CS320 Week 7: Project 2

Anthony Fillmore

2/22/2024

I approached unit testing on Grand Strand Systems mobile application in an iterative way, developing a specific feature and then proving the individual components of that feature. I made sure to test those components through different permutations each component could see, using techniques like boundary testing to ensure values just within and without a boundary behave as expected.

Having clear requirements allowed me to build the application code and then prove that the application code fulfilled those requirements. The requirements were the work drivers for me – the blueprint to understanding what the application needed to do, and how I could prove it. There was a common need throughout each service that values needed to be within certain ranges. These requirements became my tests; we may have a date value that needs to be in a certain format (datetime) and below a certain max length (10 characters). We can prove our application enforces this by testing – that when we have a datetime format and a length of 9, we are successful. We also test that if the format is NOT datetime OR has a length greater than 10 (two separate tests here), we throw the appropriate exception. We can rinse and repeat this for the rest of the requirements and as we write our tests, we prove we’ve met our goals.

I ended with an 83 percent coverage percentage, which I felt very solid about considering JUnit was factoring in my test code into the percentage for some reason. I tried researching why this was happening, but I never really found an answer, I have to think there is some way to disable certain files from being scanned. I’m confident my test coverage is 90+ percent if the coverage tool wasn’t finding things it thought needed to be tested in my test code (how do you write tests for tests ha?). I made sure that I had permutations to cover major angles of each service and I really utilized the requirements as my litmus test to uncover gaps.

I enjoyed writing code in this class, I’m not the biggest Java fan but I found the tests to be easy to pick up based on my experience with Ruby and Python. I think especially my Python experience helped me here, the defined Setup – Execute – Assert phases are very similar methodology wise, which helps when structuring my tests. I learned a good amount unit testing while writing the code and tests, I feel way more confident using Eclipse and setting up an environment where I can both code and test for future projects.

I ensured my code was technically sound first, by following the requirements given. For example – In my Task class, there was a requirement that taskID was an immutable attribute after being set one time, in this case about Class instantiation. I also did not provide a way to update the the taskID, making it clear that the field was meant to not be changed. In both my Task and Contact services, I made sure to combine multiple checks into single if statements by chaining them together with OR statements. I can check if something is NULL OR meets value length limitations. By following this pattern, I can streamline my code and make sure to avoid unnecessary lines. I also have a pattern I follow where when I’m going to set or delete something from a HashMap, I check and make sure the object exists before I act. This protects my application from crashing on trying to act on something that doesn’t exist. I do this in several places throughout my code, a specific example is in my Task service on lines 43-45.

I ensured my code was efficient in a few ways, one I have already mentioned is combining multiple checks for attributes into IF statements by chaining them with OR statements. Line 40 of my Task class is a specific example of this – I check for NULL or taskID.length() > 20 and throw an exception if either condition is met. I also use a HashMap in both the Task and Contact service to store multiple objects, which helps me store multiple key pair values in a single structure. I can easily parse the HashMap for unique ID’s and retrieve their values, and I can also ensure that I don’t store duplicate ID’s by checking the HashMap if something exists before I create a new object. Lines 7-11 in my Task Service class is one example of where I use this. Lastly, I made sure not to add any code that wasn’t specifically aligned with requirements or testing. The code doesn’t have any bloat or unneeded attributes or methods, which is an efficient approach.

One of the main techniques I used for this project was Unit testing, focusing on isolating individual components and making sure they work regardless of the rest of the application. For example – in the appointment milestone I had a testing suite for the appointment class and for the appointment service class. I focused on testing constructors, getters, and setters individually to ensure each piece works individually. This also parallels white box testing techniques as the tests required knowledge of the internal workings of the code. I also employed boundary testing, making sure I tested with values that were just inside and outside the limits given by our requirements. Examples for this include testing that the appointment class was successful with a date just after the current date and throws an exception with a date in the recent past. This type of testing ensures that the input parameters we expect are handled according to requirements and that values close to the thresholds are handled appropriately.

I didn’t employ integration testing, system testing, performance testing or user acceptance tests for this project. Integration testing combines modules in code to ensure that the pieces and parts work correctly in tandem. Integration testing usually means testing the end state of a process – For example when a database method is called, a connection string is created, a sample set of data is inserted into the database and the connection is closed. We could measure that: The database data reflects the expected updates and that there is no open connection to the database. System testing entails testing the ENTIRE system end to end. This can also be called “black box” testing as it isn’t concerned with the inner workings of the application, just the input to the application and the expected outputs. Performance testing involves simulating different loads on the application and ensuring that things like response time and resource usage stay within a defined acceptable range. We might simulate 100 requests to our API, 1000 requests and then 10,000 requests – measuring the response time along the way to ensure we stay within a certain range of responsiveness. This type of testing can provide valuable information and allow us to adjust the number of resources we provide to these processes. Lastly, with User Acceptance testing, we allow our end users to use the application in ways they normally would. Users will often do things that we might not expect and give us considerations for different permutations that we hadn’t considered. For example – We may have a certain error message occur when a resource is unavailable. By simulating this situation with an end-user, we can gain valuable feedback on whether the message is appropriate and clear. We also are more likely to find bugs and missed requirements as users do things that we didn’t expect – for example a user might change their password with a character we were supposed to filter for, allowing us to correct this mistake before productionizing our application.

All these testing techniques should be employed over the lifecycle of a project. Unit testing should be done iteratively as features are delivered to ensure code works and meets requirements. Every project benefit from unit testing because it helps produce bug free code and allows us to quantitatively measure delivered requirements. Integration testing has a huge benefit in helping assure that code works end to end. Testing the moving parts of code with real test data and then measuring outcomes helps us prove that changes to isolated components integrate into the system without issue. I imagine if we had an ATM application, a great integration test would be testing that when a user deposits money in the ATM system, that a job triggers to update the users account with the deposited funds, and then a job is triggered to inform the customer of a successful deposit. Performance testing helps us fine tune our application, which is crucial in meeting business objectives and end user satisfaction. Measuring things like response time, server resource consumption or database read/writes allows us to benchmark acceptable metrics. It can also give us an idea of how and when scaling might be appropriate as we reach certain resource consumption thresholds. User acceptance testing helps us get ready for production by allowing users to use our application as they would with normal usage. For example – if we were building a website, we could have a sample group test the sites’ different links and input forms. This would give us valuable information about response times, website layout and uncover possible permutations we hadn’t considered. We want our products to be user-friendly and efficient, and user acceptance testing is a great way to flush out these details.

I tried to adopt two mind-sets while building this project, attempting to put on different hats while writing my code, versus while testing it. While I’m writing my code, my mindset is to make sure my code meets requirements, delivering the expected features. I want to protect my code from failures, gracefully handling things as I go and making sure my code has a happy/unhappy path. As a tester, my goal is to break the code any way I can. I want to poke holes in the work done, throwing the “kitchen sink” at the code in attempts to prove it is **NOT** sound. It’s important as a tester to appreciate the complexity of the code being tested as things are interrelated – for example the Task Service class relies on the Task class object instantiation. We want to be sure when instantiating a Task class object that we ONLY pass values to the Task class that are acceptable, otherwise the object creation will throw exceptions. It benefits us to protect the class in this way to ensure objects we create from the Task Service are successful.

Limiting bias while reviewing your own code isn’t entirely possible, and that’s why having others review your code is crucial. One technique I have is I try to take a break between writing the code and starting my testing and review. This gives my brain a mental pause, a pallet cleanse if you will, between changing hats. Bias is still a big concern regardless of techniques employed, I’ve submitted plenty of code that misses a permutation and doesn’t catch what I thought it did. For example – having a requirement to sanitize a string to ensure it is transformed to lowercase, has less than 10 digits, is not nil, and must be alpha-numeric. I may cover all permutations with my unit tests but if I don’t write the corresponding integration tests, I may be passing something from an exterior method that breaks the new change I’ve added. If I’m solely responsible for testing my own code, this can easily be missed and is a great example of why code and peer reviews combined with automated testing in CICD pipelines is crucial to eliminate testing bias.

The importance of being committed to quality in our field is multi-faceted, it affects reputation, cost, time, safety and more when it comes to the products we build. Cutting corners can increase these “costs” and lead to financial burden, to more serious implications like injury and loss of life. Technical debt will always occur, but mitigating what and how much is important. Sound testing plans that incorporate a layered testing approach that includes unit, integration and other testing methodologies is the first step to lessening technical debt. Clear documentation, code and peer reviews and solid requirements are also crucial in reducing this debt. For example – requirements that are vague or not accurate will lead to features being developed inaccurately. Releasing a feature that is half-baked causes rework, in the form of coding, testing, and releasing production updates to fix these features. Dealing with this technical debt reduces the amount of time engineers innovate and build new features, leading to customer dissatisfaction, all things we are trying to avoid with our commitment to quality!

Anthony Fillmore

**Resources:**

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). Software Testing: An ISTQB-BCS Certified Tester Foundation Guide (4th ed.). BCS Learning & Development Limited.