Throughout my program, I used the testing approach of JUnit tests to ensure that my code was aligned to the software requirements. JUnit tests help programmers assert statements and values as rather true or false and checks those tests against the code to see if it executes as it is expected to. I was able to look at the software requirements for this specific project and create tests that checked to make sure the requirements were met. For example, one requirement was that the Contact ID could not be longer than 10 characters and could not be null. Therefore, I was able to create tests that asserted an ID longer than 10 or equal to null as false, which led to an exception thrown. Further, I asserted that an ID equal to or less than 10 is a true and valid instance. I was able to create tests that covered all the cases and edge cases for each requirement. All the tests created passed, which indicates that all the requirements have been thoroughly tested. One cannot just create JUnit tests to check their program, though. It is extremely important to ensure the JUnit tests are effective, too. It is generally considered acceptable for JUnit tests to have a coverage of 80% or higher (Pittet, n.d). I ensured that for all my tests, there was greater than an 80% coverage rate. This helps create an overall sense of security and efficacy within the tests that covered the entire program.

I initially experienced some difficulty and confusion while writing the JUnit tests, but with each milestone I gained experience and confidence that ultimately led me to reach over 80% test coverage for each of my tests. I ensured my code was technically sound through the JUnit tests by checking to ensure that all the requirements were met, and all edge cases were tested and passed. For example, the contact object was to have a phone number string that was required to be exactly 10 characters long, no more and no less. I used JUnit tests to ensure that if a phone number instance was less than or greater than 10 characters, as well as if it was set equal to null, an illegal argument exception was thrown, and the test would not pass. This helps ensure that the code is technically sound by checking that all requirements are strictly met. The following lines are where this criterion was tested:

```
//Test to make sure number is not too long (must be exactly 10 digits)
@Test
void testNumberTooLong() {
Assertions.assertThrows(IllegalArgumentException.class, () -> {
new Contact("12345", "Tabitha", "Pawlowski", "4844005000000", "10 South Street");
});
        }
//Test to make sure number is not null
@Test
void testNumberNull() {
Assertions.assertThrows(IllegalArgumentException.class, () -> {
new Contact("12345", "Tabitha", "Pawlowski", null, "10 South Street");
});
}
//Makes sure that the code allows a number to be exactly 10 digits long
@Test
void testNumberMustBe10Digits() {
Contact contact = new Contact  ("12345678", "Tabitha", "Pawlowski", "4844005000", "10 South
Street");
assertTrue(contact.getNumber().equals("4844005000"));
}
```

In addition to just being technically sound, the JUnit tests were also created to ensure that my code was efficient. One way the tests made sure the code was efficient was by making sure certain values were able to be updated so that anytime a new value had to be added, an entirely additional task did not always have to be created and added to the list. For example, the task service had to be able to test that the task's name and description could be updated given the value of its ID. The following code checks that these requirements are met:

```
//Test to make sure you can update a task's name given its id
@Test
void testUpdateTaskName() {
newTask.addTask("12345",  "Tabitha Pawlowski", "Execute a Test");
newTask.updateTaskName("12345", "Taylor");
for (int i=0; i < newTask.tasks.size(); i++) {
if(newTask.tasks.get(i).getTaskId() == "12345") {
assertTrue(newTask.tasks.get(i).getTaskName().equals("Taylor"));
}
}


}
//Test to make sure you can update a task's description given its id
@Test
void testUpdateTaskDescription() {
newTask.addTask("12345",  "Tabitha Pawlowski", "Execute a Test");
newTask.updateTaskDescription("12345", "New Test Description");
for (int i=0; i < newTask.tasks.size(); i++) {
if(newTask.tasks.get(i).getTaskId() == "12345") {
assertTrue(newTask.tasks.get(i).getTaskDescription().equals("New Test Description"));
}

}

}
}

}
```

The software testing technique that I employed in this project was the use of JUnit testing. JUnit testing is an open-sourced testing technique that can be used in Java to ensure that the code written meets the requirements and expectations set forth for it. A programmer can write tests that assert statements to be true or false, and the result of that test being ran tells the programmer whether the code behaves expectedly or does not meet the requirements given. Within these JUnit tests, I used Assert True and IllegalArgumentException statements to check if the code was executed as it should have. If a test passed, I knew it was likely that the statement tested was correctly programmed. There are other software testing techniques that were not used in this specific project. One example of a testing strategy not used is integration testing, which incorporates, or integrates, more and more small pieces of code into the test until the entire program has been tested. This can be good for small projects or when you need to test a project as you develop it. In addition, system testing is a technique where a near-finalized project is looked at for its efficiency and effectiveness. This technique should be used in a situation where it is a small, easy, and predictable project where just enough testing at the end would be appropriate, and where the program's efficiency is of the utmost importance. JUnit testing, on the other hand, is useful in a wide range of scenarios. JUnit testing allows concurrent testing with programming, which allows testing to be done as the program is being coded, not leading to massive amounts of work needed at the end of the project. In addition, testing does not need to be revised every time a change is made to the code since each JUnit test only tests a small statement in the program, instead of the entire program such as in other testing techniques.

During this project's early stages, I was a little confused about what the JUnit tests needed to accomplish to become effective and secure. As I continued, I realized that creating multiple test cases for one criterion was most likely necessary to check all scenarios where the instance could or could not be valid. For example, when the appointment ID cannot be longer than 10 characters, I must test that the ID is able to be less than or equal to 10 characters but cannot be over 10 characters or set equal to null. At first, I expected my program to run as it should, but was not cautious enough about the edge cases. As I added more JUnit tests to my program, I increased my coverage, better ensuring that my program had been thoroughly and wholly tested. I initially did not appreciate the complexity of the code I was testing by just assuming that if the ID was not over 10 characters or not set to null that it would work as expected. However, it is equally as important to make sure the program works correctly when the ID is less than or equal to 10 characters, too. By acknowledging the complexity of the code itself, I realized what tests were needed to ensure the code was correctly written.

From a tester's standpoint, it is easy to think that the testing done is "good enough" when in fact much more testing can, and should, be done to ensure that the program thoroughly compiles and executes the way it is expected to. It is important to get rid of this bias when testing your own code. As I realized through missing rubric points on the first and second milestone, I had to stop assuming that my code was “good enough” and add extra tests to actually prove it was “good enough”. I realized by the third milestone that adding additional tests to check for all edge use cases will help me reduce this bias and have better overall test coverage for my program. Without this set of extra, outside perspective “eyes”, it would be even more difficult to realize and reduce this bias, making it an extra important issue to consider.

One of the most important aspects of being a software engineering professional is maintaining discipline in your commitment to quality. Quality of code is how one knows that the program is effective and correct, working the way it is supposed to, and is free of any bugs or weaknesses. When testing code, it is important to not cut corners that could lead to bugs or defects being left unnoticed and not fixed. Technical debt occurs when short cuts are taken that later lead to a loss of resources, time, and money. As a skilled and honest practitioner in the field, it is important to reduce your amount of technical debt as much as possible. For example, when developing a project, one might be tempted to limit the number of tests they create to save time and energy. However, as the project nears completion and must be tested before deployment, several bugs may arise that force the project to be pushed back and reworked. This will cause the team or business a heavy loss in resources, time, and profit that could have been prevented if no initial short cuts were taken. To be a reputable software engineering professional, one must put in the appropriate work as soon as possible to prevent major issues from arising.

References

Atlassian. (n.d.). *What is code coverage?* https://www.atlassian.com/continuous-delivery/software-testing/code-coverage#:~:text=With%20that%20being%20said%20it,fairly%20low%20percentage%20of%20coverage