**Module 7 Project**

**Project 2, Summary and Reflections Report**

Kyle Keiper

Computer Science, SNHU

CS-320 Software Test Automation

Professor Rangitsch

August 13, 2022

Project 2, Summary and Reflections Report

In reading through the requirements, I realized I have already answered questions 1a and 1b in a previous paper. I am going to defer to that paper for my answers to those questions. It begins as such:

My testing approach begins with the constraints described in the assignment. I copy those into the TestCase files, and then develop a test for each constraint. Sometimes, I combine multiple constraints into a single test method containing multiple assert statements. As an example, one of the constraints given is that the Task’s ID cannot be longer than 10 characters. I copied that into my TaskTest.java file, then rewrote the comment until I had a testIdNotLongerThanTenCharacters method which expects a TaskException object to be thrown.

Code Coverage comes from the Test classes covering possible outcomes. My quality is high, but not 100%, as there are branches of code that I am not testing in the TestCases. For example, I did not test to make sure deleting a Task that doesn’t exist returns a null object. By reviewing the highlighted lines in Eclipse, we can see what test cases we may have to write in order to cover all branches of logic.

Part of it just comes from experience. I’ve written code for 10+ years now, so sometimes, it’s hard to go back to what makes it sound. Probably the best way to do so is to reference the above question; by creating test cases, then running coverage on the project, we can see which branches of code are never checked for errors. When I ran coverage on the TaskService project, it told me that TaskService.java:61 was not covered in the TaskServiceTest.java file. This helped me know that I need to add a test that makes sure calling TaskService.delete with an invalid ID returns a null object.

I’m not sure how Tests make sure my code is efficient. Testing my code makes sure it works, but it makes no statement about how many clock-cycles I could potentially save if I rewrote a branch. Computers today generally are powerful enough that efficiency and clock-cycles are a “solve it if needed” issue. I tend to write my code to be readable, rather than efficient. As an example, I created a custom TaskException class, and an enum consisting of strings. By combining those two things, I was able to create a system where I only change error messages in one file, and can test what the root cause of a TaskException is by comparing it to an enum field. In TaskTest.java, I expect a TaskException to be thrown when I set a task’s name to null, then I assert that the exception’s message type is the same as ErrorMessages.ERROR\_NOT\_NULL. In this sense, I’ve prioritized readability over efficiency, because try-catch blocks generally are not very efficient. (Keiper, 2022)

In looking at question 2a, it appears this was mostly answered in our Software Testing Techniques Journal, so again I’ll defer to that paper for answers:

I would say I definitely used Structure Based and Specification Based techniques, specifically Decision Coverage, Statement Coverage and Equivalence Partitioning. Decision and Statement Coverage are fairly self-explanatory, with it just being that I ran the Coverage As option to make sure I had 90%+ of my code tested. Equivalence partitioning was done by deciding what was valid (unique IDs) and what was not (IDs already in use), then making sure one item in the bad partition caused the entire test to fail.

One that I didn’t use was State Transitioning. The deliverables I have been working on are stateless items, in that there’s no transitioning from Contacts to Appointments to Tasks. As a result, there are no valid tests to write (as of yet) that would test to see if it’s possible to move from Appointments to Contacts, as an example.

While going unused, State Transitioning is probably one of the more interesting ones. Finite State Machines are curious pieces of code in engineering. I like to dabble in game design for fun, so I’ve used state machines to handle what gets rendered based on what state the game is in. There are valid and invalid transitions, so using Transition Testing to make sure exceptions are thrown when moving from, for example, STATE\_MENU to STATE\_GAME\_WON. It wouldn’t make sense for a player to be able to skip everything between those two game states, so a State Transitioning Test would be ideal to make sure that doesn’t happen. (Keiper, 2022)

To what extent did I employ caution? I would have to say I was not very cautious; reading the information this week about escapements and released defects made me realize that I only did the bare minimum in designing my tests for the class. I remember reading the requirements and writing my tests to cover them, but I didn’t use my imagination to try to think of how those should be extended for safety.

In trying to limit bias, I took a Test-Driven approach. It’s easy to write the code first and then assume it works; it’s like proofreading our own papers - we miss the mistakes we made because we read right over the weak spots without noticing. Test-Driven development addresses this by writing a test with the requirements *that fails*, then writing the deliverables until the test passes. This helps alleviate mistakes of not seeing your own errors.

Why is it important to be disciplined in quality? Well, a lack of discipline can end up costing millions of dollars. No one wants to end up like the Ariane 5, just because they didn’t notice they were coercing a 64 bit float into a 16 bit integer. A mistake like this can cost hundreds of millions of dollars. Thankfully this was an unmanned flight, so no one was injured, to my knowledge, but imagine something where people are physically injured; the mental fallout of someone becoming a casualty of my code would be nearly unbearable. That’s why it’s important to be thorough when writing tests.

Lastly, avoiding technical debt is done by writing thorough tests at each iteration of software. The debt is incurred when changes in an iteration of software affect previously functional features. Writing thorough tests during each iteration helps us catch when our changes break the code (because they will!) since the test runner will tell us that previously working tests are now failing. It gives us the chance to address undiscovered debt before the next iteration becomes available.

#### 

#### **Reference Page**

Keiper, K. A. (2022). Approach to Unit Testing [Unpublished paper]. Computer Science, Southern New Hampshire University.

Keiper, K. A. (2022). Software Testing Techniques [Unpublished paper]. Computer Science, Southern New Hampshire University.