Joseph Kawamoto

CS-320 23EW2

Professor Tanisha Jacks

10 December 2023

Project 2 Final Paper

During unit testing, the approach I took was to ensure that each of the required functionalities were met and that any input stipulations (for example character count) were followed. For each of the three milestones, there was an input limit for certain variables. To make sure that my code adhered to these guidelines, I set up tests that would check for the number of characters being input and return failures if the input did not adhere to the character limits. For example, in the contact class of from milestone 3, exactly 10 characters needed to be input for the phone number variable. Any more or less than that and the input would not be accepted or the JUnit test would fail. These types of characters limits are to stop the wrong data from being input, but also to protect the system from malicious actors.

I ensured that my code was technically sound by ensuring that it met all the requirements necessary for the assignment. For each class that I wrote, I would go through each of the steps one-by-one to ensure that I was correctly implementing the functionality that was required by the assignment. During the testing phase of writing my code, I tried to implement tests that could verify the technical soundness of the system based on the different class-specific requirements. For example, in module three, the ContactTest module featured tests that I wrote to validate each criterion of the Contact class. One of the specific conditions was that the unique ContactID did not exceed 10 characters.

*if (contact.getContactID().length() > 10) {*

*fail (“Contact ID has more than 10 characters.”);*

*}*

I used an identical statement to ensure that the first name variable did not exceed ten characters either. Looking back at it now, what I should have done is test the fringe examples, like when there are exactly ten characters. This would have increased the coverage of my tests and gave me more confidence that the code was written correctly. I also tried to keep my code as concise as possible when writing it, I figured that this would increase the readability of my code which would make it easier for me to come back to and work on at a later date if necessary. My in-line comments could definitely use some improvement however. Looking at it now, they are very sparse and don’t give quite as detailed an explanation of what the code does as I would have liked.

While I was writing my code, first I would go through all of the requirements and write my code to ensure that it followed them. Then when it came time to test, I would try to find errors in the code and try to show that it didn’t work. Rather than writing the tests to show that the code worked, I tried to write tests to prove that it didn’t. By doing this, I found it easier to come up with tests.

The main software testing technique that I used for this project was unit testing. Unit testing is where you test the individual components of the code to ensure that they all work in isolation, before you integrate them together. This was accomplished by using JUnit testing to check each of my methods in the classes that I created for each week’s milestone. Unit testing is helpful because it breaks what can be a very complicated program into more bite-sized, digestible chunks. It is much easier to come up with tests for an individual method than it is to create tests for an entire system.

@Test

@DisplayName("Task ID cannot have more than 10 characters")

void testTaskIDWithMoreThanTenCharacters() {

Task task = new Task("Name", "Description");

if (task.getTaskID().length() > 10) {

fail("Task ID has more than 10 characters.");

}

This code above is a test that I wrote to ensure that the unique task ID that was given to each new task object did not exceed 10 characters. This is a good example of unit testing I believe because it highlights that each individual test only checks for a single functionality. What I should have done to improve this and add coverage to my tests is to check what happens when the ID has less than 10 characters, and write another test to check what happens when the ID has exactly 10 characters. This functionality should have been given three different tests to prove that it was functioning the right way. Unit testing is perfect for testing projects that are modular or have individual parts that need to be tested in isolation or for projects that are using agile sprints. Unit testing can also be used to boost the confidents in the reliability of each unit of code before they are all integrated together.

The main software testing technique that I omitted (but wish I hadn’t now that I’m thinking back on it) was boundary testing. This form of testing checks how software behaves when you get near the limit of acceptable input. Like I stated in the paragraph above, I should have checked what happens when the character count for the unique task ID was exactly 9, 10, or 11. This would have given me more confidence that the code was working as intended. Boundary testing is important for projects where user inputs are taken especially if the inputs or length of the inputs might vary. They can also help you uncover possible security issues. Another form of testing that I didn’t really use for these modules is integration testing. The reason for this is that each of the individual methods I created didn’t rely on each other, they worked in isolation. The only functionality that required the other tests to work was the deletion or removal methods form the contact, task, and appointment classes. The deletion and removal methods rely on their actually being a contact, task, or appointment object already created for them to work because other wise there is nothing to remove/delete. Integration testing checks how different code components interact with one another and ensures that they work together as intended. Integration testing is perfect for ensuring that the system works together overall.

When writing my code, I tried to be as meticulous as possible. First, I would write a few lines and then re-read them to make sure that I hadn’t left anything out and that everything I was writing was following the assignment guidelines for system functionality. I would code deliberately and slowly, ensuring that I didn’t forget anything. Once the code was fully written, and before I did any unit testing, I would give my code a thorough once over visually to make sure that it followed the assignment functionality requirements. To accomplish this, I would read through each method and each class and make sure that every variable was accounted for and every if statement made sense. More often than not, I would catch a few “else” statements that I had repeated or had omitted entirely. Once I felt that my code had passed my visual inspection, I would move on to testing it.

My mindset as a software tester was to assume that I had done something wrong when I was coding. I would try to find test that would prove that I hadn’t done the assignment correctly. As a software tester I paid very close attention to detail to ensure the reliability of the code I had written. To limit bias in my code review, I aimed to be as objective as possible, focusing more on adhering to the requirements than to proving that the code I had written was correct. I can definitely see why bias could be a huge limiting factor, because nobody wants to admit that the code they just spend hours writing is flawed in any way. During module 3, I kept getting errors saying that my unit tests were failing. Which meant that either my code was written incorrectly, or my tests were broken. Neither of which felt good because I had spent so much time trying to get them both right.

Being disciplined and committed to coding and testing quality as a professional is an important part of ensuring long-term professional success. It will be hard to hold a job if you are constantly turning in broken tests or faulty code. That’s why it is so important to take writing your code seriously and to work diligently to ensure that your tests are coving as much as possible.

References