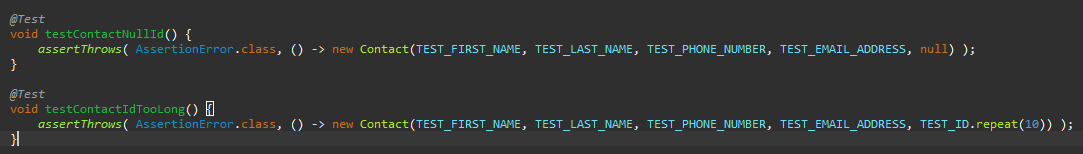
**Summary**

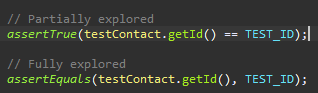
My unit testing approach for each of the three features was to test each software requirement individually. For example, each feature uses a class for in-memory data storage—Task, Contact, or Appointment—and another “Service” for managing large amounts of that data. The former bundles data like a unique identifier, first and last name, and description, and they each have different specifications like being immutable or having a character length. Thus, I used tests like this one for each value:



Doing the same for every specification of every value ensures that my tests align with the software requirements.

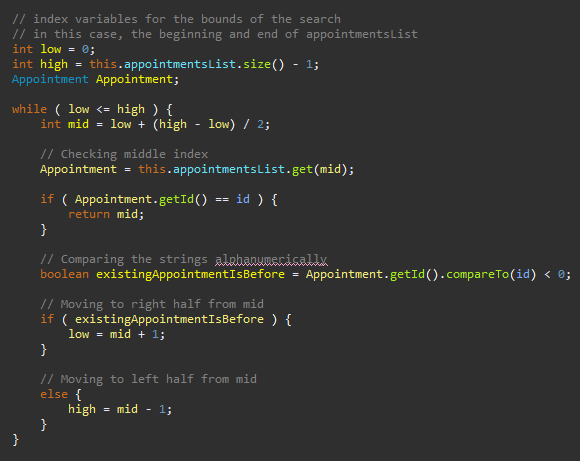
The program’s JUnit tests have an overall coverage of 84.1%, meaning that approximately that percentage of the code is being explored. This means that the vast majority of the lines of code in the program are executed during the tests. I believe this shows that they’re effective.

While I was making the tests, I made sure to use the best practices and reduced variance as much as possible. As shown in the previous code snippet, I used constants like “TEST\_FIRST\_NAME” or “TEST\_EMAIL\_ADDRESS” for consistency. Although it might sound obvious, small optimizations like this can have a huge impact on testing. I also encountered scenarios like this one:



While lines are effectively the same, the first is considered “partially explored” when executing the JUnit test. This reduces the test’s coverage—subsequently affecting the coverage of the overall program. A small detail like this can create inaccurate test results which would then lead to inaccurate conclusions on its effectiveness or the program as a whole. Identifying this issue helped me create more accurate, technically sound tests.

The biggest contributor to my code’s efficiency was my implementation of binary search in all the Service classes. They manage lists of Task, Contact, or Appointment objects, and these lists can be cumbersome on larger scales: taking longer to search, insert, or remove items. Binary search is much more efficient because it divides the list into halves rather than checking each individual item from the beginning:



It checks the middle item, if that Appointment’s id is less than the target, it moves to the lesser half, otherwise it checks the greater half—rinse and repeat until it either finds the matching Appointment or runs out items to check. I also used this to increase the speed of inserting items to the list by having it return the last index it checked.

**Reflection**

In a previous module, I talked about how my tests employ white-box strategies rather than black-box or experience-based ones (Marberry, 2025). Not only were they made by the same person that wrote the rest of the program—me—they’re also based on the inner workings of the classes they’re testing; a notably white-box approach. For example, many tests, like the ones which verified that member variables adhered to their character limits among other specifications, were created knowing that invalid values throw an AssertionError. Thus, the test fails when this error isn’t thrown. This is the exact kind of “specification-based” (Hambling et al., 2019, p. 116) that defines white-box testing methods.

On the other hand, black-box and experience-based testing strategies focus more on “what” the software is supposed to do rather than “how” it does it. A black-box test could be as simple as a checklist that the program can make and update appointments or tasks. I’d describe it as a “requirements-based” test. Compared to white-box testing, it’s higher level—meaning further abstracted from the inner mechanisms—but it’s still focused on the software and business requirements of the program. Experience-based tests, on the other, other hand, aren’t based on any formal specifications or requirements. As I said in a previous module, “Instead, they rely on users’ and testers’ experience. It can range from “guessing” potential errors or weaknesses to a “high-level checklist” of key areas” (Marberry, 2025).

Each testing method has its pros and cons. I’ve said that black-box and experience-based testing work best for larger projects with dedicated testers, and that white-box testing is the least cumbersome for smaller projects or when developers are the ones doing the testing (Marberry, 2025). It would be difficult for a tester to create white-box tests unless they were involved with development. For this program, since the developer and tester are the same person, me, white-box testing is the most natural.

My mindset with testing was to treat the software requirements as a checklist. I’d go down the list of features to make sure each individual specification was being tested and verified. I was cautious in not assuming that the code’s interrelationships worked exactly as expected all the time. For example, when testing adding an item to the Service classes, I didn’t just test adding one at a time. In practice, they could have dozens, hundreds, or more items, so I made sure to test that, after multiple additions, everything worked as expected. Notably, I made sure to test my method of “rounding” the result of a binary search for quicker insertions. That way, I’m testing the technical requirements of the software on top of its functional and non-functional ones.

I tried to limit the bias of my tests by not cherry-picking the values I used to best suit the code. In other words, I tried not to use only the values that I, as the developer, know will work. I also didn’t make assumptions based on my knowledge of how the program works. For example, when testing adding a Contact object to a ContactService, I used lines like these:



I know that testService.hasContactId returning true *should* mean that the matching Contact object is in testService, but just to make sure, I added the second assertion. My assumption was based on my bias and prior knowledge as the developer of both methods. This kind of bias, intentional or not, can influence software tests. If the person making the test is already familiar with the program, it can be difficult to imitate or predict the experience of an end-user that doesn’t know anything about it.

It’s important not to cut corners when developing or testing software because it almost always comes back to bite you. More than just writing “clean”, easy-to-read programs, low-quality code is exhausting to debug, and update. It becomes as much, if not more, work to understand as it was to create—something I would lump under “technical debt.” I think some good strategies for avoiding technical debt are taking mental breaks when needed, collaborating more frequently, and deliberately refactoring your code to be simpler to understand as you scale it. Of course, I also think constantly refactoring can be counter-productive to actually finishing the program, but, in moderation, it’s very important. For example, I’ve mentioned a few times that the Service classes all use binary search. The exact method for which is nearly identical between them. While it reduced the technical debt to reuse code I had already tested and verified, in hindsight, it could’ve been consolidated into one function. Perhaps the Service classes could share a parent class literally called “Service”. Thinking about a program’s shortcomings like this is another important discipline. Even if their supposed fixes aren’t implemented, I think analyzing potential optimizations beforehand can make the technical debt easier to address when the time comes.

References

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.

Marberry, M. I. (n.d.). *CS 320 Module Five Journal Marberry*. ms, Southern New Hampshire University. Retrieved June 22, 2025, from <https://learn.snhu.edu/d2l/le/content/1918310/viewContent/40602817/View>.