**7-2 Project Two Submission**

Evan Bush

Southern New Hampshire University

CS-320: Software Testing, Automation, and Quality Assurance

Dr. Karl Lewis

February 19, 2023

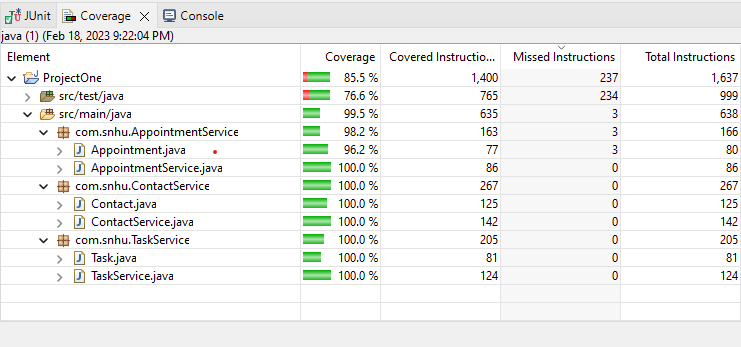
**7-2 Project Two Submission**

**Summary: Alignment to Requirements**

The main extent of my testing approach is aligned with the software requirements based on my test classes running a test or two for each of the specific requirements listed in the assignments. My approach also was to follow a test-driven development method for creating these services. I created all of my tests before writing any actual code for the Contact.java, ContactService.java, Task.java, and TaskService.java classes. This approach made me focus on creating only what was necessary to pass each test and ensuring I only wrote the necessary code to handle each requirement.

**Summary: Effective Tests**

The overall quality of my JUnit tests in the contact service and tasks service is good. This is because my test coverage for all files was 99.5%. This coverage rating does not account for all paths along each decision tree, so it still has some more testing that could be performed.

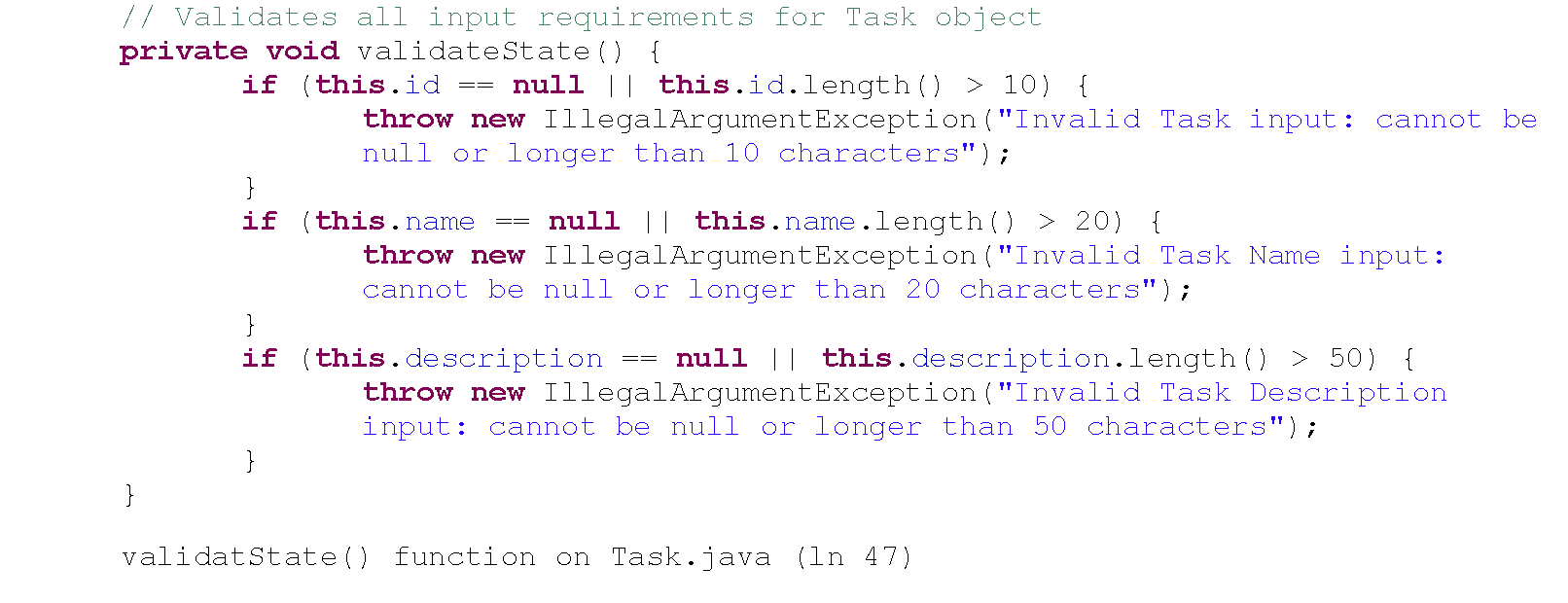


**Summary: Technically Sound Code**

I ensured my code was technically sound by making sure every test passed without any issues or problems. The tests present on lines 12 of ContactTest.java and lines 13 of TaskTest.java also made sure each class was built properly by instantiating a new object with the correct variables and calling them back with their respective getters. This helped make sure the objects were accepting the right inputs and that the object could be created properly.

**Summary: Efficient Code**

I made sure my code was efficient by limiting the amount of code needed to create quality work. For example, on lines 63 of Contact.java and lines 47 of Task.java, in my validateState() function,



I could have created a more complex way to make sure each input was correct or duplicated code to have each validation present in the constructor and setter methods, but by creating a separate private function within the object, I was able to reduce the number of lines of code needed to handle the main requirements. For example, the function would work through each validation, throwing an error when needed and halting progress at that point. This kept my code simple and not overly complex.

**Reflection: Techniques Employed**

The main technique I employed was following the test-driven development method. I wrote the unit testing before writing any of the code. This helped me focus on only writing code that was necessary for the requirements to be met. Although this isn’t exactly a testing technique, it helps me employ portions of some techniques I might not use in scenarios like the past 3 milestones. For example, static testing. These milestones are based on unit testing and executing code, but since the test-driven method involves reading and understanding the code you are writing based on already written tests, it forced me to read and manually review each line of code as I wrote it – which is a key aspect of static testing. Other techniques used surround types of dynamic testing and specifically ways to handle unit testing. For example, equivalence partitioning was used to verify input validation. While I didn’t have a specific test per constructor argument that included a null value, a correct value, and an incorrect value, I had multiple tests that checked for them. The initial tests to make sure we can create the objects under test showed that correct inputs will work. Then for each argument, I had tests to ensure an error was being thrown when a null value was present or the character length was incorrect. This prevented us from having to check every single character length that could be in question.

**Reflection: Other Techniques**

The techniques I did not use were most of the static testing techniques. This is because static testing mainly involves different types of reviews for documentation, requirements, and user stories. There were also black box testing techniques I did not use throughout the milestones. One of the techniques was boundary value. Since we were not dealing with integer inputs and we employed the equivalence partitioning technique, I didn’t see a need or ability to test the boundary input and the input above and below it. While working with the Date object, we could have done this, but, after reading the documentation for the class, it should throw an error if the month and day fields are out of the accepted bounds. This also could be handled through a try-and-catch statement in the user interface portion of the code that would accept inputs.

**Reflection: Uses and Implications of Techniques**

When it comes to test-driven development, the practical uses and implications for different types of projects and situations are all the same. It is a development method to help ensure the developer is writing efficient code with fewer bugs and errors. Since the developer is writing the tests first, they can focus on writing code that will only handle each requirement and nothing else.

The manual review of the code being written has the benefit of providing the developer with a better understanding of what he is creating. This can assist them in building better documentation by naming their variables, detailing their methods, and making an overall better project that others can easily understand and work on together.

The practical use of equivalence partitioning is to limit the amount of input testing a developer or tester has to perform through their unit testing. There could be thousands or more possible combinations for some tests. The process of testing all possible values is called exhaustive testing and can be physically impossible to accomplish at one hundred percent success. Equivalence partitioning helps limit the test to categories or chunks of possible inputs. For example, an input of age to determine if a user is 21 years or older to access a brewery web page, the tests can chunk the possible inputs into a too-young or an appropriate age category. They will only need to test a number below 21 or a number at 21 or higher. This saves them from having to test up to 100 inputs.

The full process of static testing has tremendous practical use while working through various development projects and situations. Reviewing everything outside of the code from the birth of a project will ensure errors in documentation and requirements will be handled before a developer can start the actual development of the project. This can save on the exorbitant cost and time needed to fix errors in the later stages of a project.

Finally, the last black box testing technique mentioned has great implications in many projects and development situations as well. Boundary analysis can be used to provide more focus on input testing alongside equivalence partitioning. In my earlier example, we can choose values that are as close as we can get to our boundary values. For instance, the accepted values can be between 0 and 100 for the ages the user can input. We can test the inputs of -1, 0, 1 and 99, 100, and 101 to further increase our depth of testing without turning it into an exhaustive practice.

**Reflection: Caution**

The mindset adopted was to follow test-driven development. When I typically take on a project, I usually build out each section based on the requirements and then run my tests through the main function. This process usually takes a while to set up and involves a lot of rework of already written code to fix errors or issues. Writing the tests first, helped keep me focused on ensuring I was writing simple code that handled passing each test instead of getting carried away with writing complex functions. This approach promoted a sense of staying cautious and playing it safe throughout the entire process.

**Reflection: Bias**

I eliminated any bias while reviewing my code through the test-driven development method again. Writing the code after writing the tests ensured that my testing was only there to help me as a developer meet the requirements set by the client or project. If I did it the other way around, I can see myself being biased toward my code. For example, I could have tailored some of the tests around the code I already wrote to ensure a passing result. This could mislead the entire testing report.

**Reflection: Discipline**

Being disciplined in the commitment to quality as a software engineering professional is extremely important. When an engineer cuts corners when testing or writing their code it can lead to costly errors later in development. For example, they might not set up testing properly and when changes take place in the code base, tests will provide false positives. This will eventually lead to a product being released with defects and bugs. My plan to avoid technical debt as an engineer is to not run and hide from testing, regardless of how mundane or boring it could be in a larger project. I would prefer to spend my time creating effective tests and constantly testing updates or new code than have to dig into thousands of lines of code to find where a bug or error is hiding.