PROJECT / RELEASE 1

Project Design Document

Team 1a - JSCOPE
Oliver Gomes (odg1896@rit.edu)
Patrick Lebeau (pml4324@rit.edu)
Jonathan Ho (jlh5360@rit.edu)
Connor Bashaw (cdb9772@rit.edu)

1 Project Summary

FundGoodDeeds is a community driven fundraising platform designed to connect individuals or organizations in need with donors who wish to contribute directly to meaningful causes.Users can browse and contribute to individual needs or combined bundles of related needs while admins can oversee donations, reporting and systems integrity.

The system follows clear separation between the user interface, controller and model logic, as well as data persistence. It enables extensibility for the future. The first release focuses on implementing core interactions among needs , bundles, and ledger subsystems, for donation tracking and transparency.

2 Design Overview

The Design evolved from simple class sketches and noun verb analysis of project requirements to a structured multi subsystem architecture emphasizing separation of concerns, high cohesion and low coupling.

Model–View–Controller (MVC) separates presentation, business logic, and data.

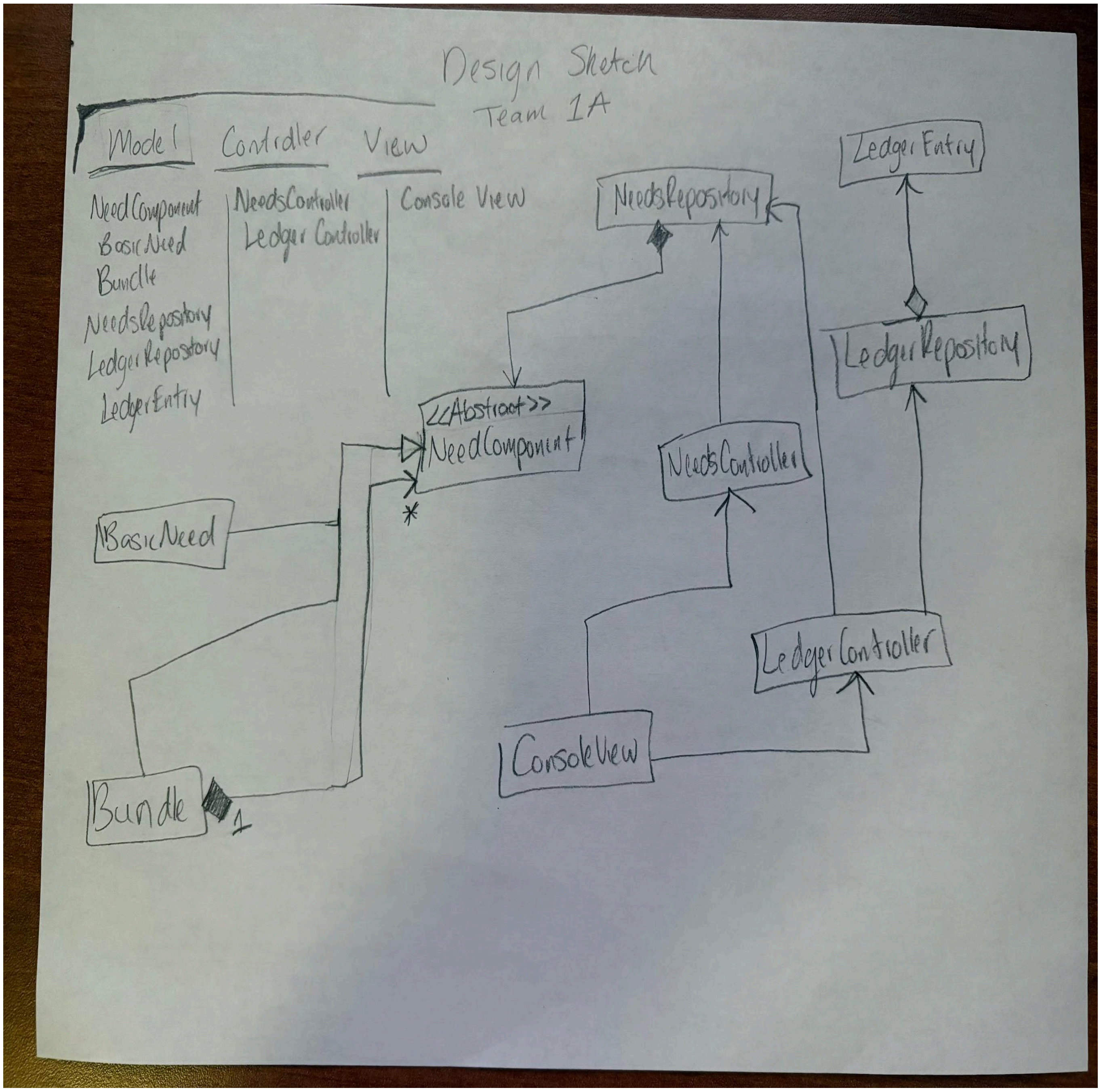
Composite Pattern models *Bundles* of *Needs* uniformly, enabling recursive operations such as total cost or fulfillment status.

Observer Pattern allows the UI and reporting components to react automatically to updates in the data model.

Early prototypes revealed tight coupling between UI and logic, these were then handled through interface abstractions. Dependency inversion was applied to allow testing and future integrations with external API’s or databases.

Rejected alternatives include direct database coupling in controllers and hard coded ui bindings. Assumptions are 1 active donor per session and stable in memory data storage.

3 Subsystem Structure



- **Need Management Subsystem** – manages creation, retrieval, and status of Needs.
- **Bundle Management Subsystem** – groups related Needs using Composite pattern.
- **Ledger Subsystem** – records donations, calculates totals, and ensures transaction integrity.
- **User Subsystem** – handles Donor, Admin, and general user operations.
- **UI / Controller Subsystem** – manages user interactions, events, and display updates.

4 Subsystems

4.1 Need Management

Class Need	
Responsibilities	Represents a single item of aid (description, cost, status).
Collaborators (uses)	Ledger, Bundle, NeedRepository

Class INeedComponent	
Responsibilities	Common interface for Need and Bundle to support Composite operations.

Class Need Repository	
Responsibilities	Handles collection and retrieval of Needs in memory.
Collaborators (inheritance)	Need, BundleController

4.2 Bundle Management

Class Bundle	
Responsibilities	Composite that aggregates multiple INeedComponents. Calculates totals recursively.
Collaborators (uses)	Need, Ledger

Class BundleController	
Responsibilities	Handles creation, update, and retrieval of Bundles.
Collaborators	Bundle, NeedRepository, Ledger

4.3 Ledger subsystem

Class Ledger	
Responsibilities	Records all Donation entries and computes totals.
Collaborators (uses)	Donation, Need, Bundle

Class Donation	
Responsibilities	Represents a single contribution (amount, donor, timestamp).
Collaborators	Ledger, User

4.4 User Subsystem

Class User	
Responsibilities	Base class with shared info for all our users
Collaborators (uses)	Ledger

Class Donor	
Responsibilities	Extend User, initiates donations
Collaborators	Donation

Class Admin	
Responsibilities	Extend user manage system data
Collaborators	NeedRepository, Bundle controller

4.5 UI/Controller Subsystem

Class FundGoodDeedsApp	
Responsibilities	Entry Point, initialize ui and subsystems
Collaborators (uses)	NeedController, BundleController

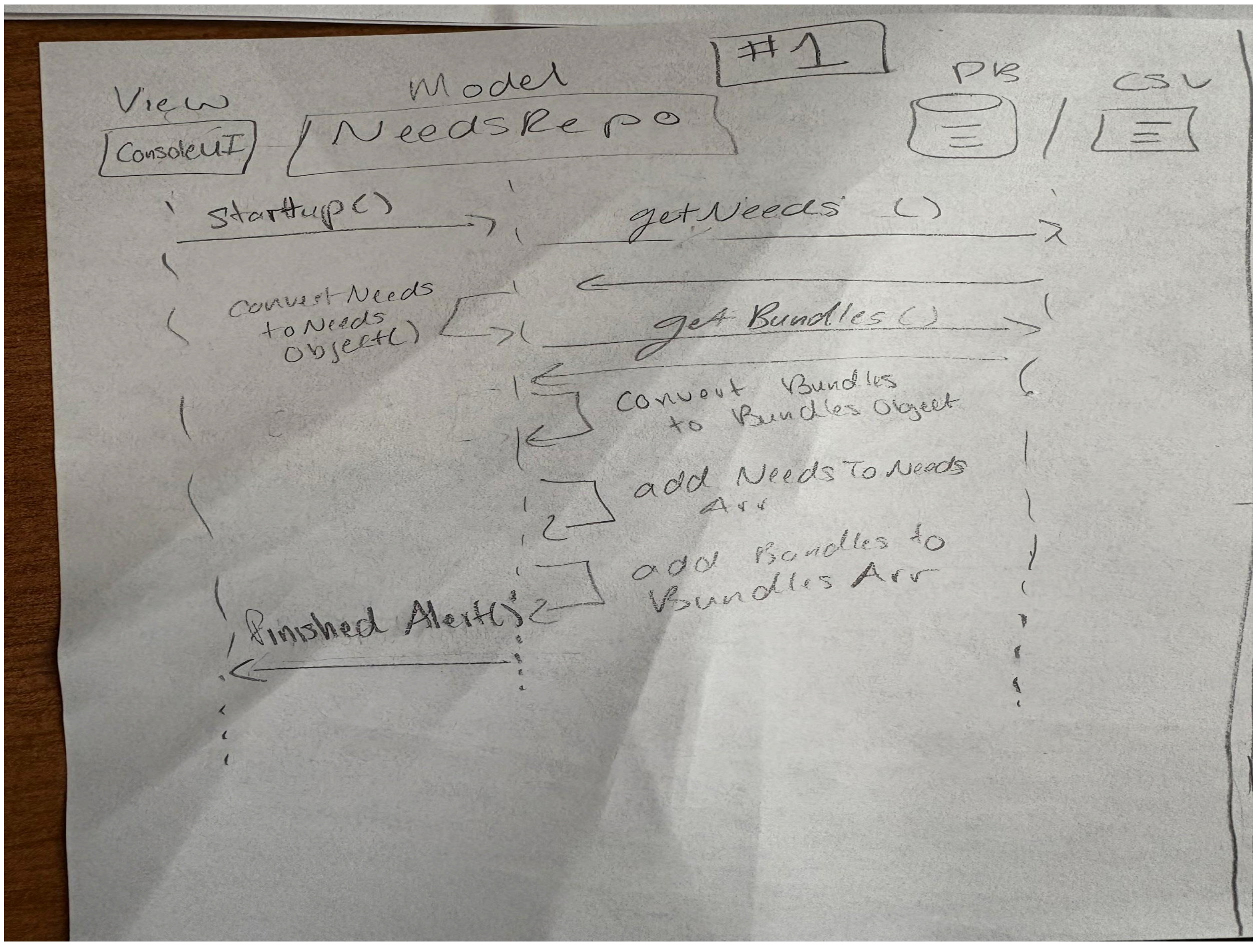
Class NeedController	
Responsibilities	Handles add/update/view Need actions.
Collaborators	NeedRepository, Ledger

Class BundleController	
Responsibilities	Manages bundle creation and fulfillment display.
Collaborators	Bundle, NeedController

Class UIObserver	
Responsibilities	Observes data model changes to refresh interface.
Collaborators	Need, Ledger

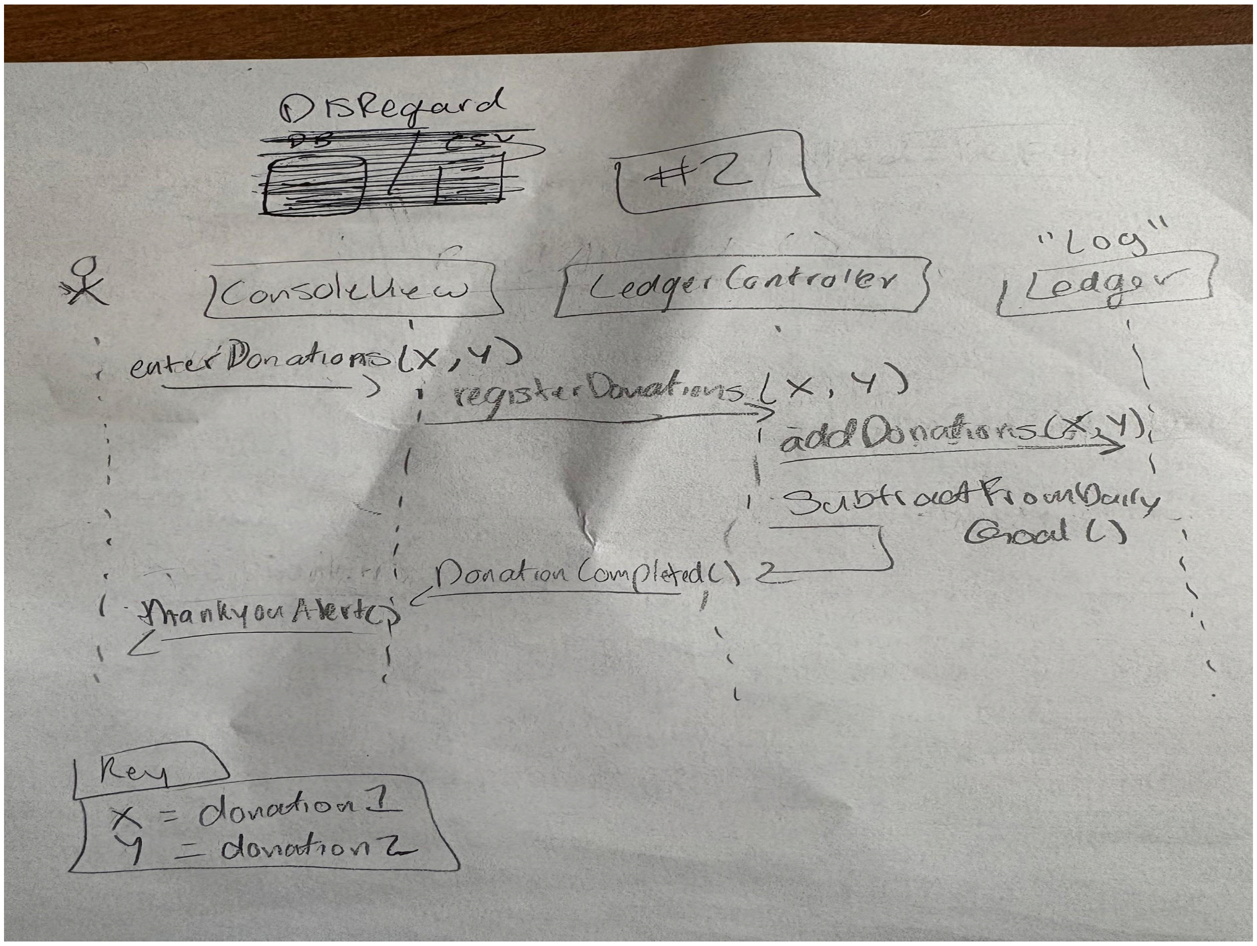
5 Sequence Diagrams

5.1 Description of labeled Sequence diagram #1 and (what feature / operation / scenario the diagram shows).



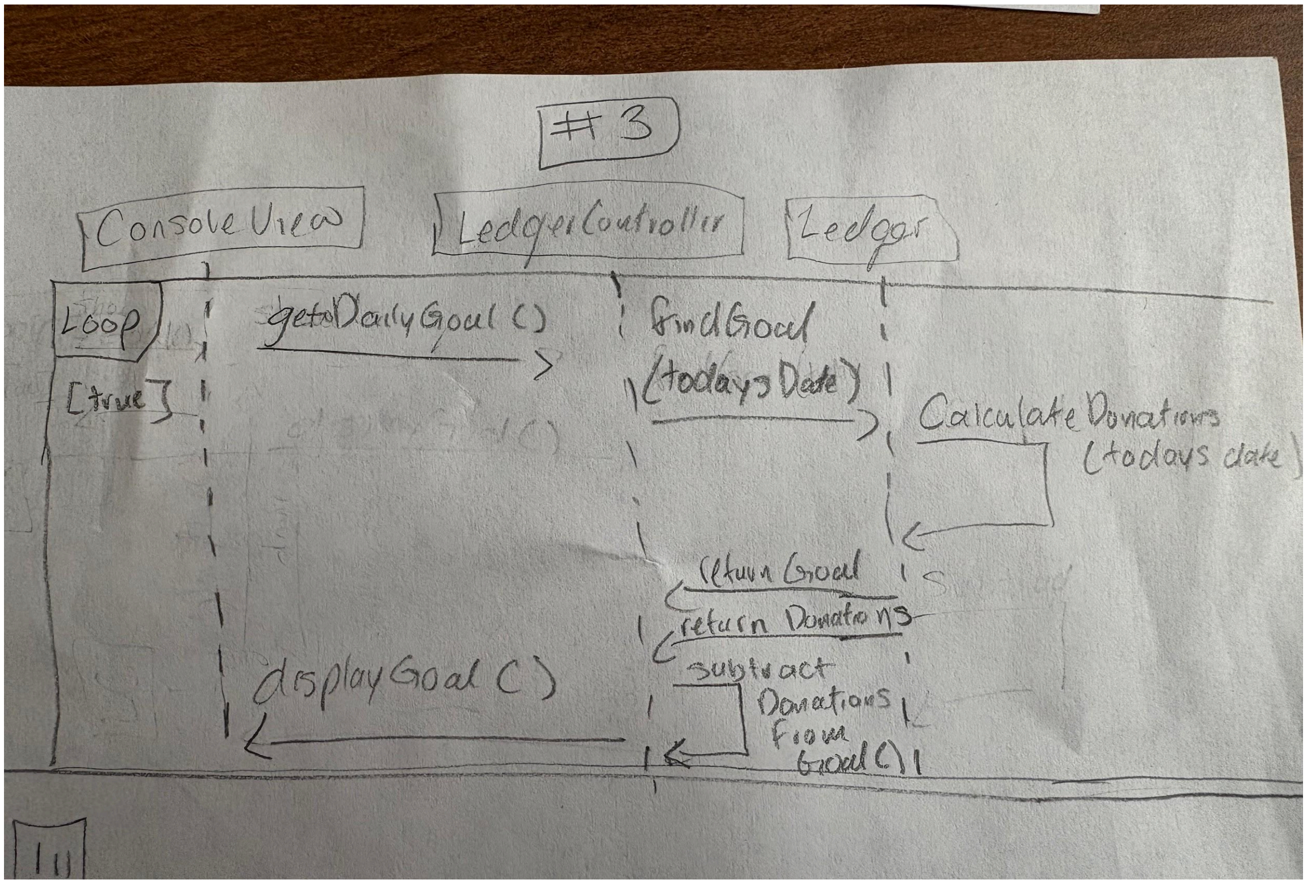
This diagram shows the initial startup of the CLI user interface. The ConsoleUI calls the startup method, NeedsRepo retrieves all the recorded needs/bundles from the CSV file and converts each record into their respective objects for further manipulation. After adding all the objects to their respective arrays, NeedsRepo alerts the ConsoleUI that all of its operations are completed and the data is ready to display to the console. (In an ideal scenario we will have the NeedsController between the ConsoleUI and NeedsRepo to respect the MVC architecture. - this is just our first draft)

5.2 Description of labeled Sequence diagram #2 (what feature / operation / scenario the diagram shows).



This diagram shows the MVC architecture of adding entries to the Ledger for data persistence. The user enters a donation in the ConsoleView, which is then registered in the LedgerController. The LedgerController handles appending the entry to the Ledger, then subtracts from the daily donation goal. LedgerController then notifies the ConsoleView that the donation has been completed. After receiving the notification, ConsoleView displays a thank you message to the user.

5.3 Description of labeled Sequence diagram #3 (what feature / operation / scenario the diagram shows).



This sequence diagram expands on the previously defined MVC architecture, giving some insight into the main loop. ConsoleView retrieves the daily goal from LedgerController. LedgerController then retrieves the total number of good deeds fulfilled on today's date from Ledger. During this, the Ledger calculates the number of donations on today's date, then returns the goal and number of donations to LedgerController. After receiving the data, LedgerController subtracts the number of donations from the daily goal, then notifies the ConsoleView to display the current goal after all operations are completed.

6 Pattern Usage

6.1 Observer Pattern

Observer Pattern	
Observer(s)	UIObserver (view layer), ReportView (optional)
Observable(s)	NeedRepository, Ledger
Notification Method	notifyObservers(), update()
Event Sources	Need status changes, Donation recorded

6.2 Composite Pattern

Composite Pattern	
Component	INeedComponent
Leaf	Need
Composite	Bundle
Client(s)	BundleController, NeedController, FundGoodDeedsApp

6.3 MVC

MVC	
Model	Need, Bundle, Donation, Ledger, NeedRepository
View	UIObserver, FundGoodDeedsApp display components
Controller	NeedController, BundleController

Glue/Contracts	Simple view models,using INeedComponent interface

6.4 Dependency Injection / Inversion (DI/IoC)

DI / DIP	
Abstractions	INeedComponent, (optionally ILedger, IRepository<T>)
Concrete Implementations	Need, Bundle, Ledger, NeedRepository
Composition Root	FundGoodDeedsApp (wires controllers, repos, observers)
Consumers	Controllers, views using interfaces rather than concretes

6.5 Repository Pattern

Repository Pattern	
Repository	NeedRepository (in-memory R1; DB later)
Aggregate Root	Need (and logically Bundle via INeedComponent)
Clients	NeedController, BundleController, Admin workflows

7 RATIONALE

The system is organized using the Model–View–Controller (MVC) architecture and the Composite pattern, with clear separation of responsibilities among classes.

Benefits, a huge separation of concerns. The Model layer includes classes such as NeedComponent, BasicNeed, Bundle, NeedsRepository, LedgerEntry, and LedgerRepository, which manage all data and business logic. The Composite relationship between Bundle and NeedComponent allows bundles to contain both basic needs and other bundles, providing a flexible way to represent complex financial structures without duplicating logic. Repositories isolate file storage and retrieval, allowing other components to focus on logic and presentation. The Controllers (NeedsController and LedgerController) coordinate data flow and enforce rules between the Model and the user-facing View. Importantly, the NeedsController manages creation and validation of BasicNeed and Bundle objects.

Drawbacks, the MVC architecture requires a controller to be made for each view. In this case, if we add another logging element to the system we would need to build a controller for it. The repository classes must contain lots of data validation from the csv files as the logs are the main source of truth. If they contain errors then the system fails. The main drawback is that this design introduces more classes and relationships, which increases complexity and coordination effort.

10/10/2025 – Component vs. Composite

Chose Composite (INeedComponent → Need/Bundle) so totals and fulfillment logic work identically for single needs and groups. This avoids duplicate code and simplifies UI/controllers.

10/12/2025 – Central Ledger

Introduced Ledger as the single source of truth for donations/transactions. Keeps financial logic cohesive and auditable. This hopes to prevent scattering donation math across controllers.

10/13/2025 – MVC Separation

Refactored to MVC where controllers handle orchestration, views only render,models encapsulate rules. This reduced coupling and made unit tests for business rules straightforward.

10/14/2025 – Observer for Live Updates

Added Observer so UI auto-refreshes on Need/Ledger changes. This removed manual refresh calls from controllers and clarified responsibilities.

10/15/2025 – Repository Abstraction

Adopted Repository (NeedRepository) to isolate storage concerns. R1 uses in-memory lists; R2 can switch to DB without controller changes.

10/16/2025 – DI at Composition Root

Applied Dependency Inversion by wiring interfaces in FundGoodDeedsApp. Enables mocking NeedRepository/Ledger in tests and future strategy swaps.

10/17/2025 – Donation Allocation Strategies (Future-proofing)

Defined a Strategy interface for bundle allocation. R1 defaults to Equal Split; left hooks for Priority/Single-Need strategies required by stakeholders.

10/18/2025 – Rejected Alternatives

Rejected controller-direct DB calls (tight coupling), and view-driven business logic (violates MVC). Also rejected hard-coded allocation rules (blocks future change).

10/19/2025 – Subsystems & Contracts

Locked subsystem boundaries (Need, Bundle, Ledger, User, UI/Controllers) and interfaces (INeedComponent).