7-2 Project Two

Dane Clark

Department of Computer Science SNHU

CS-320 Software Test Automation & QA

Linda Gaston

08/09/2022

My approach to verifying that each requirement was met within my code was making sure the main areas actually worked as intended. That is ensuring that the code accepted valid inputs as well as throwing error messages when an input was outside the parameters. I did this using the “acceptTrue” test method. This test is used to show that a statement I input is true. For example,

@Test

**void** testAppointmentsClass() {

Appointments newApts = **new** Appointments("1111111111", date , "Vet Appointment");

assertTrue(newApts.getApptId().equals("1111111111"));

assertTrue(newApts.getApptDate().equals(date));

assertTrue(newApts.getApptDiscript().equals("Vet Appointment"));.

We can see from this section of code that we are testing the creation of an appointment using the AppointmentClass. First, we set the parameters with the newApts, then we check using the asserTrue method to ensure that what we set actually matches what is saved as a new Appointment. The reason for using this method is to make sure that the appointment creation is made. If everything is working as intended each parameter is stored within the appointment array as a new appointment. The real purpose of this method is to see if we get errors that should otherwise be true. Overall, the JUnit testing used throughout project one and the previous modules could have covered more of the code percentage. I feel that the more code coverage there is the more accurate the results will be. In addition, having a higher percentage of code coverage helps to ensure that there is not a lot of redundant code. Or rather code that is not needed to complete a task. This could mean cutting down some of the code to be more streamlined or adding more tests to test areas of the code that aren’t necessarily part of the functionality. For example, testing that a print statement is actually printed. Making the focus of our code the overall functionality will lead to higher test coverage.

Writing code can get very tedious. Syntax is a big issue especially when switching between coding languages. To ensure that my code was technically sound I often made sure to test code as I completed tasks. This would allow me to ensure that these areas were working as intended before moving onto the next requirement. If I left it all until the end, I could end up having to troubleshoot a large amount of code rather than a small amount at a time. This style of writing code may take a bit longer, however, I feel that it does lead to few issues in the long run. Additionally, building in outputs to the console was another way to ensure that the code was doing what I wanted it to do.

@Test

**void** testUpdateName() {

**int** size = TaskServices.tasks.size();

System.out.println(TaskServices.tasks.get(size - 1).getTaskId());

System.out.println(TaskServices.tasks.get(size - 1).getTaskName());

TaskServices.updateTaskName("1000000004", "Task Name Goes Here");

System.out.println(TaskServices.tasks.get(size - 1).getTaskName());

assertTrue(TaskServices.tasks.get(size - 1).getTaskName().equals("Task Name Goes Here"));

}

Above I use a print method to make sure that the update is actually happening within my test. This was to give me a visual of what the code is doing rather than just relying on the JUnit test saying that it did. The next the to focus on was efficiency. I mentioned before that it is easy to add in redundant code that my not get covered effectively using the JUnit testing. For me the best way to ensure that the code was efficient was to make sure that the requirements were met effectively. This would leave little room for redundant code since the task would be complete. Using comments within my code also helped me to keep track of what was already done and how it was being executed. When using comments, I was able to cut down how much code was actually needed to complete each requirement. Using the least amount of code to complete a task isn’t something I try to do, however. I often want to make sure that the code is executing the task properly first before looking for solutions to minimize the amount of code used. This can sometimes lead to something more efficient, while other times there may not be a better solution. My goal was to ensure that the requirements were met effectively. Using JUnit testing helped to show how efficient the code is. When looking at the coverage of the tests you get a better idea of how much extra code there is. This is not cure all as some of that could mean you haven’t tested for certain areas of the code.

Dynamic testing was the preferred method of testing I used throughout the project. The reason for this was to ensure the functionality of the code that was written. I wanted to be sure that the requirements were met and were functioning as I intended them to. A good example of this is when I tested to ensure that a task was deleted.

**void** testDeleteTask() {

TaskServices.*addTask*("Some Task Name", "Some Task Description");

TaskServices.*deleteTask*("1000000003");

TaskServices.*searchTask*("1000000003");

*assertTrue*(TaskServices.*searchTask*("1000000003") == 2);.

Here we see that we used the assertTrue method to show that the task had been deleted. This showed me that the code was performing its function. That is to delete a task given its id number. With in the code there was a search method to find tasks based on their id number, we use it here to see if the id number can be found after using the deleteTask method. This type of dynamic testing helped to fix areas that were not working properly and even more, it allowed me to find out why the code was not working. Often times it was a syntax error, but other times it was something a bit more involved. Such as something not being declared earlier in the code. Static testing or static analysis was an area where I did not focus one too much. A lot of what encompasses static testing is more geared toward code review. Though I did do this to some extent it was not as robust as it would be in a live environment. A team would typically review code looking for errors in syntax and overall code writing. Most of the way I was able to find those error was directly through the use of dynamic testing methods.

Both types of testing methods are essential for a well written code that performs properly. We want a second set of eyes looking over our code and the way it is written. Static testing is great for finding small errors that the originator of the code might have missed. The use of code walkthroughs helps the developers talk out what their code is doing to someone else. This is akin to reading a written paper out loud. When you hear it said out loud you get a better idea of how it sounds. Often this can point out errors or redundancies in the way the code is written. Dynamic testing is running the code in a test environment to ensure that the code is performing the way it should. In other words, does the code work? Dynamic testing is important in ensuring that we deliver a product that addresses the requirements asked and actually executes those requirements. Both are needed to ensure that we have the most effective code we could deliver.

I tried to focus on making sure the code I wrote effectively addressed the requirements asked for by the client. Ultimately, for me I want to make sure that I deliver a product that meets the needs of the client. I did apply caution in my work when writing the best way to handle the situation. When I come to testing, I wanted to make sure I tested the core functions of the program. That is making sure the parameters that were set worked properly and if they didn’t find ways to remedy those problems. As mentioned previously, I also made sure to not write too much code at one time before testing. I tested small sections of code then fixed what needed to be fixed before moving on to another area. It became apparent how important the complexity of the code became. For example, creating a unique id that could not be changed, then using that id to find an appointment. It is important to first test that the unique id is created. Then testing to ensure that it will always be unique. Doing this is a major component of the project. Without that the rest of the functions do not work. Since we need a unique id to create a new appointment or task, that is paramount for the overall functionality of the program. Likewise, without the unique id I would not be able to find an appointment to make changes. This interrelationship of the code is important to test because, while some parts may work as a stand-alone function, the overall task of the code may not work together. This was something I really took notice of while formulating my tests.

Limiting bias was admittedly a challenge. Since I am the originator of the code and I know what I have written and how I expect it to function, I struggled with testing for everything at first. I made sure to try and test all areas of the code regardless of how well I think I wrote it. Bias could play a major role on the developer side when testing your own code. For instance, when creating measures for throwing exceptions, I knew I wrote the exception properly. In addition, I knew there were basic checks in place to ensure that the code would throw the exception properly. When testing, I did not test if the exception was thrown, but rather if the parameters were within the acceptable range. For example, testing that the name of a task was within ten characters. However, the issue is that I did not test to see if the exception was thrown if the parameters were outside those limits. This is something that would not happen if there were separate teams, one for development and one for testing. This is the challenge when it comes to testing your own code. Limiting bias is a skill set as much as any other when it comes to software development. A lesson I learned quickly. Ultimately, I tried to pretend that I was looking at someone else’s code for the first time and develop tests that way. Rather than going in knowing what I had already written.

Throughout this project, as well as this course all together, I have learned that being diligent in testing and focus on the quality of the program are the most important areas of software engineering. When a project is complete the end result has your name on it regardless of how it functions. Attached to that is a reputation that the client will have the final say in. It is our job to ensure that we are delivering the best possible product to our clients for the sake of our company. If we cut corners when testing or writing programs, we can easily make mistakes that would normally be avoidable. This in turn would cost us financially. The reputation of the company is tarnished and the financial woes that follow that could lead to larger consequences. My plan for avoiding technical debt is to focus on the client. Ensuring that the client comes first will prevent myself from committing silly errors such as rushing through things or overlooking areas that I believe are technically sound. Essentially, getting out of my own way when it comes to development and testing. Removing ego and staying humble are things that go a long way in most professional programs. If I had skipped some of the tests that I though may not be necessary, like the one that checked if the error was thrown in the task name example earlier, I could have jeopardized the entire project. Sure, the parameters would not match, and the program would not work, but there would be no indication as to why it was not working. Imagine if your password for Netflix did not work. Now imagine that Netflix did not give a reason why could not log in. Instead of getting the message saying that your password or email did match, it just reloaded the login page. This would be both frustrating and confusing. My goal for the future is to ensure that these types of error are addressed. I want to deliver quality programs to my clients and to do that I must take my time and remove my own biases.