to write than the code, had I not designed the software from the ground up to be as robust and straightforward as possible. Perhaps that is the mindset change I need to make, generally- designing software from the start to be as user-error-immune as possible.

Assess the ways you tried to limit bias in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims.

Similar to my other answers, the general idea was to exhaust every possibility. There might be bias, but if there is, then it’s for sure not gonna be relevant- mathematically calculating all the possible inputs and ways that something cn go wrong, and then accounting for each of those possibilities tends to ensure that the marginal change in code doesn’t matter.

Finally, evaluate the importance of being disciplined in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.

Being disciplined in code writing is how I have always approached things- though perhaps not often to the extent that I have had to write these sorts of tests. I generally have wrote software for my own purposes, where I know exactly how things work, so cutting corners is a possibility that I am aware of, but not ever seeking out. I think that the simple act of having to write tests ensures that you never cross over that line- you know you’ll have to fix it later and make more work for yourself. That’s honestly the biggest takeaway, if you know you’re gonna write a bunch of tests, then you know you’re gonna have to fix the errors anyways, so you write better as a result.