Project Two

My tests were written to cover almost 100% of the code requirements. The requirements were explicit and that is reflected in the tests I wrote. For example, an ID string within any class can’t be longer than 10 characters. That is easy to test for. Try instantiating a task with an ID string longer than 10 characters. If it can be done, then the test fails. Otherwise, it passes. The only explicit requirement that I didn’t fully cover in my code is the unique ID creation for either the contact or the task classes. What I did instead was assumed that the user would enter an ID that will be unique. The reason I did this was because creating tests for enforcing uniqueness would be complex. Checking for uniqueness would be resource-intensive or it would require complex algorithms to enable scaling as the theoretical database of contacts or tasks grew in size.

Additionally, there are more common-sense rules around creating tasks and contacts that are not part of the listed requirements. For example, there is no requirement ensuring that an ID is exactly 10 characters. Someone could enter 9 characters or just 1 character and it would be valid. But this presents the impossible question: when are we done testing? We can never cover all test cases.

My Junit tests were effective on the basis of coverage percentage because they all had 100% coverage. But code coverage I think is not always a good indicator of robust software. If I only write one basic test and it passes, I can say that I have 100% code coverage. However, in these assignments, my tests were line-by-line equivalent to the listed requirements.

One way I ensured my code was technically sound was creating a function to check the safety of values entered into the constructor for task and contact. If you want to create or update a task or contact in the task list or contact list, you must enter values that are checked for validity. But even this is not a foolproof solution. The way I wrote the code, you could create a task or contact outside of the list and update it with values that don’t pass through verification. However, it couldn’t be added to the list. This could create ambiguity in future code where tasks or contacts are created outside of the scope of the list.  
 Another way I ensured my code was technically sound was by making it testable. I could have wrote functions that created or updated tasks or contacts but didn’t return anything. How would I have tested those functions? It would take more work. Instead, I made the functions return strings or null like in TaskService.java on line 77 the updateTaskName() function. If the function executed properly, it should return a string. If it didn’t, it should return null. This gives the test an easy way to determine if the function is working as intended.

Throughout the modules I’ve used equivalence partitioning with invalid input as a primary technique. The language in the requirements says that a class variable shall not be updatable. I decided to write tests and provide invalid input to each one. The test passes if the invalid input causes an instance of the class to throw an error or return null.

I didn’t use equivalence partitioning with valid inputs. I assumed that valid inputs are all valid. I also didn’t use boundary analysis. I maybe should have included that however, since I didn’t test for any empty string values. I think that boundary analysis might be especially useful when working with math equations and numbers. From my limited experience, I suspect that equivalence partitioning with valid inputs might be especially important when working with APIs.

I demonstrated caution by creating functions that checked if inputs were valid within each class. When you’re testing, you have to think creatively. Users can’t read your mind – they may try to use your code in ways that you didn’t think of. They may try inputs that you don’t expect. Creating a valuesSafe() function was important to ensuring that the code was used as it was intended by the requirements.

Software developers think in terms of making sure their code does what it’s supposed to do. But limiting testing only to positive testing leaves code open to lots of bugs. The requirements were simple enough that I didn’t write positive tests. That’s probably not great practice in real life. To be thorough, I maybe should have used limit testing to check the low and high limits of valid inputs. In my testing approach I focused only on negative testing. I thought the code should be able to handle incorrect input gracefully.

Personally, I see the importance of being disciplined for two reasons: to at least keep my job, and to prevent harm or damage to ordinary people. It’s important to stay disciplined to keep your employment, but sometimes employment is at risk based on your supervisor’s lack of discipline. I hope that I never cut corners based on a deadline or to boost company profits for shareholders/executives. I hope that I always stay disciplined in writing safe code that doesn’t bring harm to the average person that may use it.