# CS 320 Summary and Reflections Report

Dakota Cromer

SNHU

CS 320

Joseph Rangitsch MSCS

June 20, 2021

During the course, I took what some might consider a unique approach to unit testing and aligning code with the requirements. I chose to use exceptions and exception handling in order for me to take advantage of the string matching in the exception handler without needing to create my own pattern matching code. An example of this from the task class can be seen below

private String checkDescription(String description) throws Exception {

if(description==null){

throw new NullPointerException("Cannot be Null!");

}

Pattern p = Pattern.*compile*("^[0-9a-zA-Z]{0,50}$");

Matcher m = p.matcher(description);

if(m.find()){

return description;

}

else {

throw new Exception("Last Name must be no longer than 50 characters!");

}

}

The above code, first checks if the description is null and if it is, it throws a new exception with the text “Cannot be Null!” This code is useful because rather than setting flags, and manipulating them, I’m able to take the code, and if an exception is thrown in the code block, I can use a try catch block to handle the exception in the task service to properly inform the user of the improper input in the strings.

Anyone can write software but writing software that meets the requirements and functions as required is a must for computer programmers. The best way to test code to ensure proper functionality and that it meets the requirements outlined is the use of JUNIT tests and my code here is no different. I used JUNIT tests to test the functionality of my code and ensure it meets the requirements. An example again of my task unit tests can be seen below

*@Test*

void test() throws Exception {

Task t = new Task("DoAThing", "This task does a thing");

}

*@Test*

void testBadTask() throws Exception {

Task t = new Task(null, null);

}

The code block above has 2 tests, the first which creates a normal task, and the second which creates a task with null inputs. The first one completes successfully. The second one fails after throwing an exception. One more bit of code that needs to be added is the use of a try catch block in the testbadtask so that I can ensure that the program does not terminate fully, but when an exception is throw, by the guidelines specified in JUNIT 5, the test fails. I found writing the JUNIT tests to be a bit difficult for several reasons. First, I had to ensure that the throws Exception declaration is in each test case. Second, I needed to ensure that I was able to properly handle exceptions in my code and so I needed to write JUNIT tests to test the try catch blocks. The nice thing about it is that I figured this all out on the Contact and ContactService classes and so writing them for Task and appointment and their corresponding services was much simpler as they followed similar implementations and therefore similar test cases could be used.

The technique I utilized most for software testing was the creation and use of unit testing. I created unit tests that followed the requirements and ensured that my code functioned correctly according to the System Requirements documentation. Some of the Software testing techniques I did not use, mostly because the complexity of the system was not as complex as would arguably be needed to use them, are Integration testing, Performance Testing, and Acceptance testing. Integration testing would require me to be writing code that needs to be integrated into a larger system. I would be running my code on a fork of the system code and ensuring that it correctly integrates with the system and does not cause any errors that can result in a disruption of the system. Performance testing would require code with sufficient overhead to warrant testing the system resources to make sure the code executes in a timely fashion and doesn’t cause caps in CPU usage or RAM usage etc. Acceptance testing would require a QA team and a team of external users to test the application prior to deployment into production. I don’t have access to external users and I would be a QA team of one person therefore, using Acceptance testing is a nonstarter.

To be completely honest, I work on a billion dollar exchange as a QA engineer so I did not find the code to be terribly complex. However, I can see where newer developers might see more complexity due to interactions of various classes so looking at the system from that point of view, I believe the complexity of the system and the interactability of the code between the task, appointment, or contact and the corresponding service class could cause some confusion if integration between the service and it’s object class is something a developer hasn’t had the opportunity to explore yet.

I found limiting my bias to be extremely difficult. No one wants to think that their code has issues. By writing the code and then testing it myself, I already know how the code works and so I know the ways to make it fail. That said, there are probably ways to make it fail that I was not able see because I have what is known as creative bias. I know the system so I can’t think outside the box enough to make it fail. Examples of this would be writing the test cases to take advantage of the EXCEPTION handling rather than properly forcing an object class to have bad code via the editing of memory.

I believe discipline is a very important thing that QA engineers must have if they want to succeed in their job. You must break a system in every way you can for you to find the cracks. If you are unable to do this, the results can be catastrophic should a customer find it. Examples of this can be see in the MT Gox hacks that caused the system to allow hackers to buy bitcoins for pennies rather than the full price of the bitcoins at that time. The ways I avoid technical debt are usually through the use of software specifically designed to test code by breaking it in the more complex ways such as passing garbage into the inputs, or editing the memory to have garbage inputs.