While creating the models, classes, and tests, I played very separate roles. I tried to be a diligent programmer when creating the models and classes. However, when creating the tests, I took more of a devil’s advocate approach, trying to display all issues with the coding.

One way of doing this was testing the boundaries of parameters. For instance, if the parameter can’t be null, greater than so many characters, and can’t be an empty string then I would have a test such as this.

if (fullName == null || fullName.length() > 20 || fullName.equals("")) {return false; }

I would then use the method passing 4 different parameters. The first would be a valid argument, next a null argument, then a string that is too long, and finally an empty string. This would test all possible scenarios for the method and its parameters.

The Task and Contact classes each had more than 3 parameters in their constructors. With multiple ways to fail each parameter multiplied the possibilities. This is why I found it much more efficient to test only one valid and one invalid constructor and then have full coverage testing the getters and setters.

To ensure the code is sound I tested the intentions of the code to make sure that it accomplished what was intended. For example, to test the method in ContactService that deletes an ID I added three contacts and then deleted one. I then made sure the size was correct. Lastly, I searched for the deleted contact to make sure it was gone. This made sure that the adding and deleting was all done correctly.

@DisplayName("Test deleteContact")

@Test **void** testDeleteContact() {

String firstName = "Greg";

String lastName = "Hoffman";

String phoneNumber = "1234567891";

String address = "123 Main Street";

**boolean** testBool = **false**;

ContactService test = **new** ContactService();

assertTrue(ContactService.contactList.isEmpty());

test.addContact(firstName, lastName, phoneNumber, address);//object ID 0

test.addContact(firstName, lastName, phoneNumber, address);//object ID 1

test.addContact(firstName, lastName, phoneNumber, address);//object ID 2

assertEquals(3,ContactService.contactList.size());

test.deleteContact("1");

assertEquals(2,ContactService.contactList.size()); **//test one**

//loop through to look for ID

**//Test 2**

**for**(**int** i = 0; i < ContactService.contactList.size(); i++) {

**if**(ContactService.contactList.get(i).getContactID() == 1) {

testBool = **true**;

}

}

assertFalse(testBool);

}

One of the ways I chose to ensure efficiency is by first making sure that if a string fails to update that the string will still be the same as before there was an attempt.

tempTask.updateTasks("1", fullName, "New description"); //bad ID

*assertNotEquals*("New description", TaskService.*tasks*.get(id).getDescription());

*assertEquals*(fullName, TaskService.*tasks*.get(id).getName()); //original string

To further test this, I also made sure that objects where not instantiated when they weren’t called. This saves computer resources and therefore time as well.

*ContactService test =* ***new*** *ContactService();*

*assertTrue(ContactService.contactList.isEmpty());*

In this course, I applied unit testing and static testing to the three modules, both of which are types of whitebox testing. Static testing involves analyzing the code and comparing it to specifications to detect bugs. I frequently used this method to locate and correct problematic code after one of my JUnit tests failed. While wrote the class methods, I aimed to align the logic with the specification document. During unit test creation, I re-examined the specifications to identify potential errors. Occasionally, a JUnit test would fail, prompting me to review the static code to determine the cause. In one instance, I used an incorrect attribute within a logic branch, which led to the test failure. I also learned that ensuring an exception is thrown when expected is insufficient; catching the correct and expected exception is equally crucial. For example, I was testing that a constructor would fail due to invalid input, and although an error was thrown, it wasn't the one I intended. Consequently, in Module 5's assignment, I began verifying the exception type to ensure it matched my expectations and prevented false positives.

Modules 3 and 4 involved minimal system testing and no integration tests, whereas Module 5 included some system tests. Each assignment required us to create a base class and a service class that interacts with it. In theory, the base class could be tested through the service class, treating it as a system. I did some of this in Module 5 but not in Modules 3 and 4. Integration testing involves testing the entire application by combining all systems. Thus, system and integration tests could be equivalent. Ideally, I could validate that the TaskService, AppointmentService, and other classes function well together at their integration layer, though I haven't done this. All my JUnit tests were manually run, I didn't employ automated testing. Ideally, a service could run the JUnit tests after each build event in Eclipse, which is more significant for large applications. Additionally, no security scanning was performed. Libraries and components, such as SPRING or Java components, need to be scanned for security vulnerabilities. However, since I'm not extensively using libraries, databases, or shared components, security testing can wait.

Automated tests are crucial for large applications and should be integrated into a continuous integration pipeline, especially if continuous delivery is practiced. Automated tests can occur at the build stage, pre-deployment, or even post-deployment. My projects typically run unit and system tests at build and integration tests pre-deployment. Unit testing is always beneficial, helping to prevent small issues from becoming major problems and hardening the code against missed logic branches and specifications, such as null values or maximum string lengths. System and integration tests become increasingly important as the code base grows and interconnected components multiply. Security testing is also vital, particularly where data is concerned. Currently, there are no databases or user inputs in the system, so protection against improperly formatted input, such as SQL injection or buffer overruns, isn't necessary.