Mikaela Spence

CS320

**Summary and Reflections Report**

This report consists of the summary and reflections for Project 1 in CS320. This project involved developing the code for an application’s Tasks, Contacts, and Appointments. I was given requirements by a company for these deliverables and wrote the code and complementary JUnit tests to ensure that the code was what the client was looking for. The following is a summary and reflection on what was completed.

**Summary**

**Alignment to Requirements**

Aligning my code with the requirements was an important step in the process. The requirement elicitation portion of this project was done via written instructions. The code was written for each requirement, and complementary JUnit tests were written for each section of code. To develop my Junit tests, I used the given requirements such as “Name cannot be longer than 20 characters” and developed tests where the name is valid and then one with more than the allotted number of characters to make sure that an exception is thrown. I also made a case for null, since the name field cannot be null and made sure an exception was thrown. I repeated this for each field based on the requirements given in the assignment.

**Effective Tests**

For each Junit test file, I ran the program as a Junit test. I then ran it again using the “coverage as 1 Junit test” option in Eclipse to see the coverage percentages. This helped me find ways to improve my coverage. I made sure to write test cases that covered edge cases I could think of, as well as making sure that valid inputs behave as expected. I attempted to write test cases that were independent of each other, that way having one test be successful or fail still allows the next test to be checked. It was also important to make sure the tests were deterministic, where each result must always be the same when given the same input (Hambling et al., 2019). I revised my tests as needed to improve coverage. Finally, I submitted a screenshot of the coverage percentages which ensured that each aspect was covered at least 80%.

**Technically Sound Code**

I try to ensure my code is technically sound by following good practice and logic. I follow best practice by following coding conventions, reviewing the code, and performing testing input (Hambling et al., 2019). I will use the given requirements to guide my work. I try to label things appropriately with comments and follow a logical flow in my classes and methods. I use Junit testing to check for missing pieces. For example, I will add comments and label my variables in a way that is easy to follow and understand.

Ex) //Duplicate task throws exception

@Test

**void** addTaskDuplicateTaskId() {

TaskService taskService = **new** TaskService();

Task task1 = **new** Task("1", "TaskName1", "TaskDescription1");

Task task2 = **new** Task("1", "TaskName2", "TaskDescription2");

taskService.addTask(task1);

*assertThrows*(IllegalArgumentException.**class**, () -> {

taskService.addTask(task2);

**Efficient Code**

To ensure my code was efficient I tried to use industry standards in my coding with naming variables and methods. I also tried to keep things on one line when possible for conciseness. For example: // Validate and set the task ID

**if** (taskId == **null** || taskId.length() > 10) {

**throw** **new** IllegalArgumentException("Task ID must not be null and cannot be longer than 10 characters");

}

**this**.taskId = taskId;

Instead of putting the two arguments on separate lines, I used OR to put them on one line. In addition, I used comments so that someone else could look at my code and hopefully be able to follow quickly and easily what is happening.

**Reflections**

**Testing Techniques Employed**

The software testing technique used for this project was unit testing. Using JUnit 5 in the Eclipse IDE, I performed JUnit tests and checked coverage. This testing method checks that individual methods, classes and functions worked as expected within my code. Here is an example from my work in the past few weeks that shows the results from running coverage as a JUnit test:

A screenshot of a computer

Description automatically generated

**Other Techniques**

There are many other software testing techniques that were not used for this project. One example is integration testing, where you check that different pieces of your code work well together without errors input (Hambling et al., 2019). Another example would be system testing, where you check that your code behaves as expected in a certain environment, or within the intended system. Acceptance testing is another, where a user checks and verifies that the product meets the given criteria. Regression testing is a type of testing that is done to make sure that things still work as expected after changes have been made to the code input (Hambling et al., 2019).

**Uses and Implication of Techniques**

Integration testing would be done a bit later in a software development project, once there are several pieces of code to check that were sometimes written by different people. You would want to make sure that the pieces all work seamlessly without errors and that the code still works as expected as pieces are added to a project. System testing would be done next, after the integration testing to ensure that the code works as expected in an environment. This step makes sure that the program can run in the environment it is being created for. Acceptance testing would come after system testing. In this stage, we are making sure that what was created matches what the user is looking for by having them verify everything is correct and works as they expected. Finally, regression testing is a type of testing that can occur throughout the coding process and makes sure that after changes are made to your code, it still works. After changes are made, tests should be re-run to double check that everything is still covered and runs smoothly.

**Caution**

While working on this project, it was important to exercise caution. Care must be taken to make sure that the code is written without errors and meets the requirements set forth. It is important to consider how each part of the code will interact with other pieces, and this is easiest to do while initially writing the code, instead of trying to go back later and make things fit together input (Hambling et al., 2019). It was important to keep in mind, for example, that the Contact class will be interacting with the ContactService class. Instead of writing them as completely independent classes, keeping in mind that they will be talking to each other, and using variable names and methods that take that into consideration will make things easier in the long run.

**Bias**

For this project, bias is another thing that had to be considered. It is not optimal to write and test your own code. It can be very hard to identify errors and edge cases in your own work, and it tends to be better to have someone that did not write the code do the testing of it input (Hambling et al., 2019). Flaws in your own work can be difficult to detect, and it is important to keep that in mind. For this project there was no one else to test my code, so knowing that there is bias in the code is something to acknowledge. To overcome this to a degree, you need to change your mindset from making code work to figuring out where it doesn’t.

**Discipline**

Finally, discipline is an important principle in software development. Looking for short cuts or cutting corners can lead to code with significant errors input (Hambling et al., 2019). If defects are not managed appropriately, the technical debt increases input (Hambling et al., 2019). Technical debt is the potential costs that can accumulate for a project as a result of using quicker, cheaper solutions. This is not necessarily the upfront costs of a project, but the costs over time that result from using less robust code. As a practitioner, I plan to avoid technical debt by writing code that can be expanded, is sustainable, and follow best practices whenever possible instead of looking for quick fixes.

**References**

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.