Elijah Harris

# Software Design and Engineering

## Original Code Overview

The original implementation of the Mass Tracker App was functional but suffered from architectural inefficiencies and lack of scalability. The project’s codebase violated key principles of software design by intermingling database access, business logic, and UI logic within the same classes, primarily in DashboardActivity. This lack of separation of concerns resulted in tightly coupled code, making the system difficult to maintain, test, and extend.

To address these issues, I refactored the project to adopt the Model-View-ViewModel (MVVM) architectural pattern. This approach ensures a clean separation between the UI, business logic, and data access, promoting modularity and testability while improving readability and maintainability.

## Enhancement Objectives

The main goals of the refactor were:

1. **Adopt a scalable architecture**: Implement the MVVM design pattern to achieve a clean separation of concerns and align with modern software engineering practices.
2. **Enhance modularity**: Create reusable components for business logic and data access to facilitate future development and collaboration.
3. **Improve documentation**: Add detailed comments to explain the functionality of individual methods and tricky code segments, making the codebase more accessible to a diverse audience, including team members with varying expertise.
4. **Support decision-making**: Ensure the design is coherent, technically sound, and facilitates organizational decision-making through better collaboration and communication.

# Key Enhancements

### Refactoring for MVVM Design

* + **Original Issue**: The DashboardActivity contained all the logic, from UI updates to database queries. This tightly coupled design made it difficult to extend or test the application.
  + **Enhancement**: I introduced three distinct layers:
    - **Model**: Responsible for data management, represented by the MassRepository and DatabaseHelper classes.
    - **View**: Handles the user interface, represented by DashboardActivity.
    - **ViewModel**: Acts as a mediator between the View and the Model, encapsulating the business logic, represented by DashboardViewModel.
  + **Impact**: This refactor improves testability by isolating logic from the UI, making unit testing possible without relying on the Android framework. It also simplifies future development by creating a clear separation of responsibilities.

### Introduction of the DashboardViewModel

* + **Original Issue**: The original code directly manipulated the UI from database operations, which violated separation of concerns.
  + **Enhancement**: The DashboardViewModel mediates between the View and the Model. It fetches data from the MassRepository and exposes it to the UI via LiveData.
  + **Impact**: The use of LiveData allows the UI to automatically observe and react to data changes, eliminating manual UI updates and reducing the likelihood of bugs.

### Enhanced Documentation

* + **Original Issue**: The original codebase lacked detailed comments, making it difficult for new developers to understand the functionality of specific methods or logic.
  + **Enhancement**: I added comments explaining the purpose of each class, the functionality of individual methods, and any non-obvious code logic. For example, I detailed how LiveData is used in DashboardViewModel to update the UI reactively.
  + **Impact**: Improved documentation promotes collaboration by making the code more accessible to team members with varying levels of expertise.

## Alignment with Course Outcomes

1. **Employing Strategies for Collaboration**
   * The introduction of the MVVM architecture enables easier collaboration by clearly separating responsibilities. For example, UI developers can work on the View layer without affecting business logic, while backend developers can focus on the Repository layer.
   * Detailed comments and documentation support diverse audiences, from developers to stakeholders, by making the codebase easier to understand.
2. **Designing Professional-Quality Communications**
   * By structuring the code into clear layers and providing comprehensive documentation, the project communicates its design and functionality effectively to team members and stakeholders.
3. **Designing and Evaluating Computing Solutions**
   * The decision to adopt MVVM demonstrates an understanding of algorithmic principles and software engineering standards. This design optimizes trade-offs between modularity, scalability, and complexity.
4. **Demonstrating Techniques and Tools**
   * The use of LiveData in the ViewModel demonstrates the application of innovative techniques for reactive programming, while the refactoring into MVVM highlights the use of industry-standard tools for clean architecture.
5. **Developing a Security Mindset**
   * While this enhancement focused primarily on architectural improvements, the modular design paves the way for future enhancements in security. For example, centralizing database queries in the Repository enables consistent validation of inputs and outputs, reducing potential vulnerabilities.

## Reflection on the Process

Through this enhancement, I deepened my understanding of clean architecture principles and the MVVM pattern. The refactor required significant effort to untangle tightly coupled logic and reassign responsibilities to appropriate layers. The biggest challenge was ensuring that data flow between the ViewModel and the View remained seamless, which was addressed by leveraging LiveData effectively.

This process reinforced the importance of modularity and documentation in creating maintainable software. It also highlighted the value of collaboration, as this refactor sets the stage for smoother teamwork and future enhancements.

Ultimately, this enhancement improved the project's scalability, readability, and testability, aligning it with professional software engineering standards.