Joseph Cassello Jr

CS 499 Enhancement One: Software Design/Engineering

Professor Phillips

January 25, 2026

Artifact Description:

For the software Design and Engineering category of my ePortfolio, I selected the Weight Tracking App that I originally developed during my CS 360 Mobile Architecture course. This is a standard Android application that helps users log their daily weight, set goals, and handle SMS notifications for progress updates. When I first built this about a year ago, my main goal was just making sure the “Save” button worked. Because of that, the original code was a bit of a mess–it was a monolithic structure where everything from the database logic to the UI button–mapping was crammed into a single file called DashboardActivity.kt. It worked, but it wasn’t built to last or be expanded.

Justification for Inclusion:

I selected this artifact because it represents an architectural overhaul that I also see in my professional IT role. In my current job, I deal with legacy systems all the time that are brittle, meaning if you change one small thing, the whole system might crash because the parts are too tightly coupled. Even though an end user wouldn’t notice a visual difference in this app, the “engine” under the hood has been altered. I migrated the project to the Model-View-ViewModel (MVVM) design pattern, which is the industry standard for professional Android apps.

Instead of the Activity doing everything, I created a WeightRepository to handle data and a WeightViewModel to handle the business logic. I also implemented ViewBinding to replace the

old findViewById calls, which makes the app faster and prevents the common crashes that happen when a UI element isn't found correctly. This shift demonstrates that I can take a functional student project and turn it into a sustainable professional product. It shows I understand that software isn't just about what the user sees, but about how code is organized to support future growth and stability.

Alignment with Course Outcomes:

This enhancement meets Course Outcome 3, which focuses on designing computing solutions while managing design trade-offs. The specific trade-off I managed here was initial complexity versus long term maintenance. It's always faster to just throw all your code into one file, but that creates a nightmare for testing and scaling later. By choosing to implement MVVM, I traded that initial "quick and dirty" speed for a structured system where I can now swap out a database or update the UI without the whole thing collapsing. I also feel I addressed Course Outcome 5 by cleaning up the way the app handles hardware interactions. By moving the SMS notification triggers into ViewModel, I ensured that the app follows a strict, logical flow–checking permissions and state before it ever tries to touch the phone's cellular hardware, which is much more secure and predictable than my original setup. These align exactly with the outcome coverage goals I set during my Module One plan, and I have no updates to those plans at this time as the refactor followed the original roadmap.

Reflection on the Process:

Refactoring this app was honestly more of a headache than just building it from scratch would have been. The biggest issue was the sheer amount of technical debt I had to pay off. My original code was so tightly coupled that trying to pull the data logic out of the Activity was like trying to untangle a drawer full of old cables. The real challenge was getting the LiveData

observers to actually behave. Once I got the UI to react to the ViewModel automatically, it solved one of the most annoying bugs from the original version: the data wiping out every time the screen rotated.

Dealing with the Android Lifecycle is always a pain, but moving the data into the ViewModel makes the app feel like a real product instead of a prototype. This process taught me that professional software engineering isn't about the flashy parts of the app; it’s about the boring stuff—the structure and organization—that keeps the whole thing from falling apart the next time a developer has to fix a bug or add a feature.