My testing approach was aligned very closely with the requirements among all three features. For example, when it came to the Appointment testing I specifically tested for the date functionality as seen here:

void testAppointmentPastDate() {

Date pastDate = getFutureDate(-1);

assertThrows(IllegalArgumentException.class, () -> new Appointment("1", pastDate, "Birthday Party")); }

This test directly addresses the requirement and common sense that an appointment date cannot be set in the past. I tried my best to get the best coverage of the code by testing all public methods, when testing for int values doing both positive and negative, testing boundaries, and testing each requirement explicitly. Each test was also done with only one test at a time. This way if there was a bug it could easily be noticed based on the naming.

I exhibited technical soundness in this test here:

private Date getFutureDate(int days) {

Calendar calendar = Calendar.getInstance();

calendar.add(Calendar.DATE, days);

return calendar.getTime(); }

This was a helper method used for more cleaner and maintainable tests.

I showed efficient code with this test:

void testAddAppointmentSuccessfully() { AppointmentService appointmentService = new AppointmentService();

Date futureDate = getFutureDate(1);

appointmentService.addAppointment("1", futureDate, "Birthday Party"); assertThrows(IllegalArgumentException.class, () -> appointmentService.addAppointment("1", futureDate, "Birthday Party")); }

This test just like my others demonstrates how minimal I tried to keep each one.

For my techniques I used Unit Testing, Exception testing, Date validation, and Boundary. Junit testing is used to test components by themselves. Each test only tests one thing. Each test also uses @Test to annotate test methods. Exception testing is used to verify error conditions. It also tests the systems error handling. It uses assertThrows for checking the exception. Date validation checks the actual date. It checks for whatever rules were set in place, in this case no past dates should be able to be used. Boundary testing is used to test the edges of acceptable values. What happens if the last accepted value is used, and what happens if the value exceeds the limit.

The mindset I adopted was that of someone who might use the system. Afterall the users will be the ones encountering any bugs in the system. I tried to think of anything a user might accidentally do while using the system. I also thought of times when I found bugs in software and how I did that, whether accidental or on purpose. With this mindset I also tried to be as unbiased as possible. A problem I have is on work I have performed it is sometimes hard for me to see errors. This is because I performed the work I did with no intention of errors. I needed to be more optimistic for these tests and keep an open mind on every possibility.

When starting to learn something new I always go into the experience with one thing in mind. Rules are rules for a reason. At some point a rule was created to avoid something from happening. This applies to cutting corners for me. Cutting corners is usually not a good thing to do as it can lead to disastrous results. I am able to stay disciplined and follow best practices because for one it leads to understandable code to someone new to a project, I won’t have to go back and explain what each thing does. I also do not want a failure to come up later on in a project and have it be an issue because corners were cut and then have to go and fix those corners. It creates a lot more of a workload and unnecessary stress. To avoid technical debt I believe it would be good practice to write tests alongside the code that requires it. Constantly refactor and update tests. If an update is made to a method, go and update the tests too. Cover as many scenarios as I can and always follow best practices. They are best practices for a reason, and creates a good foundation for good coding.on, and creates a good foundation for good coding.