I used a similar testing approach for all three features of implemented in the project. Effectively I used edge cases, code coverage, and what amounts to white box testing for all the features. The major advantage to this was that when I tested the edge cases which were length and characters allowed, I was able to reduce my testing from every possible case to a handful(typically 5-6) test cases that ensured the features functioned in the correct way. Most test cases for user input covered minimum allowed length, empty input, blank input, maximum allowable length, 1 character above allowed length and characters that were not allowed in the cases of phone numbers or names. This type of testing allowed me to test the most likely failure points while not having to test exhaustively, while granting me a high level of confidence that the features would function as intended. Every test case I built was based off the requirements listed if a field was changeable and if the output matched the expected output. More specifically, I wrote test cases that focused on particular methods, and tested all edge cases. If a field was required, I ensured the field was properly assigned or if it threw an exception. If a field was mutable, I tested to ensure any changes followed the same initial input requirements. This could catch unintended bugs like a direct assignment to a variable rather than using the validation methods to ensure the input was formatted correctly.

I knew my Junit tests were effective from the number of bugs I detected in my initial code and the results aligning with expectations. For example, through the building of the multiple modules I made multiple mistakes, typically when I copy pasted several lines of code that used the same methods for validation. In numerous cases, I found that while my initial assignment of the variable (such as name) was correct, when I went to change the field, I was actually changing the description field, or a different name field (first name rather than last name). I also achieved an exceptionally high-test coverage in terms of lines covered, as consistently there were only 3 lines “untested” and those lines are only likely to be triggered when the there are several quadrillion entries, and even then, it would still rarely trigger. Those lines of code were also tested with a smaller sample size but could not be tested under the exact conditions, which specifically was a variable that could accept any of 62 characters in 10 separate positions. Everything told, I had a roughly 99% instruction coverage, with all except the three branches described above being tested numerous times throughout the tests.

I enjoyed the process of writing tests, with Junit’s focus on exceptions I was also forced to write my code in a way that handled and threw exceptions, which is a feature I rarely use otherwise(even though I should). Typically, throughout my testing, when possible, I would ensure each instantiation of an object or action taken on said object did not throw an exception unless the action was intended to. In several cases this made me find errors/typos in my branch logic that allowed a piece of code to throw an exception when it should not or throw an exception for perfectly valid input. I also tested that input was correctly handled by using “assertTrue(SomeString.equals(intendedString))” or that the opposite happened, where for whatever reason an assignment should not have taken place, I used the inverse of the above string “!” to ensure the string was not erroneously assigned. This specific approach caught many errors, from invalid assignment to incorrect assignment this testing by far was the most useful as those can be some of the hardest bugs to find in a large project. An incorrect assignment does not typically flag any type of error or exception, so these bugs can be especially hard to detect during manual testing. The major way I checked the efficiency of my code was to ensure my testing was not redundant to the maximum extent possible. While some redundancy is required to ensure proper testing, its important (at least in my mind) to not have overly redundant testing whether that be from incorrect test execution/placement, or from being “extra thorough” in a way that is unnecessary. I think the separation of the testing of the Service class and the main class assisted this mindset the most as even the name of the file specified exactly what was intended to be tested. Excluding a specific test case that is intended for searching a large vector, the rest of the tests could execute in a shorter amount of time than the test took to prepare in memory. That one large test case was also significantly slower than it could be in a real-world project as the search method is a sequential search by index and not any advanced search method like partitioning, binary tree search or, Hash mapped data. Realistically, with an optimized search method this portion would likely take a tenth of the time it does currently.

As far as testing techniques go, I already touched on that above, but I will elaborate on those techniques. First, I used white box testing, where I tested the inputs and outputs of methods and classes, rather than testing the interior components of those method directly or testing the interactions between two methods directly. An example of this was that I never actually tested my string validations directly. I tested the methods that used those methods and checked the input and output of those methods, effectively indirectly testing my string validation. While this method of testing is useful in some applications, it would not be suitable for all applications where the methods depend on each other mutually for their specific tasks, for example, two methods that work off the other’s calculations in a loop until a desired result is achieved. In this example it would be better to test each method directly, as one small bug could amplify an issue or cause confusion when attempting to isolate bug. I also used edge case testing, as exhaustive testing is rarely feasible when a large number of inputs is involved. Even in a short string of numbers and letters, edge case testing is the preferred method as testing every character would take an unreasonably long time. Exhaustive types of testing are better for a small number of variables with direct interactions and specified options, such as three different sets of five check boxes that control user preferences on a website, such as color schemes and layouts of information. In the above-mentioned case, it would be feasible and important to test all configurations as the set of variables is small, and the interactions could result in undesirable effects like text being displayed in the same color as the background, or layouts with high opacity that make options difficult to see for visually impaired users. I also did not use test-driven development in the strictest sense of the technique, meaning while I developed the code for the tests, I chose to write the code and then the tests rather than writing the tests and then aligning my code to meet the requirements. I focused on making my code meet the requirements then attempting to break my code however I could, excluding a couple area where I knew it would be impractical to write code for the issue, like when a user escapes the string input for code injection, something that would even be difficult to do in Junit automated testing due to language conventions and IDE’s treating it as an error. Practically speaking, I think I used the best testing methods I could for the task at hand, while there are plenty of cases where the testing methods, I used would be inappropriate like I mentioned above, they were effective for these projects. Testing methods need to be applicable to the situation, you would not do agile test-driven development in a waterfall style project, and you would not want to test your agile projects months or years after you complete your first sprint. Its good to pick solid testing methods and techniques, it is better to pick methodologies and techniques that make sense for each specific project if possible.

In terms of my mindset, I approached all of these assignments and the project as a learning opportunity. I wanted to become familiar with the testing techniques and the documentation surrounding Junit5, as to be honest Junit5 has significantly improved my view of Java, as compared to C++. While Java still has limits that C++ does not, this automated testing has made Java a reasonable alternative to me in the future for projects I would otherwise code in C++. Throughout the project I employed caution in having confidence that my code would work. Instead of assuming I wrote the code right, I assumed I made mistakes that I should not have and tested extensively to eliminate my errors. As far as I am aware, there was one major mistake I did make in the last assignment with an improper reference that I did not catch until the final assignment, but outside of that I caught far more bugs than I expected and, in all honesty, drastically improved my bug isolation and troubleshooting abilities. It was important to understand the code and the relationships involved between pieces of code because it allowed me to focus my testing in areas that were most beneficial and impactful. As mentioned above, I did not directly test my string validation method, but that is because I understood that testing every method that used that method would yield the same results more efficiently, if an error happened, it was easy to isolate due to me understanding how the code worked together and where a failure would occur and under what circumstances. I limited my bias by treating the code as if a monkey banging on a keyboard wrote it. I understood that if I took any failures as a hit to my pride, then I would likely miss bugs, either by expecting something to work, or by subconsciously avoiding issues I may have in my code. I could definitely see bias coming into play if required to test your own code in a professional setting. If you, for example, were quickly approaching a hard deadline and were expected to test all of your own code it would be easy to ignore a rare bug or avoid a particular “unlikely” scenario in your code to save time or reduce the chance of having to rewrite a large portion of your code. On the flip side, I think if you are working on your own project, or something you are heavily invested in (like your own professional product) you might go overboard in your testing, building massive test cases that were exceptionally redundant because you had invested so much into the code. One big issue in game development for indie developers is over testing or optimizing code or working alone on a project because it is your baby and you will not let someone else mess it up. Many indie developers spend years developing a game, that had a functional minimum viable product within months, a couple examples include Rust (which was entirely rebuilt from alpha multiple times despite how much players liked the game) and Kenshi, which took nearly a decade to develop and was built by a single programmer until 3 years prior to competition, where the developer started to allow outside help to finally complete the game. Both of those examples had other time delays, but over testing features to avoid the smallest potential of a bug (while missing massive ones) definitely contributed to those delays. One thing I take from both of those examples is that no matter how much you test your game, on day one of release someone, somewhere will find a massive game breaking bug that was missed despite how meticulously the game was tested, and you will work overnight to fix it. To me, that means its important to test edge cases, its important to be thorough, but its also important to disconnect yourself a little from your project so that you can reduce your blind spots and learn.

Finally, I think that a commitment to quality is a must. If you are making a game where upon release players spawn and fall through the ground, because you did not think to check that, you obviously were not focused on quality. In the same regard, over concentration on meticulously and exhaustively testing your code just leave you with nothing to release. I think it is important to write the best code you can the first time, write effective and efficient tests of your code that intentionally go after potential weaknesses, avoid building a spaghetti code nightmare. The easiest way you can do this is test driven development, or if you just want to write your code first put aside time in each session for building unit testing and automated tests. If you do this, you can make minor and major changes to your code without the worry of breaking everything. Even if you are in no way responsible for writing tests for your code, do it anyway. While it may not help you initially, it will enable you to maintain and improve upon your code base as it gets larger. From my understanding, many people end up being given some legacy code they are responsible for maintaining, additionally, they are typically tasked with maintaining the code they write, and eventually that can become a huge library of code, in some cases written years prior to you needing to touch it again. Writing quality code helps but spending a couple hours developing unit tests for the code could save you dozens of hours in the future, especially if you need to change the code, without impacting any legacy systems that still use it. Those old unit tests will come in extremely handy in ensuring the legacy systems still work, while additional unit tests can be added to ensure the code meets the new requirement as well. I think a method similar to this would help many programmers avoid technical debt, especially when it comes to personal projects, that they may only be able to work on a couple times a month. I know I have come back to a project (without version control or unit testing) and messed everything up, and in some cases eventually scrapped the whole project because I did not effectively document it, I wrote an intricate web of spaghetti code, or manually testing it after changes took up the majority of my time after even minor changes. Its sad, but I want to say I have significantly more abandoned or dysfunctional/dead projects on my hard drive than I have complete, and by a wide margin (though that ratio is changing regularly as I improve my skills and acknowledge my previous issues).