**Summary**

**Alignment with Software Requirements**

The software had requirements for each attribute (variable) of Contact and Task as well as requirements for each method (function) of Contact Service and Task Service that were fulfilled in the code (described below).

***Contact, Task, and Appointment***

Each attribute has requirements related to its length, existence, and/or mutability. The ID variables cannot be changed; they are immutable. The “final” keyword is used in their initialization and a setter method for them is omitted to reflect this requirement. To check whether variable values are null, too short, or too long, conditional statements using the “or” || operator throw exceptions when constructing either a Contact or Task object.

A black background with white text

AI-generated content may be incorrect.

Figure 1 - Part of Task's constructor method

***Contact, Task, and Appointment Service***

The methods in the Service classes need to be able to add, delete, and update Contact, Task, or Appointment objects held in a data structure. To achieve this, an ArrayList is used to hold the objects, “for each” loops iterate through the list to check objects’ ID values, Boolean variables change depending on outcome, and setter and getter methods update and retrieve values.

A computer screen shot of a program code

AI-generated content may be incorrect.

Figure 2 - addContact()

A computer screen with white text

AI-generated content may be incorrect.

Figure 3 - deleteTask()

A computer screen with white text

AI-generated content may be incorrect.

Figure 4 - Example of update() function

**JUnit Test Effectiveness**

I made sure that the base and Service classes each had at least 80% coverage when running their respective test files. This is “considered reasonable” as a coverage percentage for JUnit testing (García, 2017). All tests cover above this average, with the highest example being 100% coverage for Task in Figure 7. Meeting this standard means my JUnit tests covered most of the branches (or scenarios) for each aspect of the tested classes.

A screenshot of a graph

AI-generated content may be incorrect.

Figure 5 - Contact coverage

A screenshot of a graph

AI-generated content may be incorrect.

Figure 6 - ContactService coverage

A screenshot of a graph

AI-generated content may be incorrect.

Figure 7 - Task coverage

A screenshot of a graph

AI-generated content may be incorrect.

Figure 8 - TaskService coverage

A screen shot of a graph

AI-generated content may be incorrect.

Figure - Appointment coverage

A screen shot of a graph

AI-generated content may be incorrect.

Figure - AppointmentService coverage

**JUnit Test Technical Soundness**

I ensured my code was technically sound by refactoring until all tests passed. I used the corrected annotations like @Test and @BeforeEach after importing the JUnit library (and specific modules of it in-code) to organize my tests by the setup and exercise stages (García, 2017). Then, I added assertions to measure whether the tests passed or failed, fulfilling the verification stage of test design (García, 2017).

I also edited Contact and its related classes to include more coverage by considering other possibilities, such as values that are *exactly* the limited length or adding multiple objects to the ArrayList at once. This extra thought and code improved my test coverage and, therefore, my code quality. I also made sure to include both positive and negative scenarios for constructors and setters to test whether they went through or threw exceptions when they were supposed to (García, 2017).

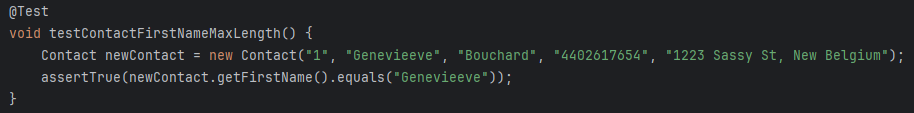


Figure 11 - Example of testing for exact or maximum length

A screen shot of a computer code

AI-generated content may be incorrect.

Figure 12 - Test for adding multiple Contact objects into ArrayList at once

**JUnit Test Efficiency**

One indicator of my code’s efficiency is its quick test runtimes. No test takes longer than a second to complete, fulfilling the goal of speedy execution (García, 2017). An example is shown below in Figure 13, and all other tests had similar execution speeds.

A screenshot of a computer error

AI-generated content may be incorrect.

Figure 13 - TaskTest runtime

I also made my code efficient by considering enough scenarios to achieve 80% coverage without venturing too far into edge case scenarios, given the simplicity of the application. Because the requirements did not mention empty strings being acceptable or unacceptable, I declined to test scenarios for them. Each test was specialized to fulfill the “single responsibility principle” in that it tested for one quality, such as a length that is too long (García, 2017).

A screen shot of a computer

AI-generated content may be incorrect.

Figure 14 - Example of single responsibility principle

That said, my code could be made more efficient with functions that can replace the somewhat convoluted for-each loops and Booleans holding the Service classes together. I found resources online that mention using streams, but it’s hard to use new in-code functionality that I don’t yet understand. Perhaps in using streams instead of the code’s current logic, the number of lines and runtime could be reduced, resulting in cleaner code.

**Reflection**

**Software Testing Techniques**

***Techniques Used***

The code was repeatedly manually inspected by the developer (me), but testing was automated through a sequence of JUnit tests, so it could be said that both manual and automated testing techniques were used at different points in development. I wrote the tests for all milestones based on a format developed in the first one (and referenced from multiple sources), but when tests for the base format failed, further tweaks were needed, and I had to manually trace where the errors originated to then write working tests. For example, in the Appointment classes, the constructor was catching the incorrect date before the setter, and I needed to make an Appointment object with a “good date” before I could test the “bad date” with the setter.

At the heart of the milestones is functional testing. Each test is designed to make sure that its corresponding piece of code does what it’s supposed to do. The function “testContactServiceAddContact()” from ContactServiceTest.java ensures that the ContactService.java function for adding a new Contact object to the contacts list does just that. In checking test execution times to make sure each ran within an acceptable standard, non-functional testing was also lightly performed.

Each constructor, setter, and service function was tested individually for cases involving their basic possibilities (e.g., null values, too-long values, too-short values), so perhaps it goes without saying that dynamic unit testing was a heavily implemented technique. As functions were added, their tests were added and executed, so regression testing was repeatedly used as unit development continued. By repeatedly cross referencing the assignment guidelines (AKA client requirements) with the code and developing until all errors were addressed, static testing was performed.

***Techniques Not Used***

The units were not integrated (save for some crossover here and there with getters and setters), and the system is not currently complete, so neither integration nor system testing has been performed. The client has not yet interacted with the system, and a user interface does not yet exist for it, so acceptance and usability testing techniques have not been used. Because the software is not yet being developed as a system, security is also not being considered, and security techniques like penetration testing are not being used. That said, unit tests that check each function’s ability to properly validate variables *are* valuable components of holistic security.

***Practical Uses and Implications***

Static testing in the form of manual reviews, as informally done here, are practical for finding defects in a system early in the SDLC. This saves money and time early on by preventing the discovery of said defects later, when fixing them would be much more difficult given the integrated nature of a finished system. Dynamic testing, performed here in the form of automated JUnit tests, is useful for testing actual scenarios a system might go through. These scenarios look at how the system responds to varied input, from correct, expected values to wildly long or wrong values. The results of this kind of testing can inform developers on what else is needed to secure the system from denial-of-service attacks, for example, or resolve other strange behaviors given different inputs.

Functional testing checks that software does what it’s supposed to do while non-functional testing checks that the software runs optimally given different contexts. For example, a functional test looks at whether a video game successfully lets the player character jump at any non-cutscene moment whereas a non-functional test for the same game scrutinizes its performance (i.e., speed, crash resistance) when 10, 100, 1000 and so on players are online at a time. Both are crucial in making sure the delivered, finished product fulfills its requirements and performs well enough to continuously do so.

Unit, integration, and system testing all go together to gradually check that a system still works as its pieces are put together. Once units are proven to work without defect, they are integrated. Once they are integrated without defect, the system is tested as a cohesive whole. At this point, acceptance and usability testing becomes possible, since users will have a system to interact with. Having users interact with and give feedback on a working system is a great way to find out if requirements are being met as intended before the product is fully launched. With any changes made to the software (at any stage), regression testing is performed to see whether the changes have introduced new defects.

**Mindset**

***Caution and Complexity***

The Service classes relied on and used definitions in their related base classes to work correctly. Similarly, all Test classes were built based on existing structures in their related base and Service classes. For example, Service and Test classes use get() and set() functions from the base classes to manipulate objects in ArrayLists like “contacts”. Because of this, each set of Contact, Task, and Appointment classes are intertwined.

Keeping these interrelationships in mind is important for maintaining caution. When any pieces of code interact, they have the potential to change each other in undesirable ways, especially in the case of logical bugs. Because the ID variable for each class is required to be immutable, caution was exercised by using the “final” keyword in its initialization and making it a private variable with no set() method. These steps make it so that ID values are unlikely to be accidentally altered.

Caution was also exercised by testing for situations beyond the base requirements, like adding multiple Task objects to the “tasks” ArrayList, for example. Further, the annotation “@BeforeEach” was used in Test classes to initiate new instances of their base classes (such as AppointmentService for AppointmentServiceTest) to avoid issues with lingering values from previous tests.

***Limiting Bias During Review***

As the developer of the code *and* the person reviewing it, it’s easy to encounter bias and hard to detect blind spots. Thankfully, the Eclipse IDE and JUnit Library have built-in ways of delivering feedback that made refactoring easier. Upon the first submission of the Contact classes, JUnit test coverage was not above 80%. Eclipse highlighted parts of the Contact class with a message indicating that not all branches were covered for different parts of the code, like the constructor. This led me to evaluate what I’d missed in testing, and I concluded that, to reach at least 80% coverage, I had to additionally test for instances in which input was *exactly* at the character limit or when set() methods were called with invalid inputs. Adding these tests boosted the coverage percentage, but it required some time and thought to *know* to add them. When one is the developer and tester of their code, it can take longer to realize what’s missing than if they had a dedicated, separate tester.

Bias toward one’s code can also manifest as an unwillingness to accept suggestions for fear that they are criticisms (Hambling et al., 2019). Since I worked on these classes alone, suggestions had to come from myself, even if they meant extra work. This was salient during work on the Appointment Test classes, which required instances of Calendar and Date objects to execute properly. I began working by copying and pasting the same lines of code in each test() function for setting up the “date” variable, but I knew it was bad practice to repeat the same code over again. This feeling compelled me to make a setUp() function in the Test classes, then call it for each subsequent function that needed a date variable. Changing code to use best (or, at least, better) practice is necessary for limiting bias and continuing to grow as a developer.

***Importance of Discipline***

Testing software is a rigorous and thorough process. Its goal is to mitigate damage and losses by finding defects and confirming working code early and often in the SDLC. Despite this importance, writing test cases can feel repetitive, time-consuming, and maybe even mind-numbing, so it requires discipline and organization for success. For example, the variables of Contact required test cases related to their limits regarding length and null-ness. Yet, with only these two limits, multiple test conditions are born, like the variable firstName being too long, null, or at the maximum possible length. For phoneNumber, which requires a length of exactly ten characters, an additional condition exists for checking if the value is too short. Considering these branches is how tests are written to thoroughly cover the code.

Being thorough throughout the SDLC and striving for at least 80% code coverage in initial test set-ups is how to avoid test/technical debt. The week after submitting the Contact classes, I refactored my code because I was in technical debt—ContactTest had less than 80% of Contact covered. To successfully model my future work for Task and Appointment classes, I needed to fix this deficit early by considering which branches I’d overlooked. This approach of striving for higher quality tests early in the project’s lifespan is how I’ll handle future work, because it made Task and Appointment development quicker and smoother than if the technical debt had continued to grow.

References

García, B. (2017). *Mastering software testing with JUnit 5: Comprehensive guide to develop high quality Java applications*. Packt Publishing.

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing : An ISTQB-BCS certified tester foundation guide - 4th edition.* BCS Learning & Development Limited.