Mike Sabeiha

SNHU: CS320

Module 7

2024 12 14

**Summary of Work**

**Aligning Work with Software Requirements**

The testing requirements were consistent across modules, and I feel that the testing approach I used was well aligned with the software requirements. I often struggle with over-engineering my designs to account for every unusual situation. This leads to time wasted and can lead to overly complex “spaghetti” code that proves non-functional. Ultimately, I believe the minimums were met and that the work I did was in line with expectations without being overly complex or devoting too much effort to addressing low-probability scenarios. All three modules, Contact Service, Task Service, and Appointment Service, all held roughly the same structure which made work convenient and portable. Any problem I found in one component was likely to appear in other parts so that meant any solution could also be used. For instance, they all had validation features to ensure inputs were not null but also did not exceed a certain length. To meet this requirement, I used the following lines of code which tested input for either of two criteria and threw exception errors if they were met.

“

if(firstName == null || firstName.length() > 10) {

throw new IllegalArgumentException("Invalid Name");

}

else {this.firstName = firstName;}

“

**Measuring Efficacy with Coverage Tests**

My JUnit tests were challenging to build because they forced me to rewrite my code so that it would be easier to test but also more resilient. I felt that the quality of these tests were more than enough in the end. As the testing rigors improved, so too did the quality of code and the quality of the tests. I was able to utilize the automated tests to check weather certain basic mechanisms were functioning correctly, such as user inputs, creating objects, and the like. Although I was worried about overdoing the automated testing, after the work was done, I was able to keep the testing to a minimum and add more coverage to suit the unique demands of each module.

The coverage tests helped me to confirm that my testing was adequate, and the coverage percentage needed was a minimum of 80%. On a live project, I would shoot for coverage in excess of 90% and attempt to approach 100% during late-stage or post-development. Not in the pursuit of perfection but because a coverage percentage that is struggling to improve can be an indication of faulty code.

During this project, I was able to push coverage rates beyond the necessary 80% mark on all modules and was only able to do so once I added more robust validation checks in the subject code. From here, I know that I have workable code that provides groundwork, upon which I can build more complex testing outside of the usual exception checking and get these rates over 90%. Then from there, I can finish the project and gradually progress toward nearly 100% as the project’s development closes. This tells me that my JUnit tests were of sound quality and effective in keeping me abreast of the systems good functioning (EDUBCA, 2023).

**Quality of Technical Design**

While the testing environment helps to provide a protected space where code can be observed and improved in action, success here doesn’t necessarily ensure results. By adding certain blocks of code, assuming they pass their respective tests, I’ll know that my code is likely going to be effective in live use.

For example, in the module Appointment Service, the code was technically sound and helped to guide the creation of an object which would hold the unique information of that appointment such as its description and date. The testing for this block helps to ensure that a basic function such as creating an object and assigning its attributes happens upon request and without error. The following lines of code helped to test it by feeding invalid data and checking that it throws an exception.

“

@Test

//test to see if null additions are rejected

void testAddBadTasksNull() {

Assertions.assertThrows(IllegalArgumentException.class, () -> {

TaskService.NewTask(null, "x");

});

“

**Efficiency of Code**

While the efficacy of the code as well as its technical structure are all important, the entire purpose of computer systems is that they do what humans can do but much faster, more consistently, and far more often without failure. This can’t happen if the code itself is not efficient. We, as developers, cannot ensure that the systems we’ve built are any good unless we have a gauge of their efficiency. This can be measured in lines of code written, the speed of execution, the resources needed to function, and other such metrics. When working through this project, I wanted to be sure that I did not overload the testing modules with redundant and unnecessary code.

I built code into each testing module to feed the object creation functions with input that is null. No component of the system is supposed to accept null input. So without insisting upon each possible alternate scenario that might include null code, I instead opted to test only two scenarios that each included one null input. Theoretically, any null input should throw error and if in any case, an error is not thrown, it would indicate that some portion of the code is weak and should be rewritten along with improved testing. The following block of test code from Task Service helps to ensure that new tasks cannot be created with null inputs (Obregon, 2023).

“

@Test

//test to see if null additions are rejected

void testAddBadTasksNull() {

Assertions.assertThrows(IllegalArgumentException.class, () -> {

TaskService.NewTask(null, "x");

});

Assertions.assertThrows(IllegalArgumentException.class, () -> {

TaskService.NewTask("x", null);

});

}

“

**Testing Methods Used**

Throughout the project, I employed a few different methods to test the work being done. A well-rounded approach helped assure me that the code was both effective and in line with prescribed requirements. In all, I used both manual and automated testing. This combination of testing took much of the effort off me and also improved the consistency of results. The JUnit testing employed as the basis of automated testing was more systemic and reliable than simply using the classic “print false” checking that I usually employ while testing work on the fly. The manual testing, which included visual inspection of the code as well as designing unique and demanding scenarios to test the system helped to push beyond the limits of automated testing and allowed for improved coverage rates.

An example of this is the automated testing feature that was present in all modules which tested the unique identifier of every object created to ensure it was exactly 10 characters in length. The test created new objects then pinged the object after creation to read the unique string and throw exception if it happened that an object was created with a bad identifier. As this was not supposed to be possible, and anomaly would mean there was weakness in the code which would demand improvement, otherwise, the work would not have met basic requirements.

Another example was to pull and read dates used to build Appointment objects in the Appointment Service module. In Java, dates are handled as a separate data type different from strings or values. This meant it needed to be engaged differently than other attributes. To make matters worse, the standard Java library deprecated its old date system and replaced it with a more complex version and the two aren’t easily compatible. For this, it was important to build focused unit testing that would prevent false positives. Sometimes weak validation checks seem like they’re comparing two pieces of data when really, they’re just checking if there is data to be compared at all. Considering the unique nature of the date data-type, I built testing that would ensure validation checks weren’t producing default outputs which the testing systems would then misinterpret as a passed test instead of a failed test (Software Testing Methodologies, n.d.).

**Testing Methods Not Used**

There are dozens of different testing types, but they can be reliably bowled into categories that align their purposes or their subjects. Of the methods I did not use, Systems testing, Security testing, Regression testing, and Integration testing were some that are commonplace in development of any project.

During late-stage or post-development, it’s important to test the functioning of various components as they work together. This is typically known as Systems testing. While I made considerable progress toward a finished product, the project is still not quite a complete system, so I did not use this testing type.

Today, most software is accessed on the cloud which makes security more important than ever. Any development team would need to eventually need to test the security of their systems to prevent exploitation. This is known as Security testing and was not done because the project is not ready for deployment and doesn’t need security just yet.

When building code, it is natural to make major changes to the overall project and this usually necessitates the separation of old and new systems as different versions. When deploying the code, sometimes the new code can interfere with the old code in unexpected ways when the newest version is deployed. This interference can lead to previous solutions being neutralized and old problems resurfacing. Ensuring this is prevented is known as Regression testing and this was not done because the first version of the system is not yet complete.

Complete systems that are deployed live and in-use typically require coordination between multiple counterparties such as a user interface, host servers, and technical devices. Integration testing makes sure that software works anywhere it needs to work by assessing functions between the system under development and anything else engaged. The project has yet to reach a point where outside systems are involved and for this, integration testing was not done (Software Testing Methodologies, n.d.).

**Reflection After Work**

**Perspective**

During this project, I was forced to view my work from the perspective of someone looking for weaknesses in my code. Instead of building code that solves the exact problem I’m targeting, I was challenged to build code that solved the target problem as well as other problems that would arise during execution. This meant viewing the potential negative outcomes and finding ways to produce those outcomes, then creating features of the system that would prevent such outcomes. In other words, I needed to be cautious and risk-averse rather than focusing solely on getting results. Caution helped me better understand how the different components relied on one another to work correctly but examining whether they might fail to coordinate or support one another.

For example, in Task Systems, I built one module to create task objects and another module to interact with those task objects. If I ignore the potential failures, I would not have checked to ensure that objects were created correctly and also updated correctly. This wasn’t the case until the testing made it clear that objects could be created correctly but updated incorrectly. Following this realization, I was able to build better code for all modules that would create good objects and update them correctly.

**Limiting Bias**

While reviewing my code, I relied on the JUnit automated testing to help keep a sober view of the work. The testing tools annotate the number of possible scenarios that a certain block of code might address and then the number of possible scenarios that the code is able to address. These numbers might have seemed arbitrary but helped me to apply rigor to building the validation checks so that more scenarios would be addressed without failure.

If I was manually testing my own code, the quality of my tests and therefore the quality of my code would be correlated only with my ability to produce code and to build thorough and effective tests. However, as a person, a lack of sleep, disinterest, or poor experience could all single-handedly sabotage the project. I might cut the testing short out of frustration and ship bad code because I couldn’t find a solution to a problem. This isn’t good business sense and is also unethical depending on your circumstances. Having a good automated-testing system to support me certainly improved the quality of work.

**Importance of Discipline**

Any work requires consistency, and volatile workflows can produce poor results and failure. As a software engineer. it’s important to remain disciplined in producing code that moves the project forward while also making sure that code is at least good enough to advance the project to the next stage. If code is ineffective in itself, does not produce useful outputs for the rest of the system, or has absorbed an inordinate amount of time and resources, it can derail the project or lead to failure entirely. It’s essential to abide by the “good enough” principle. Never ship code that is not good enough while also being sure to ship code once it’s good enough.

When we prioritize fast development times over the quality of work, we often write code that gets quick results but then requires more effort later to correct the code so that it will work better with the rest of the system. This gradual accumulation of additional work is known as “technical debt” and can cripple projects. I plan to avoid technical debt by examining the role that a component will play in the larger system and making sure that I leave outline code in place for the various connections with the larger system. I also plan to build good reference structures to track helpful information such as custom objects, calendar conversions, and string swaps.

A good example of this is when building trading systems, we often track time according to Greenwich Mean Time for simplicity. As a user, I’ll never be able to easily convert this to local time because it’s just not important to me. So, it’s helpful to have reference data sets that are accessible with custom functions that take the effort off the user and convert GMT times to local times such as EST and account for things like daylight savings time. If the structure is already in place to execute a function, and there is already basic code in place for a crawlable library, it saves the effort of having to rewrite code later to make room for all of it.

It gets more complicated when you realize that exchanges, servers, and internal systems don’t all track the same time and latency through information relays can cause disruptions and result in losses. Which means that a constant reconciliation of time between all counterparties must be done, latency tracked and accommodated, and fail safes placed.

In fact, there is a field of financial math that observes dataflows and works to predict changes in assets prices based on the consistency of data feeds. If latency between internal systems and exchange systems is usually 0.003 ms but we haven’t gotten a fresh ping in 0.005 ms, this may indicate that networks are overwhelmed with a flood of traffic which is typically associated with a relatively large change in price as a large number of orders are placed near simultaneously .

Another good example, again, when building trading systems, is string swaps. Depending on where and why, the exact same security is tracked using different names and unique identifiers. While this information is typically shared with invested parties, it is not something to overlook and essential to build core functionality within internal systems that ensure the correct asset is being observed.

This can mean keeping a repository of unique identifiers, common names, abbreviations, and symbols for any security as well as where it might be relevant. It’s also necessary to build functions that can access the repository and make the requisite conversions as needed throughout the system.

If I don’t earmark code to keep space available and account for the eventual design needs of such a tool, I would likely need to rebuild the system from scratch as object libraries, controllers, and packet processors don’t always play nice. If I wait too long to ensure this essential feature is in place and functioning correctly, the code might be too complex to properly build, test, and implement a string swap device.

# References

EDUBCA. (2023, March 30). *JUnit Code Coverage*. Retrieved from EDUCBA: https://www.educba.com/junit-code-coverage/

Obregon, A. (2023, April 1). *Java Performance Tuning - Profiling, Benchmarking, and Optimization Techniques*. Retrieved from Medium: https://medium.com/@AlexanderObregon/java-performance-tuning-profiling-benchmarking-and-optimization-techniques-d71e26e9845b

*Software Testing Methodologies*. (n.d.). Retrieved from Smartbear: https://smartbear.com/learn/automated-testing/software-testing-methodologies/