**Grand Strand Systems**

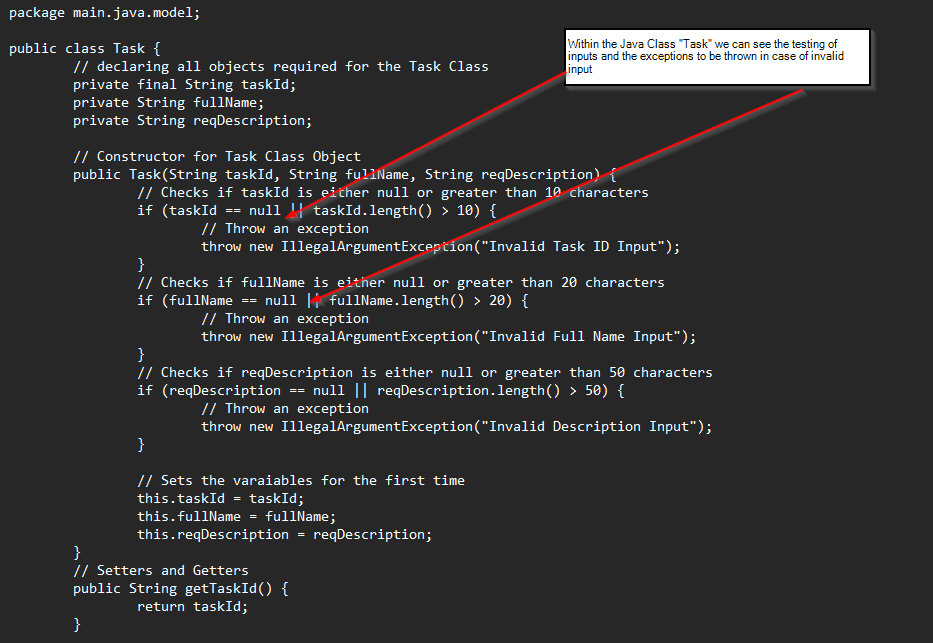
*Software Testing Summary Report*

Table of Contents

|  |  |
| --- | --- |
| Summary of Feature Code Testing | Page 3 |
| Reflections: Techniques and Mindsets | Page 5 |

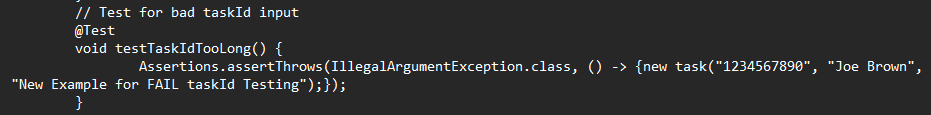
**Summary of Feature Code**

The testing approach and software requirements were aligned by the specifications elicited by the customer and placed on the assignment guidelines. Given that our assignments were only meant to create specific Java Classes, we implemented unit testing using Jupiter which is built for Java. Given that each student oversees their own code and testing, we, each, acted as the developers, testers, and documenters. Each method, or function, that was written in the classes were tested to ensure that inputs would be valid and that any exceptions like long input or null input would be caught and a message be sent to the user. Given that this is only Unit Testing, there were no users (alpha or beta) to run the code, this will likely occur later during the System Testing phase. The decision coverage percentage of the 3 different Modules is 65% from the Jupiter Unit Tests themselves; however, 100% of the requirements were tested because of the exception cases created within the Module Classes.

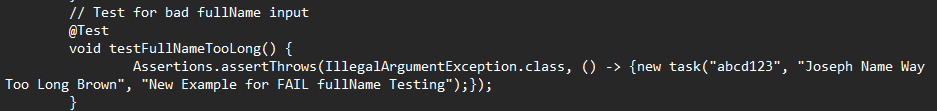


Furthermore, we can see that these exceptions are supplemented with test cases in the Jupiter Unit tests that go deeper into exception handling.

**Test Case #2: Invalid Input**



**Test Case #3: Invalid Input**



**Test Case #4: Invalid Input**

A screen shot of a computer

AI-generated content may be incorrect.

Using fundamental coding principles like commenting, creating constructors, creating setters, and creating getters were the basis of producing technically sound code. Furthermore, research was done to select the method by which the Contact Service and Task Service would hold the information that would be called by their respective IDs. HashMaps were chosen because of their dictionary behaviors; hashing to store key-value pairs in their own buckets within memory. The hashing allows for a Big-O complexity notation of O(1) which allows for a consistent execution time of the code. The Map library also allows the use of some really great functions like put() and remove() which makes the code more technically sound.

**Reflections: Techniques and Mindsets**

The testing techniques used in each of the milestones included Assertions. Specifically, assertTrue for the valid inputs and assertThrows for the invalid inputs. The assertTrue was used such that the testing cases would have a clear distinction between “good” and “bad” data setting up a Boolean statement and outcome where this true or false mechanism is used to reason the software correctness.

**Example: assertTrue**

A screen shot of a computer code

AI-generated content may be incorrect.

**Example: assertThrows**

A computer screen with text

AI-generated content may be incorrect.

Although the milestones did not implement test oracles from an external source, the valid inputs were chosen based off the requirements in the Guidelines and Rubric.

The techniques that were not used in the milestones included tagging/filtering tests and conditional test execution. Tagging/filtering tests is good for creating a wide coverage and then condensing or focusing that scope such that the requirements are met. Testing is the most expensive and time-consuming phase of software development, so it is prudent to shorten where possible. Conditional test execution is another way to select tests such that the immediate needs are met, and future feature deployments are not considered until absolutely necessary. Furthermore, nested testing was not completed in this case because the requirements were not that numerous. Nested tests are good when a developer/tester need to group the testing for a specific unit test or maybe components for an integration test. Nested tested along with conditional test execution can be very useful to isolate a specific bug, or a set of them.

As the software tester, I needed to look for errors and flaws in the code developed. This was difficult because of the personal bias within my own mind. As I was developing the code, initially, once it had a sound logical structure, it was not easy to shift the mindset that what I had built was sub optimal or subpar because of the work that was put into the project. Therefore, more time and caution were needed to examine the code and build the necessary Jupiter Unit Tests around the IF statements to ensure that the decisions being made and assumed were correct.

As a software engineering professional, it is important to not cut corners because of the cascading risk effect that bad, or poorly tested code, can have on a company, consumers, and possibly society. The risk in loss of money, consumer trust and even human life is a heavy and devastating burden to bear and therefore a strong discipline to not cut corners and consider every possibility in software development is paramount. By following the best practice guidelines like separating duties (creating independence) between testers/developers, creating decision tables, carefully planning/eliciting software requirements from stakeholders, and being wary of the historical challenges, and/or failures, of previous software development projects will help me to avoid technical debt as a practitioner in the field.