Mark Holden

CS-320 Software Test Automation and QA

6/19/2022

Summary and Reflections Report

The approach that I took to unit testing for each of the three features was slightly different. While implementing the first feature, I had no existing code, and as a result I followed the three rules of test-driven development (TDD) closely. I did not write any production code until I had a failing unit test, I wrote no more test code than was required to have a failing unit test, and one I had written enough production code that the unit test would pass, I wrote no further production code. For the second and third features however, I already had most of the tests written because of the similarity between each feature. For the tests that were not exact duplicates, there was only minor refactoring necessary to meet the requirements. For those features, there was also a great deal of copy/paste/rename in the production code, but since the tests were already in place it was easy to know when the refactoring was complete because all the tests would pass. Had we been working in .NET, there is a neat feature that would have made the whole assignment trivial, and that is custom validation attributes. Some of the validations required for the assignment are already built into .NET, but for the ones that aren’t, developers can create and apply their own custom validations in one line to any object, and the implementation of that validation in it’s own file that can be reused.

The extent to which my testing approach aligned with the software requirements was within reason. There are examples of requirements that do not have a specific test for them, and there are examples of instances where the requirements were not clear, and I made a judgement call to add a test case covering an edge case not specifically called out in the requirements. The examples where a requirement does not have a test for it specifically is the requirement for both Contacts and Tasks where the Id of the object is immutable after creation does not have a test specifically for it because the private variable is final and set in the constructor. I intentionally did not create a mutator method for the property because it is immutable and as a result there is no method for a test to call to verify the functionality. The example of where the unit tests that I wrote go beyond the specific stated requirements is for the fields where the requirement states that the field in question “shall not be null”. The requirement does not indicate if an empty string is a valid value, but my assumption is that if null is invalid, then empty is most likely invalid as well. To cover both of those cases, the JUnit ParameterizedTest enables one to set the attribute NullAndEmptySource to add test cases for both null and empty string.

Code coverage is only one metric that can indicate the quality of a unit test suite, and it is not a good one. At work, I strive to follow the rule that unit testing should have 100% code coverage, but that alone does not guarantee that the tests are any good. The tests should not only be checking for edge cases and that the system under test can follow the affirmative code path when inputs are all as expected, but they should also test that the correct behavior occurs when any exception that can be thrown is thrown. Beyond the percentage number, the requirements of the project must be the driver for which the tests are developed. Since I followed the principles of TDD during the development of this solution, I have high confidence that the tests reflect the requirements and that the code reflects the tests. The principles of TDD that were followed during development were that production code may not be written until a failing unit test is written, as soon as a test fails no more test code can be written, and compile failures are failures.

Writing JUnit tests is similar to other testing frameworks that I have used in the past for .NET such as NUnit and XUnit, so the experience in writing it was not unusual or difficult. The primary challenge was the limitations of Java – in .NET, I use LINQ expressions quite a lot at work and Java streams fall short of the ease of use of LINQ.

The method by which I ensured that my code is technically sound was a heuristic approach based on five years of experience as a software engineer. I do not pretend that I am infallible or even have any special ability, but code reviews, research, mentorship, and practice are the ways that I have accumulated my heuristic model. Some of these heuristics are based on the SOLID principles, the don’t repeat yourself (DRY) principle, and different design patterns that I have picked up over the years. By no means does this guarantee that my code is technically sound, especially considering that I primarily work in C# .NET. I was intentional about writing clean code when I developed this solution, so it is a demonstration of my current skill level as a software engineer. A specific line of code from a test that may highlight that the overall code is technically sound is:

ArgumentException thrown = Assertions.assertThrows(ArgumentException.class, () -> {

new Task(taskId, "Do Work", "Do all of the things");

});

This line of code indicates a few things. First, the test expects that an ArgumentException will be thrown. The ArgumentException is a custom exception type. While custom exception types is not a particularly advanced skill, it does a good job of communicating an issue to a calling function without cluttering up the return type. Second, the assertThrows method accepts a function as a parameter, so declaring an anonymous function using the lambda syntax could be an indication that the code is technically sound. Finally, the test accepts multiple value sources for the taskId parameter, so it tests three of the cases where the ArgumentException would be thrown for one implementation of the test method. Using the DRY principle in this way doesn’t automatically mean that the code is technically sound, but these things could be indicators that the author knows what is going on.

To ensure that my test code was as efficient as possible, I used a private system under test (SUT) field in the test classes so the system under test would only need to be constructed once (with the exception of the constructor tests for the Task and Contact objects). The line of code that would indicate that I am thinking about efficiency when I develop software is:

\_sut = new TaskService();

There is only once instance of the SUT for all the Service tests, and there are as few instances as possible of the object in the object tests as possible while still thoroughly testing the constructor. With regards to the efficiency of my production code, I really did not apply any special techniques to optimize efficiency because these classes are all very simple, and premature optimizations without knowing what needs to be optimized is not only a waste of time, but it also can create unnecessarily complex code for no reason.

The software testing techniques that I employed for each of the milestones were unit testing and TDD. I was mainly using TDD during the milestone for module three, but once I had the base code written, it was much easier to copy and paste the files wholesale and make minor modifications to meet the requirements of modules four and five. For the minor differences between the classes and services in the modules, I used TDD once again to make sure the code that I wrote was covered, and that I wrote no more production code than was necessary to make the tests pass. The characteristics of TDD are that the code author cannot write any production code until there is a failing unit test, and once a unit test fails you must stop writing test code (failing to compile is considered a test failure). This technique is sometimes called “red, green, refactor” because the test must be red (failing), then green (passing), then you refactor your code and the test should still pass.

Some software testing techniques that I did not use during the milestones were integration testing and smoke testing. I did not write any code that connects any of the services to one another, so writing integration tests for classes that do not integrate into one another would be impossible. The characteristics of integration testing are that all integrated systems that are being tested are deployed to the same environment and test cases are executed that cause one system to call another or potentially many others and then when the process completes, the outputs are verified to be the expected values. I also did not bother smoke testing my code because the requirements were very simple, and the solution is at 100% code coverage. The characteristics of smoke testing are that with enough knowledge about a system, an engineer can know what parts of the system are most likely to be affected by a change, so the focus of testing can be on those sections only without having to run extensive manual tests.

During this project, the mindset that I adopted was to meet all the requirements precisely and cover any obvious cases that could be extrapolated from the requirements. In acting as a software tester, I did not knowingly employ caution to any extent because the software was simple, and I have years of practice writing similar systems. In my opinion, practice working on complex systems makes it easier to think of a system that only has three object models and three services as a simple system in comparison to the system that I work on during the week which has hundreds of objects spread across hundreds of services in different microservices that all work together in an orchestration of a source of truth database for each service plus an additional event driven architecture and keeps a query database up to date to reduce the load on primary services for simple data retrieval. An example of the simplicity of the project is that none of the objects had any dependencies on any other objects, and the services were simple create, read, update, and delete (CRUD) services. During my interview for the job I currently hold, the first phase of the process was to write a CRUD service from scratch while sharing my screen with one of the principal software engineers for which I now work that had at least the level of complexity as this project, in the time limit of one hour.

One of the ways that I attempt to limit bias during code reviews is that I rely on the research that I have done on how to be a good code reviewer. Some of the ways I apply that is to only review changes, not optimizations that I notice on code that was not part of the change. I also endeavor to apply the standard of “is this making the overall code better or worse” when there are things in the code that don’t follow my personal preferences or are not as clean as they could be. The final way that I limit bias in code reviews is that I nit-pick everything regardless of who wrote it. When I do this, to the best of my ability I phrase these nit-picks as suggestions or ask the code author to consider an alternate. On rare occasions, when I see unintentional breaking changes, I will request changes on a pull request, and actually block the user from merging. The company at which I currently work has a rule that the author of code writes all required unit tests for their own code which are also subject to code review, but that the author should not be the one to write automation tests for their code. In one specific case, I was authorized to write automation tests for code that I had written, but it was several weeks later, and our quality assurance engineers were swamped and just did not expect to get to it before we needed my code to be released. It is especially important to test exactly to the acceptance criteria in that case because I knew how it worked, so I could have easily written tests that would pass based on that knowledge that would have not been effective tests of the system. The other thing I have done in a previous job where there was no code review process, (in fact code reviews were discouraged for some reason I still have not been able to determine) was that I would review all of my code from the previous day in the morning when I got in, fix any issues and then deploy the code. Having my own fresh set of eyes on my code was the next best thing to having a different set of eyes on it. At that time, I was still a junior engineer and had only had one software engineering position for 3 years before that, so my experience was limited to the frameworks and patterns used by that company alone, and as a result I really needed those code reviews to learn and grow.

It is important to be disciplined in your commitment to quality as a software engineering professional because even in the least impactful case, the company for which you are writing code depends on the quality of that code to pay your salary, and in the most impactful case, lives depend on the software you write. I would not like to work with other engineers that are just filling a seat and collecting a paycheck and are willing to cut corners when writing or testing code. Cutting corners is unacceptable in any part of life, it is most certainly unacceptable in your profession.

The method by which I currently avoid technical debt is to fix things immediately as I see them whether I wrote the original code or not. The same is true for websites that I visit – any time I see a misspelling on a page, I seek out the contact information for the company and notify them of the misspelling. It is my opinion that if everyone follows this approach in all aspects of life, it will improve their own life and the lives of everyone around them. A specific example of when I did this was when I noticed a spelling error multiplied by copying and pasting, there a file had contains spelled “contians” over 40 times. It wasn’t my code, but that doesn’t matter. I fixed the misspelling and opened a pull request with the change right away. The same is true when I see other code smells or obvious refactors that need to be made.