My approach to testing the software aligned with the software requirements by ensuring that every requirement was tested for each parameter. Each class had its constructor tested with the format:

void testValidContact() {

Contact testContact = new Contact(validContactID, validFirstName, validLastName, validAddress, validPhone);

assertEquals(validContactID, testContact.getContactID()); assertEquals(validFirstName, testContact.getFirstName());

assertEquals(validLastName, testContact.getLastName());

assertEquals(validAddress, testContact.getAddress());

assertEquals(validPhone, testContact.getPhone());

}

Afterwards, each parameter was tested with both invalid and null data to ensure that they did not accept these bad inputs.

void testInvalidContactID() {

Assertions.assertThrows(IllegalArgumentException.class, () -> {

new Contact("Illegal Contact ID", validFirstName, validLastName, validAddress, validPhone); }); }

Update functions were also tested when relevant, to ensure that they worked as intended. The service classes are tested similarly, with additional tests to ensure the list functions were working as intended:

void testDeleteContact() {

ContactService.addContact(validFirstName, validLastName, validAddress, validPhone);

ContactService.addContact(validFirstName, validLastName, validAddress, validPhone);

ContactService.deleteContactByID("1");

// Ensure the task was deleted

Assertions.assertThrows(NoSuchElementException.class, () ->

ContactService.getContact("1")

); }

My JUnit tests averaged 80% for the three projects. Rather than spending the time to write tests that covered 100% of the code, I focused on ensuring that the code met requirements, was technically sound, and that it contained as little bloat as possible. Although 100% is realistic in small codebases such as Project One, the endeavor is far less feasible in codebases that are exponentially larger. 80% coverage tells us that most of the code is accounted for. Looking at specifics that are not covered by the tests reveal that the only lines untested are messages from exceptions caught in try-catch blocks.

The above examples show the code is technically sound, as the other unit tests for each component test each parameter of every class. By ensuring the code adhered to the requirements, then testing each of the functions that were specified, the code can be deployed with confidence that it runs without errors. I ensured efficiency by reusing the helper functions getParameter in the lists of the other service class functions, and by keeping the tests concise to reduce clutter and redundancy in the codebase.

In this project, I employed unit testing and integration testing. Unit testing is the process of testing each individual module alone. This process exposes errors in the module, so they can be fixed before it is integrated into the system. Examples are the unit tests for each parameter, contact, task, and appointment. Testing each of the classes by themselves ensures that they cannot accept bad data, and when integrated with the appropriate service class, should work as intended. Integration testing is testing the individual module’s interactions with each other, ensuring that they perform as expected when exposed to the system. I did employ security testing, acceptance testing, or performance testing. Security testing attempts to cause a malfunction in the application, testing that it is secure against common attack vectors before being exposed to users. Acceptance testing allows the client direct interaction with the software so they can verify it meets their requirements and user needs. Performance testing tests the performance of the application, monitoring how fast it processes information and ensures it is not using an excessive amount of system resources. These tests are practical in most development lifecycles to ensure that a piece of software meets every requirement, both from a technical and client perspective. Utilizing all these of these testing types allows a client to verify that a piece of software works as intended, with the added benefit of being able to incorporate future features through modules that will not interfere with current architecture, that the software will use manageable amounts of resources and is responsive enough for users, and that it satisfies the intent of the client, and that it is secure for both users and clients.

For this project, I adopted a mindset of caution. Double-checking the requirements and examining the specific lines the tests covered allowed me to continue with confidence that I was creating a piece of software that adhered to the assignment. Caution is essential to role of a developer and tester, as both need to ensure not only that code works as intended, but that future developers can also verify the functionality of the code. Moving one step at a time to create the overall system builds a strong and robust infrastructure with information that verifies it is working as intended. With systems as interconnected as they are in the modern day, caution is paramount to success. If contact was not thoroughly tested, tests for contactService might have failed, forcing an investigation into what was causing the failures. Worse still, they might have passed without indicating a system error, exposing a defective piece of software to the client and users. Bias for your own work or prefered way to solve a problem can lead to inconclusive tests that do not accurately audit the software. For example, I turned in the assignment in module three without writing null tests, needing a second pair of eyes to verify that my tests were an accurate reflection of the requirements. Getting someone to proofread the work allows us to be confident that everything aligns with the software requirements, better if the automated testing is written well and covers those bases for us alongside a form of static testing. Discipline is a key element in development, as our world is increasingly connected, exposing the software to many potential bad actors and users. Without discipline and code standards, we risk creating a subpar infrastructure that is difficult to maintain and may put users or stakeholders at risk. Testing is a tedious, boring process, but it is also a vital way of ensuring that code will meet client requirements and allows those that come after to clearly see the design choices so they can maintain or modify the software without creating more issues.