**CS 320 Project Two**

Chris Trimble

CS-320

Professor Toledo

August 11, 2025

Chris Trimble

Professor Toledo

CS-320

11 August 2025

**CS 320 Project Two**

**For Project One, I developed unit tests for the Contact, Task and appointment services in the mobile application. My unit testing approach for each feature followed a structured process: identifying all functional requirements, designing positive and negative test cases and ensuring that boundary conditions were covered. For example, in the Contact service, I implemented tests to confirm that contact creation worked with valid data and failed appropriately with null or over length values. Similarly, in the Task service, I tested description updates to ensure compliance with character length limits, while in the Appointment service, I verified correct deletion behavior by appointment ID.**

**This approach was closely aligned with the software requirements. Each Junit test mapped directly to a functional or constraint-based requirement. For instance, the requirements that task descriptions must not exceed 50 characters was directly verified in the ‘testUpdateTaskdescription()’ method, which attempted updates with both valid and invalid lengths. The alignment between requirements and tests ensured that no critical functional area was left untested.**

**The JUnit tests achieved approximately 88% code coverage, which indicates a high degree of effectiveness. This coverage percentage reflects that the majority of code paths, including both expected and exceptional flows, were executed during testing. While 100% coverage is rarely practical due to certain unreachable defensive code branches, the achieved coverage ensured strong confidence in the correctness of the implementation.**

**My experience writing the Junit teste reinforced the importance of clarity, isolation and repeatability in unit testing. To ensure technical soundness, I validated preconditions and postconditions explicitly. For example, in ‘testAddContactValidData()’, assertions were made to verify that the contact list size increased by one and that the stored contact’s fields matched the expected values. To maintain efficiency, I minimized redundant setup by using the ‘@BeforeEach’ annotation to initialize test data once per test case group, avoiding unnecessary object creation and improving execution speed.**

**In this project, I employed black box testing and boundary value analysis. Black box testing focused on validating outputs against inputs without relying on internal implementation details, allowing tests to remain robust against future code changes. Boundary value analysis was used extensively in testing input limits, such as maximum field lengths for names and task descriptions.**

**Other techniques I did not use in this project include white box testing and exploratory testing. White-box testing, which involves evaluating internal structures and paths, could have helped verify specific branches and exception handling more thoroughly. Exploratory testing, a less structured approach where the tester actively investigates the software without predefined test cases, could have uncovered unexpected edge cases.**

**In practice, black box testing is useful when requirements are clearly documented, as in this project. Boundary value analysis is especially important in data entry and validation heavy applications. White box testing is best suited for complex algorithms where internal logic must be validated and exploratory testing is effective for early-stage prototypes or when formal documentation is limited.**

**Throughout this project, I maintained a cautious mindset. Recognizing that small errors could cascade into larger defects, I paid close attention to how each service interacted with the others. For example, ensuring that deleting a contact did not leave orphaned tasks or appointments required careful consideration of data relationships.**

**To limit bias, I approached testing as though as I were an independent reviewer, even though I had written the code myself. This meant writing tests to intentionally break my implementation, rather than confirming that it worked under ideal conditions. Bias is a concern when developers test their own code because they may unconsciously avoid scenarios, they believe are unlikely or already correct.**

**Discipline and commitment to quality were essential. Cutting corners in testing can lead to technical debt, which ultimately costs more to resolve. To avoid this, I ensured that all known edge cases were covered before considering a feature complete. As an example, I tested not only valid and invalid input lengths but also null and empty string values to ensure robustness.**

**References**

IEEE. (2017). IEEE Standard for Software and System Test Documentation (IEEE Std 829-2008). IEEE.  
Kaner, C., Bach, J., & Pettichord, B. (2001). Lessons Learned in Software Testing: A Context-Driven Approach. Wiley.  
Myers, G. J., Sandler, C., & Badgett, T. (2011). The Art of Software Testing (3rd ed.). Wiley.