Dominick Hernandez

Prof. Toledo

CS 320

8/12/23

Project Two

* 1. **Summary**
  2. Describe your unit testing approach for each of the three features.
  3. To what extent was your approach **aligned to the software requirements**? Support your claims with specific evidence.

One of the main ways I decided what to test was to take the requirements and test for that. Like for example one of the requirements for the task class 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.”. So, I took that and tested it in code. I wrote tests to make sure the id cannot be set to a string longer than 10 characters. Then a test to make sure it cannot be null. I did not implement a way to update the id, so this meant that there is no way to update it. Doing this for each class, I worked out tests that would test most of the requirements for each class except NOT having a feature.

* 1. 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?

Overall, in the end I tested every requirement from both sides. For example, if a requirement was for the name to be less than 20 characters but not null, then I tested if it was null if it was above 20. The only exception is that I did not test for the requirements of a feature not being implemented. I know I could have used an assert throws unsupported operation exception test, but the problem there is that I feel I would have had to test for every name for a method that would do that. So, in the end I just did not add one.

* 1. Describe your experience writing the JUnit tests.

Honestly, I really liked making these tests. It really opened my eyes in how I sometimes read requirements and implement them wrong. Rationalizing how exactly to test for a needed feature helped me think of how to implement the features.

* 1. How did you ensure that your code was **technically sound**? Cite specific lines of code from your tests to illustrate.

I feel like I had put extra thought into testing the delete task test. I had it add a task and then delete it, I could have easily just tested to see if it was now null, but instead I added a second task, then deleted the first. This let me check to see if the right task was deleted. I also at the end test to see if I can delete the same task again.

@Test

void deleteTaskTest() {

taskService = new TaskService();

String taskID = taskService.addTask(taskId, nameTest, descriptionTest);

String secondTaskID = taskService.addTask(taskId2, nameTest, descriptionTest);

taskService.deleteTask(taskID);

assertEquals(null, taskService.taskSearch(taskID));

taskService.deleteTask(secondTaskID);

assertThrows(IllegalArgumentException.class,() -> taskService.taskSearch(secondTaskID));

}

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

For my code I used a lot of modular programming to make sure I can test for things from the bottom up. So instead of writing new code for updating a task name in task service I just used the code in the task object. This means I can test task service just to make sure it is changing the right object and do not need to test if it does not accept null and too long values for name because those are already tested for in the task object tester.

* 1. **Reflection**
  2. Testing Techniques
  3. What were the **software testing techniques** that you employed in this project? Describe their characteristics using specific details.

White box testing is the main testing technique I was using. Since we have only developed back-end code for the appointment class, we needed to test it with JUnit tests. Doing this we can test the actual code and make sure each part of code works. We also did integration testing; this testing focuses on making sure various parts of an application work together. In our programming we tested if the appointment class works with the appointment service class. And lastly, I did a bit of regression testing, which involves setting a list of tests and testing code as you change it to make sure it still works correctly. I did this as I tried to refactor my code.

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

We did not do black box testing; this is because we have not set up a way to test this code while it is running. There is no main application we can test this code in, so there is no way to test this code with a user’s inputs and outputs.

* 1. For each of the techniques you discussed, explain the **practical uses and implications** for different software development projects and situations.

White box testing is one of the most important for the beginning phases of software development. As this makes sure each individual piece of code works as is intended and helps lower the amount of time spent doing higher level testing. Black box testing is important to make sure the users experience is presented as intended. As the user does not need to know how the code of the program works, this testing is done using the same point of view and tests the functionality and ensures the code fulfils the purpose it was built for. Integration testing is especially important for any company building a large program with many distinct parts. As this testing focusses on how each part of the program works alone and as a whole, this type of testing can be seen as a middle ground between white box and black box testing. Lastly regression testing is done throughout the entire software development life cycle. This testing is done to make sure no changes to the code break other parts of the program that are already finished. These tests can be especially helpful when bug fixing or refactoring code throughout the SDLC.

* 1. Mindset
  2. 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.

The main mindset changes from normal coding to coding tests was in thinking what I did and did not want the code to do. When coding you want to code what you want the program to do, for example, if you have a dog class, and want to add a method to give them names, you will code that specifically. But for a tester you would think of what the user might do, and how they might interact with that method. What would you NOT want them to be able to do? Would you want them to be able to type in a 300-page essay as a name? Would you want them to be able to leave it nameless? What about non number/alphabetical characters? I could have tested for a million different things, and in doing so written tons of different tests. But for here I just made sure to test for things I felt were obvious, like for example, making sure the delete function of task service deletes the right task.

* 1. 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.

For one I tried thinking and pre-writing dummy code tests before writing the code. This helped in not only writing the tests, but writing the code like I was given a list of tests by someone else. If I were responsible for writing the tests for my own code consistently, I feel I might eventually leave some things out. This is simply because with only one viewpoint I might leave out some things that might be useful to test for. In a business setting this would be counteracted by having another tester proofread my tests for quality.

* 1. 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.

I feel I have come a long way from when I first started writing tests. My first line of tests was extremely generic and did not test for a lot of different possibilities. For example, in my first set of tests testing the contact class, I tested every setter in one big test. In the beginning in small programs this might be okay, but eventually this would cause terrible setbacks in developing software. As when it comes to a big multiclass program, if you test in wide sweeps like this, you might as well not even be doing unit testing at all. The point of unit testing is to be able to pinpoint any errors in the code. Having all your tests tell you there might be an error in these 10 separate parts of code is helpful, but nowhere near as helpful as if it told you specifically which part. In the future I plan to avoid these problems by making sure I test for requirements individually and collectively when necessary.