**Project 2**

Ryan Rieth

Prof. Jacks — CS 320 — SNHU

## Summary

My unit testing approach followed a similar path for each of the three milestones achieved in this course and utilized for project 1. This unit testing approach employed white-box testing, also known as code-based testing or clear-box testing. While black-box testing occurs separate from the code with a more end-user mindset, white-box testing occurs with the programming from the inside.

My first point of aligning to the software requirements was meeting the expected requirements of the assignments, as — at least in these scenarios — these sets of requirements exist hand-in-hand. One such requirement listed in the guidelines is making sure a variable does not exceed a certain number of characters. Adhering to this requirement by only accepting input less than that number and then further testing that each instance of that variable does not contain more than that number of characters also adhered to the partnering software requirement, as the variable and its setter/getter methods were scripted with this distinction in mind — if-else statements, try-catch statements, etc. This same mindset was kept through all other variable and method programming through all three milestones.

All milestones possessed coverage ratings greater than 80%; milestone 1 was the lowest with an 85.7% coverage rating while milestone 3 had the highest coverage rating of 92.5%. This data showcases a growing quality of JUnit testing and JUnit test understanding throughout the course. My JUnit tests — after discovering how to properly utilize them — are well-grounded and render the final products very solid. Perhaps a way to better my JUnit testing would be to employ tests specifically focusing on invalid inputs or method calls; while JUnit testing does show when a test fails, it can perhaps benefit a program to test that an invalid entity is properly *caught* and can subsequently be handled rather than just stating that a test failed.

Take these specific lines of code from my milestone 3:

@Test

void testNewAppointment() {

AppointmentService service = new AppointmentService();

service.newAppointment();

assertAll("default appointment",

()

->

assertNotNull(service.getAppointments().get(0).getID()),

()

->

assertNotNull(service.getAppointments().get(0).getDate()),

()

->

assertNotNull(service.getAppointments().get(0).getDescription()));

service.newAppointment(date);

assertAll("specified date appointment",

()

->

assertNotNull(service.getAppointments().get(1).getID()),

()

->

assertNotEquals(service.getAppointments().get(1).getID(), service.getAppointments().get(0).getID()),

()

->

assertNotNull(service.getAppointments().get(1).getDate()),

()

->

assertEquals(service.getAppointments().get(1).getDate(), date),

()

->

assertNotNull(service.getAppointments().get(1).getDescription()));

The scripting for this particular testNewAppointment() method continues, but for the sake of space this snapshot shows two facets — creating a new default appointment or creating a new appointment with an already specified date. Note how the default appointment checks that null values are not set for the variables, and the specified date appointment not only checks for proper length values but also null values again. This is to “overload” the JUnit null testing as well as make sure that the JUnit testing catches improper values from both slightly “different” manners of appointment scheduling. I employed this mindset for my milestones (most notably in milestone 3) to ensure the quality of my code and its JUnit testing as well as make sure my scripting was technically and logically sound. Efficiency is also present in that the assertNotNull and assertEquals JUnit assertions are lumped into assertAll JUnit assertions; while it is slightly more scripting to be done, it can efficiently display one pass for a JUnit test while showcasing a failure or set of failures if each facet does not perform as it should. Another manner of efficiency is “going down the line”, in which I test a default appointment, then one with a scheduled date, then (not listed) one with both a set date and name, and finally one with set values for name, date, and description; doing so provides a logical waterfall for other testers and developers to easily follow along in the testing and debugging. Lastly, testing all these manners of appointment scheduling falls under one testNewAppointment() method, wrapping up the efficiency mindset nicely here.

## Reflection

### Testing Techniques

The two realms of testing existed in my milestone creations: *static testing* and *dynamic testing*. For static testing, I both informally looked over and more formally reviewed what was requested and required for the individual products. I then inspected my code and corrected any obvious bugs or mis-scripts that I could find. Then, for dynamic testing, I utilized unit testing — specifically, JUnit testing. Since the JUnit testing was the focus of this course and the milestone learnings, this is about where my dynamic testing stopped for each one. However, I made sure to JUnit test the individual portions of each milestone — for example, I JUnit tested the Task.java class separately from the TaskService.java class in milestone 2.

More specifically, I utilized white-box testing as my main technique approach for the milestones. White-box or code-based testing is testing that occurs in tandem with the code, which is present and visible to the tester. This is the other side of the coin from black-box testing, which is separate from the code and is testing with a more “end user” mindset in place. White-box testing is very practical for software development projects that range from the small-scale to the large-scale as it allows issue finding as develop progresses, minimizing the number of potential errors that would be found later. Not only that, while adjusting code for these early-found issues, more efficient coding can be employed that otherwise could have been missed.

### Mindset

I employed a cautious and fixed mindset for these milestones — *cautious* in making sure the requirements were met and the testing was properly thorough and *fixed* in following the same working formula for each milestone as each milestone was similar and allowed more efficient coding for each subsequent milestone (this is acknowledged as unique for these presented scenarios and is not applicable “across the board”). Employing this dual mindset helped me better understand and appreciate the interrelationships in the code I was scripting and testing because I can see how separate projects can employ strikingly similar mannerisms both in their scripting as well as their testing. For example, while some smaller details may be different, it would be hard to tell the milestones apart if I removed their respective titles as they all operate with the same functionality in mind: variables with criteria, set and get values for said variables, match the values with the criteria, and test each of these facets of the program; while Task, Appointment, and Contact all serve different entities, they operate with the same kind of objects in mind.

Given I was both the scripter and tester, there had to be a level of bias there that was not necessarily significant to the milestones in question but would be if they were professional projects. If I code something, and it works as flawlessly as it can, then I am naturally biased to that scripting — meanwhile, the scripting could be a rudimentary rendition of a more efficient and powerful method of accomplishing the same goal. Beyond this, I have an uncontrollable bias in what I think needs to be tested; others’ perspectives can deduce more thorough ways of testing. For example, my JUnit testing in the milestones tests for proper inputs but does not necessarily test handling of improper inputs. Other developers’ perspectives could have added this to the projects. Limiting bias begins with adding these perspectives to the mix, so by asking others for their fresh eyes and differing opinions can properly flesh out the project and its testing. Perhaps even lending the code to someone else to finish up, if applicable for a project, can further this armament.

My dedication to the quality of my code is arguably the most important contribution I can give to a project as one of its developers — especially as its sole developer. A project can range from a rudimentary, small-scale operation to something vast and complex that involves sensitive information and the unadulterated concepts of *privacy* and *security*. Developers **cannot** skimp out on code quality because of these realities. Doing so results in exponential growth of technical debt, or code debt. Being a development practitioner for these projects, it is my duty to minimize technical debt (in a perfect world, eliminate it). In my unique case, I had unforeseen technical debt as my saved .java files would not, essentially, translate over to my instructor as I employed a different IDE than the usual Eclipse IDE. This taught me a lesson about technical debt I wasn’t expecting; while it was easier and far quicker for me to script the milestones in my chosen IDE, the act ended up costing me much more time — and, if these were professional projects, money as well — than if I had “bit the bullet” and scripted the milestones in the Eclipse IDE.

There is not a fully right way to go about minimizing technical debt so long as any and several steps are taken. One of these steps could be making sure a specific goal is met before that section of code is sent out, even if it includes delaying the deadline (I’m looking at you, Cyberpunk 2077). It’s a balance of utilizing a scheduled budget with understanding the risks on said budget with focusing on quick code deployment rather than “perfect” code deployment, but companies — especially gaming ones — tend to want to meet original, sometimes unrealistic deadlines instead of understanding the fact that the customer base would rather wait longer for a properly implemented game than play something “broken”. This is but one example, of course, but it is a microcosm of others.