In my application project, I created software and unit tests to ensure that the program met the requirements. My approach to creating the unit tests for this project was to ensure that all the requirements listed were met. To do this, I started by working through the requirements one-by-one to ensure that each one had been thoroughly tested in the code. I created the tests for each class before moving on to the next one to ensure that it was fully tested, and no requirements were missed. I also checked the coverage percentages for the tests to ensure that I didn’t miss any part of the code such as a constructor or getter method. All my code had 100% coverage. This does not mean that the code does not have errors, but it provides much greater assurance that the main requirements are working as intended.

When writing JUnit tests, it was important to make sure that the code was technically sound. What this means is that the code itself does not make it easy for bugs or errors to be introduced later. To ensure that the code was technically sound, I made use of language features in Java such as marking variables as private and final as well as creating getter/setter methods instead of exposing public properties. Additionally, I took steps to make sure the code was efficient. A major part of this was using the correct data structures and algorithms for the needed requirements. As an example, I used hash tables instead of lists for the collections because random access lookup was needed more than performing operations on the entire collection.

In this project, I used several software testing techniques. The first was unit testing as described above. Unit testing is testing individual units of code to ensure that they meet the required functionality. In addition to unit testing, I also did integration testing between classes to ensure that they worked together correctly. Examples of this include using Appointment, Contact, and Task classes inside their Service classes. Finally, I also did functional and non-functional testing to ensure that the code met the requirements. This entailed running the code manually and ensuring all the requirements were met.

Some common testing methods that I did not use were system testing, acceptance testing, and performance testing. In system testing, the code is checked to ensure that it runs correctly on the entire system not just in tests. This was not practical to do in this project because the code was not determined to be run anywhere yet. The next type of testing that would be done is acceptance testing or testing that is done with end-users to ensure the application meets the requirements. This is very necessary as sometimes the given requirements may not fully describe the behavior that is desired. Finally, performance testing could be used if this project were to be used in such a way that made running it difficult. An example of this would be if there were thousands of simultaneous users trying to access the application at the same time or if there were hundreds of thousands of appointments or contacts in a service.

While working on this code, having a good testing mindset was essential. This testing mindset encompasses being very detail-oriented and cautious. While writing the code, I was always thinking about ways that I may have overlooked something or a way that a future developer might mistakenly use the code in the wrong way. This is why as a software engineer you are not just writing code as quickly as possible to fix the problem. It is essential that you write well documented code without duplication and unnecessary abstraction so that it is simpler for the next person to understand the requirements and make changes. As an example of this, there were several different classes for storing data such as Contact, Task, and Appointment. Additionally, each of these had a corresponding Service class such as ContactService, TaskService, and AppointmentService. It could have been tempting to create more complexity here to abstract this relationship into something such as an Item and ItemService which each class and service needed to inherit from. However, all of this complexity is not needed in the current application and would have made it much more difficult in the future if one class and service needed to be changed but not the others.

In addition to the right mindset while developing, I tried to minimize my own bias as much as possible. Bias is undoubtedly a concern while testing your own concern as a developer. Often the way you use and interact with software is different than how a designer, tester, or end-user would. It is best practice to always have another person review the code, but if that is not possible or cannot be done very frequently, then it is essential to find good ways to limit bias as much as possible. One way to do this is to determine how other people use the software. Once you determine common use cases and different ways to use the software you can incorporate those into your testing approach to try different things that other users would do. Another way to limit bias when testing your own code is to make use of automated tools that ensure best practice. Examples of this could be linters and your IDE that check and make sure you aren’t making any silly mistakes.

Finally, it is important to be disciplined and committed to quality when writing software. As a developer it can be tempting to try to release software as fast as possible to look good to other developers or management. Additionally, tight deadlines and pressure as well as fast deployments can lead to the idea of moving quickly and breaking things as it can be fixed quickly later. While releasing software quickly is good practice for a lot of modern applications, it can be a problem when taken too far. A development culture around not testing or thinking about the technical debt of the code can end up in releasing a lot of software now but ending up with unmaintainable code and slow, difficult work in the future. Thus, it is essential to have a good balance and mindset of creating quality, well-documented, and well-test software in a timely manner. It is worth pointing out that the opposite can be true as well. Fear of making change or obsessing over perfection can create a culture where it is difficult to make needed changes and release features to your users. Like all things in software development, the right answer usually depends on the project itself as well as the company. However, in general being well disciplined and committed to quality when writing code will ensure that you are creating value for the long-term and not just the short-term requirements and needs.