To ensure my testing approach was aligned with the software requirements I checked the requirements, wrote code for functionality, and wrote JUnit tests to test functionality and operating restrictions. Both assignments, contact service and test service had a list of requirements for functionality. For example, some of the requirements for the Test Service class were “The task object shall have a required name String field that cannot be longer than 20 characters. The name field shall not be null.” After collecting the requirements for the task class I wrote the code for functionality.

| public void setTaskName(String updateTaskName) {  if (updateTaskName == null || updateTaskName.length() > 20) {  throw new IllegalArgumentException("Invalid entry for Task Name");  }  this.taskName = updateTaskName; } |
| --- |

When writing the code for that specific item, I included all restrictions in the code to ensure the customer's requirements were met. I did this for each customer requirement before moving on to writing the JUnits test.

| @Test void setTaskName() {  Task testTask = new Task("123", "Add Task",  "This is a test to add a new task");  testTask.setTaskName("Changed Name");  assertEquals("123",testTask.getTaskId());  assertEquals("Changed Name", testTask.getTaskName());  assertEquals("This is a test to add a new task", testTask.getTaskDescription()); } |
| --- |

| @Test void taskNameNull() {  Assertions.assertThrows(IllegalArgumentException.class, () ->  new Task("123", null,  "This is a test to add a new task."),  "Expected taskName to outside requirements"); }  @Test void taskNameLength() {  Assertions.assertThrows(IllegalArgumentException.class, () ->  new Task("123", "Add Task1234567890123",  "This is a test to add a new task."),  "Expected taskName to outside requirements"); } |
| --- |

While writing my JUnit test I followed the same steps as writing the code. I made a test that was targeted at testing each requirement and functionality. Using this method of writing the JUnit test makes sure that an 80% test coverage was reached.

To ensure my code was technically sound and efficient I used method names that were related to their functionality and was consistent in the whitespace I used for each method. To help with potential syntax errors I followed the suggestion from the builtin linter that comes with ItelliJ IDE. To ensure efficiency I wrote tests for all functionality and user requirements. The following code block is an example of the whitespace and naming convention I used throughout the entire program.

| @Test @Order(1) void testTask() {  Task testTask = new Task("123", "Add Task",  "This is a test to add a new task");  assertEquals("123",testTask.getTaskId());  assertEquals("Add Task", testTask.getTaskName());  assertEquals("This is a test to add a new task", testTask.getTaskDescription()); } |
| --- |

The 3 milestones I completed for this class were my first attempts at writing tests for software. I believe I was overly cautious and thorough in my attempt to reach the testing coverage goal of 80%. I made an attempt to every requirement and method for proper function. Although the code seems fairly simple the amount of work it took me to write tests that covered everything I though necessary was difficult. The most difficult part for me was testing the code without being able to run the code and visually see the results. Without being able to test in ways I was comfortable I felt like most of everything I wrote was not going to work properly. I believe that this helped with my bias.

Bias can be a difficult challenge to overcome, even in my current life safety job I after installing systems I tend to feel like I did everything right. I believe that having someone else double check and test your work is the best way to avoid bias. My lack of experience coding and writing test worked in my favor for these assignments and helped me overcome my bais. I will have to be more conscious with more experience to help avoid missing simple bugs.

Being committed to quality is important when it comes to development. I believe practicing being disciplined with school or personal project will help avoid costly or embarrassing errors in the future. During module seven readings there was many examples of code errors that were released to the live software and cause damage. For example, the explosion of the Ariane 5 is an extereme example of how code bugs can have a negative impact. During this flight a 64 bit floating point number was converted to a 16 bit integer. The conversion failed because the number was to large to store in an integer. This could have been fixed with testing and could have saved the tie and effort that was out into the space flight.