While developing the model and service classes and testing them with JUnit, I took on two distinct roles, those being as a programmer and a hacker, which is to say that at each step of the way I made precise efforts to detect for weaknesses within the code I created as a form of testing.

I utilized Unit Tests to explore the limits of the specification. For instance, the Task specification states that the full name must not be null, must not exceed 20 characters, and is required. To thoroughly test this specification, at least four different test cases are necessary.

*if (fullName == null || fullName.length() > 20 || fullName.equals("")) {return false; }*  
  
I tested a valid version of the name, a null string, a very long string, and an empty string. Two samples are below:  
  
*Assertions.assertThrows(IllegalArgumentException.class, () -> {tempTask.setName("");*

*Assertions.assertThrows(IllegalArgumentException.class, () -> { tempTask.setName("This name is incredibly long and why is this a name, anyway");*

I didn't aim for 100% coverage of the Task and Contact classes. Given that the constructors had three or more parameters to set, I didn't find it necessary to test both valid and invalid constructors for each parameter. Instead, I focused on achieving 100% coverage for the getters and setters, along with testing one valid constructor and one invalid constructor.

To ensure the software's reliability, I aimed to test "in the spirit of the code." For instance, in the ContactService class, there was a method to delete an ID. This method could be tested in several ways: The first being to add three objects to the collection, delete one, and then verify the collection now contains two items. The second was to search the entire collection to ensure the deleted item is no longer present.

Relying on only one of these methods could allow for errors. If only the collection size is checked, the wrong item might be deleted while the test still passes. If the search method is used alone, the test might pass even if all items were accidentally deleted. Therefore, I tested for both scenarios.

*@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);*

*}*

As mentioned above, it's not sufficient to simply check the size of a collection after a deletion; it's crucial to verify that the specific item was actually deleted. Additionally, a more thorough approach would involve ensuring that the remaining objects are still intact and as expected, though this was not tested in my code.

However, I did ensure that objects were instantiated only when requested.

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

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

I also checked that after a failed string update the string was still as before the 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*

For the three modules required in this application, I employed unit testing and static testing, both forms of whitebox testing. Static testing involves examining the code and comparing it to the specifications to identify bugs. I used this technique to locate and fix problematic code after one of my JUnit tests failed.

While writing the class methods, I aimed to design the logic to match the specification document. During unit test creation, I re-read the specifications to identify potential coding errors. On several occasions, JUnit tests failed, prompting me to inspect the static code to determine the cause. In one instance, I had used the wrong attribute within a logic branch, leading to a test failure.

I also realized that simply ensuring an exception is thrown when expected is not enough; catching the correct and expected exception is crucial. For example, in one case, I asserted that a constructor would fail due to bad input. An error was thrown, but it was not the one I anticipated. Therefore, I began evaluating the exception type to ensure it was the expected one, preventing false positives in my tests.

The service classes have very little system testing and no integration tests. The specifications created a model class and a service class that drives and interacts with the model class. In theory, the model class can be tested through the service class, treating this as a system test. I did some of this in later development, but the missing tests should be considered technical debt.

Integration testing would assess the entire application by bringing all the systems together for testing. In this case, system and integration tests would be the same. I could, theoretically, validate that the TaskService, AppointmentService, and other classes work well together at their integration layer, but I have not done this.

Additionally, I did not perform any automated testing; all my JUnit tests were manually run. Ideally, I could have implemented a service to run JUnit tests after each build event in Eclipse. Automated tests are essential for large applications and should be integrated into a continuous integration pipeline, especially if practicing continuous delivery. Automated tests can occur at various stages: build, pre-deployment, or post-deployment.

No security scanning was performed either. Libraries and components need to be scanned for security vulnerabilities. Security testing is crucial, especially where data is involved. Currently, there are no databases and no user input into the system, so protection against issues like SQL injection or buffer overruns is not necessary.

Unit testing is always beneficial, ensuring small issues do not escalate into significant problems. Unit tests can harden the code against missed logic branches (e.g., handling null values) and missed specifications (e.g., ensuring a maximum string length of 50 characters). System and integration tests become more valuable as the codebase grows and components become more interconnected.

My mindset varied in each phase, as mentioned initially. During coding, I aimed to interpret the true intention of the requirements, not just the literal text. For instance, the Appointment class requirement stated:

"The appointment object shall have a required description String field that cannot be longer than 50 characters. The description field shall not be null."

This means there should be a logic test for length and null values. However, the spirit of the requirement implies that a description must exist. Thus, the validation test should also ensure the description is not blank or empty, thus becoming as follows:

*private final boolean validateDescription(String description) {*

*if (description == null || description.length() > 50 || description.equals("")) {*

*return false;*

*}*

*return true;}*

When examining how the Service classes are intended to use the Model classes, I determined that using a HashMap was more efficient than using a List. Instead of writing code to search the collection for an index, I could simply use the HashMap method contains. See below:

Use of a List collection in the remove method

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

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

*contactList.remove(i);*

*}*

*}*

Use of a Hash Map (no looping needed)

***if****(tasks.containsKey(\_ID)) {*

*tasks.remove(\_ID);*

*}*

Rather than basing off the assumption that my code was correct, i tested my assumptions via using the JUnit to ensure that the correct exceptions were thrown, as shown below:

*IllegalArgumentException exception = assertThrows(IllegalArgumentException.****class****, () -> {*

*AppointmentService tempAppt =* ***new*** *AppointmentService();*

*tempAppt.addUniqueAppointment(goodDate, description);*

*});*

*assertEquals("Invalid description", exception.getMessage());*

Avoiding technical debt is a long-term commitment rather than a short-term task. This issue must be revisited regularly, as security threats emerge daily and exploits can be attempted in mere hours. Technical debt isn't always a bug; it can also be a missed use case. Continuously measuring user satisfaction, using tools like UserVoice, addressing issues found by static and dynamic scanning tools such as SonarQube, or improving code styling and quality with tools like Lint, all require significant time investment in commercially sustained software. I generally estimate that no less than 30% of an engineer's time should be allocated to these activities.

Beyond the immediate benefits of JUnit tests, they also serve as indicators of code quality. More mature software often has higher code coverage, which contributes to better sustainability. Low code coverage should be treated as technical debt. Within this app, the JUnit tests acted as a “second set of eyes,” catching several instances of improper copy/paste. To maximize the benefits of unit tests, it is crucial for the coder to set aside pride and ensure that what is intended actually occurs without side effects.