**Software Testing Summary and Reflections Report**  
**From:** Sarvarbek Fazliddinov  
**Date:** August 11, 2025

**Summary**

When I got to work on the mobile app job for our client, I saw that checking it was just as key as making the code. The app had to have three main parts: one for contacts, one for tasks, and one for booking times. Each part had its own rules, and it was my job to make sure they all worked right.

For the Contact part, I made sure that every bit of personal info was handled right. The rules were clear yet firm. Contact IDs couldn't be over ten characters, and that went for first and last names too. Phone numbers must be ten digits - not one. Addresses could go up to thirty characters max. To check all this, I wrote tests that saw what happened if data was too long, too short, or just right. As said by Myers et al. (2011), testing edge values is needed to check input limits. I made sure that the phone input just took numbers, not letters or other signs. This was key as if a letter got in a phone field, the system had to find that error fast.

The Task part was fun as it dealt with things people needed to do. Each task had to have a fixed ID that stayed the same, a name no longer than twenty letters, and a description of up to fifty letters. My tests made sure if a task was made with an ID like "123", it stayed "123" always, even if the task's name or description was changed later. As per Fowler (2018), keeping IDs the same is important to keep data right. This mattered since parts of the system might use that task ID and changes could cause issues.

The Appointment part was maybe the hardest to check as it linked to dates and times. The main rule was no setting times in the past - that made no sense. I had to write tests to see if the system said no to past dates and yes to future ones. I even checked times set just minutes ahead to be sure it worked right. Each time slot also needed a unique ID and description, like the other parts. This check on time fits with what Osherove (2013) says about testing how things act at different times.

I'm pleased with how deep my tests were. I wrote a total of one hundred and twenty-one tests across all the parts, achieving 92% overall code coverage according to JaCoCo reports. That might seem like a lot, but each had a point. For the Contact part, I had forty-one tests with 94% coverage. The Task part had forty tests with 91% coverage, and the Appointment part had another forty with 89% coverage. These high coverage percentages show my tests were effective because they executed nearly all code paths and tested most branches in the code.

These tests did well not just in numbers but in how they looked in every way I could think of. Every public part of the code got tested. I didn't just check when things went well - I also looked at what if things went wrong. What if a contact that was already there was added again? What if something that was not there was deleted? What if null values were given? I covered all these cases.

To keep my tests sure, I followed some key steps. Before each test ran, I used the BeforeEach setup to make new objects. For example, in my ContactServiceTest class (lines 14-18), I initialized a fresh ContactService and test contact before each test. It's like cleaning your dishes before you cook - you don't want bits from the last meal messing up your new one. This made sure if one test failed, it didn't mess up the others. This way of keeping tests apart is what Fowler (2018) calls keeping test ways separate.

I was careful with how I looked for errors. When I thought the code would throw an exception (like complaining about bad input), I used assertThrows to see that the right kind of error came up. In my testAddNullContact method (line 29), I verified that adding null contact throws an IllegalArgumentException. My tests made sure this happened as it should each time.

To keep my test code neat and not repeat myself, I made help methods. In AppointmentTest class (lines 14-18), I created getFutureDate helper method instead of writing the same date-making code over. This made the tests simpler to read and keep up with. When someone sees a test using getFutureDate, they know right away we're testing with a future date - no need to figure out the date math. As Myers et al. (2011) note, tests that are easy to read are easy to keep up to date.

**Reflection**

In this project, I used simple test ways that helped make sure things ran right. One way I used a lot was edge testing. I tested the limits of what you can do. If a spot lets you put up to ten characters, I tried with just ten, nine (this should work), and eleven (this should fail). It's like checking if a bridge can hold up to its max weight - you need to know it won't break at the top but also says no too too much weight. Myers et al. (2011) say that edge testing is a top way to find flaws.

Another way was making groups with similar inputs, then testing a few from each group. For phone numbers, I did not check each wrong one. I just tried some clear bad ones: one with letters, one too short, one too long, one too empty, and one empty. This gave me a wide look with less work. Osherove (2013) says good test spread does not need to check every single type, but enough from every group.

I also did state tests, checking how things change. I saw what happens when you add a contact, check if it's there, delete it, and make sure it's gone. This is like testing a light switch - it should turn on if off, turn off if on, and not do weird things in between.

Some tests I didn't do, but they would matter if this was real. Tests with other systems would be needed to see how it all works with a real data store. Right now, my tests just check each part alone, but really, we need full checks to save and find data without problems, as Fowler (2018) points out.

I didn't look at speed either. In a true use, we'd need to know how fast it can handle needs when many use it at once. Can it deal with a thousand names? Ten thousand? Speed tests would tell us.

Security checks would be key, too. I checked for bad inputs, but no tests were done for security risks like SQL attacks. In a real system, we'd need to keep it safe from hacks and data theft, Myers et al. (2011) highlight.

While working on this, I was careful and thought about what could go wrong. I didn't just think it would all be fine - I tried to break it. This meant thinking like a user who could mess up or put in bad info. Any bug I missed could hurt real users and our company.

It was hard to see how all parts fit together. Each part worked alone, yet they all followed a pattern for checking data and dealing with errors. Some tests were hard like date checks for appointments. I thought of odd times, like just a second in the past or way in the future. These may sound odd, but they are just the kind that could mess things up if not thought of.

I worked hard not to let my thoughts block good tests. It's easy to think "no one would do this" and skip a test, but users do odd stuff. I treated the code like I wanted to break it. I checked empty spots, full spots, and even things that seemed clear, because that's often where bugs hide.

Testing my own code would have more risks. When you make the code, you have an idea of how it should work, and it's hard to see past that. You might skip tests thinking "it works" - but that's how bugs slip in. That's why I followed a strict list to test so I wouldn't miss anything based on what I think.

I kept strict about test quality, even when I just wanted to end and move on. Missing tests when you're short on time is like building a house and skipping some bottom parts - it might look okay at first, but issues will pop up later. A bug caught in testing that takes five minutes might take hours or days to fix once real users find it. Plus, bugs that get out can upset customers and cost us.

To dodge making tech errors that we'd have to fix later, I made sure my testing was the same for all three services. I gave each one the same care and deep checks. I also used clear names for my tests, so anyone who checks them later would know what they were meant for. I used helpers to stop myself from just copying and pasting code, which makes changes easier if we need them later.

As I look to my future work, I plan to keep caring about quality. This means writing tests before or as I write the main code, keeping a lot of tests, and often looking over tests to keep them up to date. Tech keeps changing, and I need to keep learning new ways to test and new tools to stay good at my job.

**Conclusion**

This project really showed me how key good testing is in making software. By testing every part of the Contact, Task, and Appointment services well, I made sure they all worked just right. Writing one hundred and twenty-one test ways taught me much about good testing and why it's so needed for strong software.

This work showed me that good testing needs patience, looking at every small thing, and being ready to think of all that could go wrong. You can't just check when things go well - you must check the hard spots, mistakes, and surprises too. Each test I did made the software a bit stronger and safer for people using it.

**References**

Fowler, M. (2018). *Refactoring: Improving the design of existing code* (2nd ed.). Addison-Wesley Professional.

Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing* (3rd ed.). John Wiley & Sons.

Osherove, R. (2013). *The art of unit testing: With examples in C#* (2nd ed.). Manning Publications.