For each of the three features, my testing approach aligned with the software requirements perfectly. Before I began constructing the code for each of the three features, I first had to analyze the requirements for each feature. After analyzing these requirements per feature, I would construct a test to check the integrity of that given requirement to make sure it operates correctly. For example, most tests require validation checks to ensure users do not go over the limit when inputting certain criteria. For example, in AppointmentTest.java, there was a requirement to ensure that the users would input a date that is not in the past or null. To correspond with this requirement, testDateInPast() would test the date the user had input, checking if it passes through the validation checks of being in the future and not being null. To ensure the quality of my tests, I had to double check certain tests to make sure that they were covering the requirements they were supposed to. For example, in ContactTest, there is a function to test if an error occurs if the contact’s first name is too long. Whenever a user inputs the contact’s first name too long, an exception would be thrown if the first name is over the 10-character limit. This test would also be verified by the coverage percentage reporting that the function had passed and highlights line 33, providing a visual for verifying the test had passed. To double check this function, I would change the variable to have the first name be within the 10-character limit and then try to run the exception, which would result in a failure because the exception was not thrown. To ensure that my code was technically sound, I would have to test every corresponding requirement in the class and run it through a JUnit test. For example, in TaskTest.java, I had included validation tests—testTaskIdToolong() on line 25 and testTaskIsNull() on line 55—to make sure that the TaskId was within the 10-character limit and was not left empty. To make sure that my code was also efficient, I incorporated a HashMap in AppointmentService, ContactService, and TaskService to store their created appointments, contacts, and tasks. For example, in AppointmentServiceTest.java on line 15, the function testAddAppointment() would create an appointment with the appointment’s id number, a future time, and a simple description. Throughout this project, I used a few software testing techniques when running my JUnit tests. The most common software testing techniques I used were boundary value testing and negative testing. Boundary testing was the most technique I used, testing the values of different variables. For example, in AppointmentService.java, the requirements said that the appointment id must be less than 10 characters, the appointment date must be less than 10 characters, and the appointment description must be less than 50 characters. I would test these variables to see if the appointment can be created once each variable passes the boundary test while also testing if exceptions are thrown whenever the variables exceed their maximum character limit. For negative testing, I would test variables to ensure that an exception would be thrown if there was a null input. Using the same example, while also testing if the variables are within their respective character limit, I had also implemented tests to ensure that there were no null inputs and if there were, an exception would be thrown. Other software testing techniques that I had not used in this project were security testing and performance testing. Security testing techniques test the vulnerabilities of the software, ensuring the safety of its integrity in case of a breach. An example of this testing technique being used in the project is testing the security of appointment IDs, ensuring they cannot be accessed from unauthorized users, and their personal information is protected. Performance testing tecniques involve testing how the software handles different workloads at the same time, testing its ability to work under stress. An example of this test being used in this project would be trying to create multiple contacts, tasks, and appointments in a single test run, testing the software’s efficiency when creating hundreds of different services. Each form of testing techniques has its own respective practical uses and is used in different situations. Boundary testing and negative testing are useful for software projects that involve the use of constraints that must be enforced and ensuring their variables are not exceeding their maximum value and are not empty. Security testing is crucial in multiple software applications, mainly financial or health care systems, ensuring the safety of a user’s personal information. Performance testing is the most common form of testing in software applications. Performance testing tests the integrity of software applications and allows developers to observe how their system handles stress. There would be situations where e-commerce platforms like Amazon or Walmart must use performance testing to ensure their system runs efficiently while handling multiple users’ orders, especially during special events like Black Friday. While acting as a software tester, I had to employ caution when analyzing how each component interacts with each other. If I had not exercised caution, I could potentially overlook a defect that could lead to multiple errors. For example, in AppointmentServiceTest.java, if the boundary test was not verified correctly, it could influence the add appointment function. It was important to appreciate the interrelationships of the code I was testing because it showed how small changes in the code could lead to significant consequences. This led me to employ caution when boundary testing and negative testing, ensuring that the validation rules functioned as they should. While reviewing my code, I imagine that there would be bias in thinking that my code is efficient and sound since I was the one who developed it. But to limit this bias, I would have to test my code in different ways to ensure it was technically sound. For example, when boundary testing or negative testing specific variables like ContactTest’s first name, last name, and phone number, I would ensure that the exception would be thrown if any of these variables exceed their maximum character limit. I would also do the opposite and test the variables within their boundaries, observing that the exception would not be thrown because the code was following the validation check. Maintaining discipline and committing to quality is essential for all software engineering developers. It is important not to cut corners when it comes to writing and testing code because one error could cause multiple consequences, and it is crucial for developers to ensure the integrity of their code before releasing it to the public. As an example, for what happens when software developers cut corners, when Toyota was investigated for their unintentional acceleration case, they noticed that there were 20 to 30 lines of errors in the code that could accidently increase the acceleration of their models. This defect caused reputational damage to Toyota’s consumers, and the company was fined millions of dollars for this occurence. I plan to avoid technical dept as a practitioner in this field by analyzing and adhering to the requirements of my clients and performing multiple tests to ensure the stability of my code before moving onto the next requirement. These methods of avoiding technical debt are shown throughout each function, with each requirement having a corresponding test to ensure the integrity of the validation and being labeled how it should function.

**References**  
  
Mayne, E. (2010). Toyota Steadfast in Safety Probe. *Ward’s Auto World*, *46*(8), 16–17.