[FINAL RELEASE]

Project Design Document

Team 1a - JSCOPE

Oliver Gomes Jr. (odg1896@rit.edu)

Patrick Lebeau (pml4324@rit.edu)

Jonathan Ho (jlh5360@rit.edu)

Connor Bashaw (cdb9772@rit.edu)

Project Summary

FundGoodDeeds App (Version 2.0) is a powerful yet user-friendly tool designed to help individuals and families manage their financial needs and funding sources with clarity and precision. The system keeps track of everyday needs, bundles items, and different types of income using straightforward CSV Files.By organizing basic needs, bundles of essentials, and funding sources in simple CSV files, the app keeps a daily ledger of fulfilled needs, received funds, and thresholds, making it easy to track spending, income, and net financial impact at a glance. Users can add, edit, or remove needs and funding sources, set daily financial goals, and maintain an accurate record of their financial activity, all through an intuitive interface that ensures data is always up-to-date and error-free.

With FundGoodDeeds, the goal is to help users plan out their daily finances without needing to dig through a bunch of numbers. The app automatically calculates total costs of expenses, tracks income, and net balances, while alerting when expenses exceed predefined thresholds. Bundles of essential items, recurring funding sources, and historical logs provide valuable insights for budgeting and decision-making. Whether you're managing monthly expenses, tracking income sources, or ensuring your essential needs are met efficiently, FundGoodDeeds empowers users to stay organized, financially informed, and confident in their daily financial decisions.

Version 2.1 Expands the system with full authentication subsystem. Users can now create accounts, log in securely with encrypted passwords, and receive fully isolated personal financial information. Each user owns there own needs.csv, funding.csv, and log.csv stored in a unique directory under the applications data folder. This ensures complete privacy and clean multi user separation while preserving our CSV data model.

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.

Architecture

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

MasterController Integration Enhancement: To support Version 2.0's expanded functionality, the architecture now introduces a **MasterController**, which acts as the central orchestration layer for the entire system. While the original V1 architecture connected the ConsoleView directly to multiple controllers, the improved V2 architecture routes all communication through the MasterController. This reduces UI-controller coupling, simplifies initialization, and ensures all subsystems are loaded, updated, and saved in a coordinated manner. It holds the current selectedDate and maintains date constraints across subsystems. On shutdown, this controller calls the save method on all repositories. For each user action, it will parse the user command and delegate it to the appropriate controller. It also handles errors that might span across several subsystems.

The MasterController manages:

- System-wide **selected date** logic and enforcement of V2 constraints
- Bootstrapping of all repositories (Needs, Funding Sources, Ledger)
- Coordinated saving of all CSV-backed storage files
- Wiring together of controllers, repositories, and the ConsoleView using dependency injection
- Distribution of observer notifications across Need, Ledger, and Funding subsystems

This architectural addition significantly increases system cohesion while lowering the burden on individual controllers and the view layer. This made the overall flow way more predictable and removed a lot of confusion we saw in early prototypes. To increase multiple independent users , V2.1 introduces a dedicated user consisting of **User**, and **User Store** and login based UI Components. Authentication occurs before any subsystem is initialized. Once a user logs in and **Master controller** gets ahold of the current user Contex and shares it to controllers and repos. Repos in turn delegate to the **CSV Manager** which switches its active path to the users dedicated data directory. As a result, all needs, funding sources, and ledger entries become per user assets that cant be leaked between accounts. This maintains strict **separation of concerns** while enabling multi user operation.

Patterns

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. The repositories (NeedRepository, FundingRepository,Ledger Repository) act as Observable Subjects and Console View and SwingUIView operate as observable subjects

Repository Pattern (custom pattern) allows for abstraction of stored data access. All CRUD (Create, Read, Update, and Delete) for Need, Funding, and Ledger entities pass through repositories instead of directly manipulating CSV files.This helps to keep our code organized, easier to maintain, and output back to a data store upon request. This also makes it possible to swap CSV storage for a database in later releases

Principles

DIP via Dependency Injection allows for extensibility, since all of the objects being passed in any constructor will depend on the same interface, we can interchange the objects without any conflicts. Version 2 improves dependency management by introducing a composition root (FundGoodDeedsApp) and wiring controllers, repositories, and views in one place. Controllers no longer construct repos directly, instead they receive them via constructor injection. We use the need Component interface as an abstraction for need and bundle. But Most controllers still depend on concrete repository types, full DIP repository interfaces like INeedRepository and so on are planned for future releases.

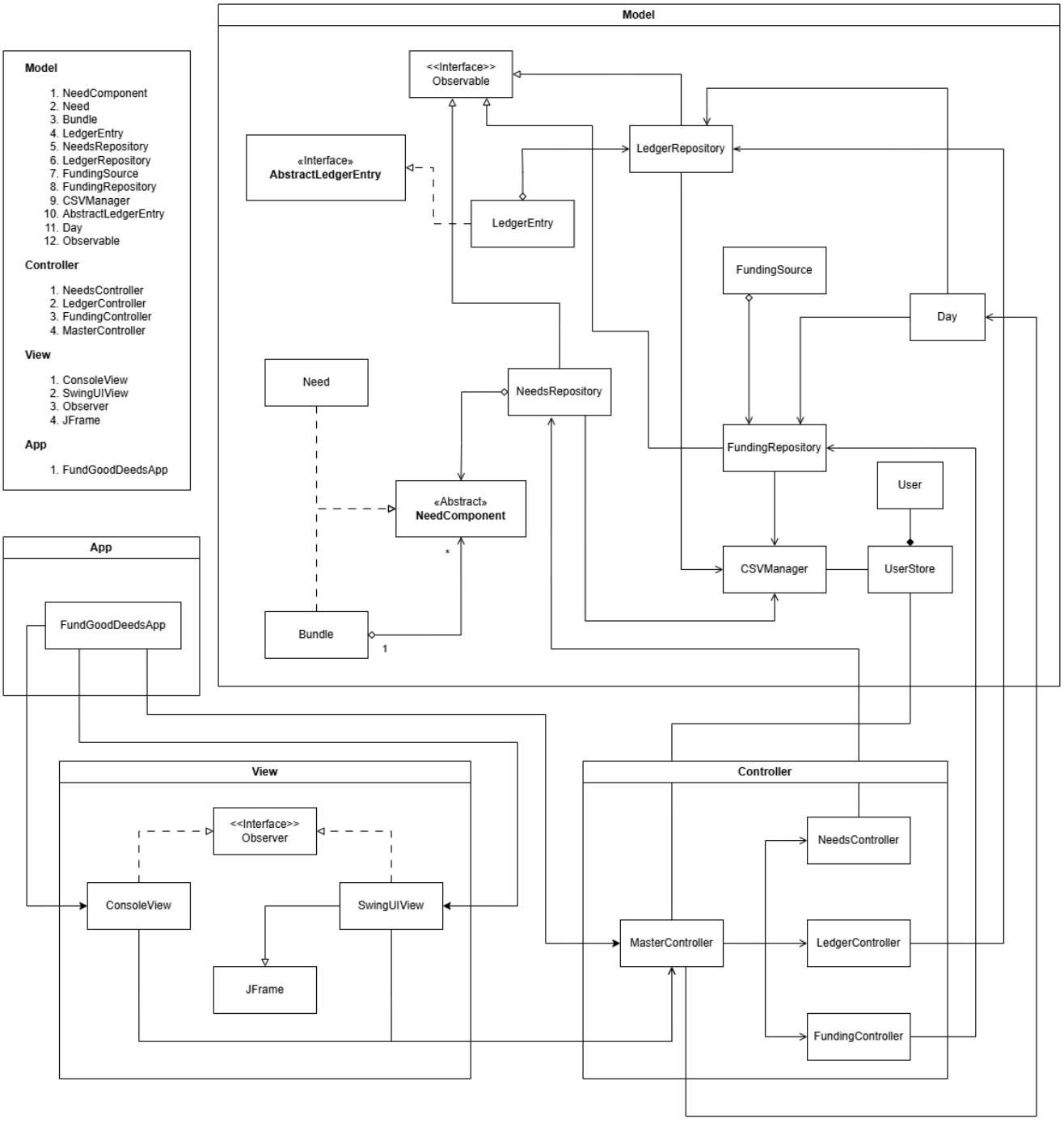
Early prototypes revealed tight coupling between UI and logic, these were then handled through interface abstractions. Dependency inversion was applied to allow testing and future integrations with external API’s or databases.

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.

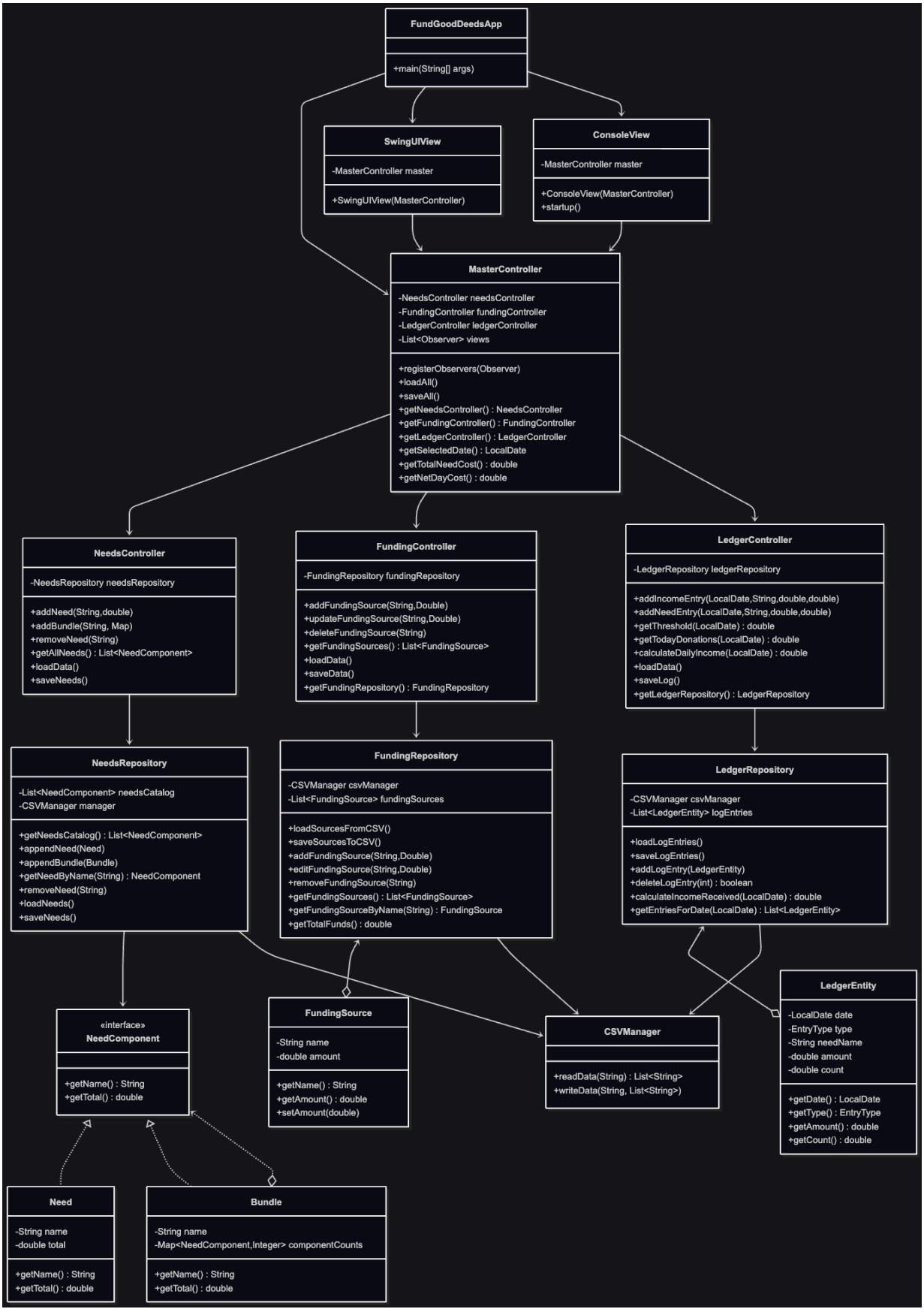
DRY Principle allows for functionality that is expected to be repeated to exist as an abstraction instead of repeated code. Our design holds a csv parser as multiple entities in the system are derived from data files and not always created by the user.

SRP (Single Responsibility Principle) unlike our first iteration, we will ensure our classes in our view are only responsible for displaying information and sending user input to the controllers. We do not want extra processing logic in the view but maintained in our controllers.

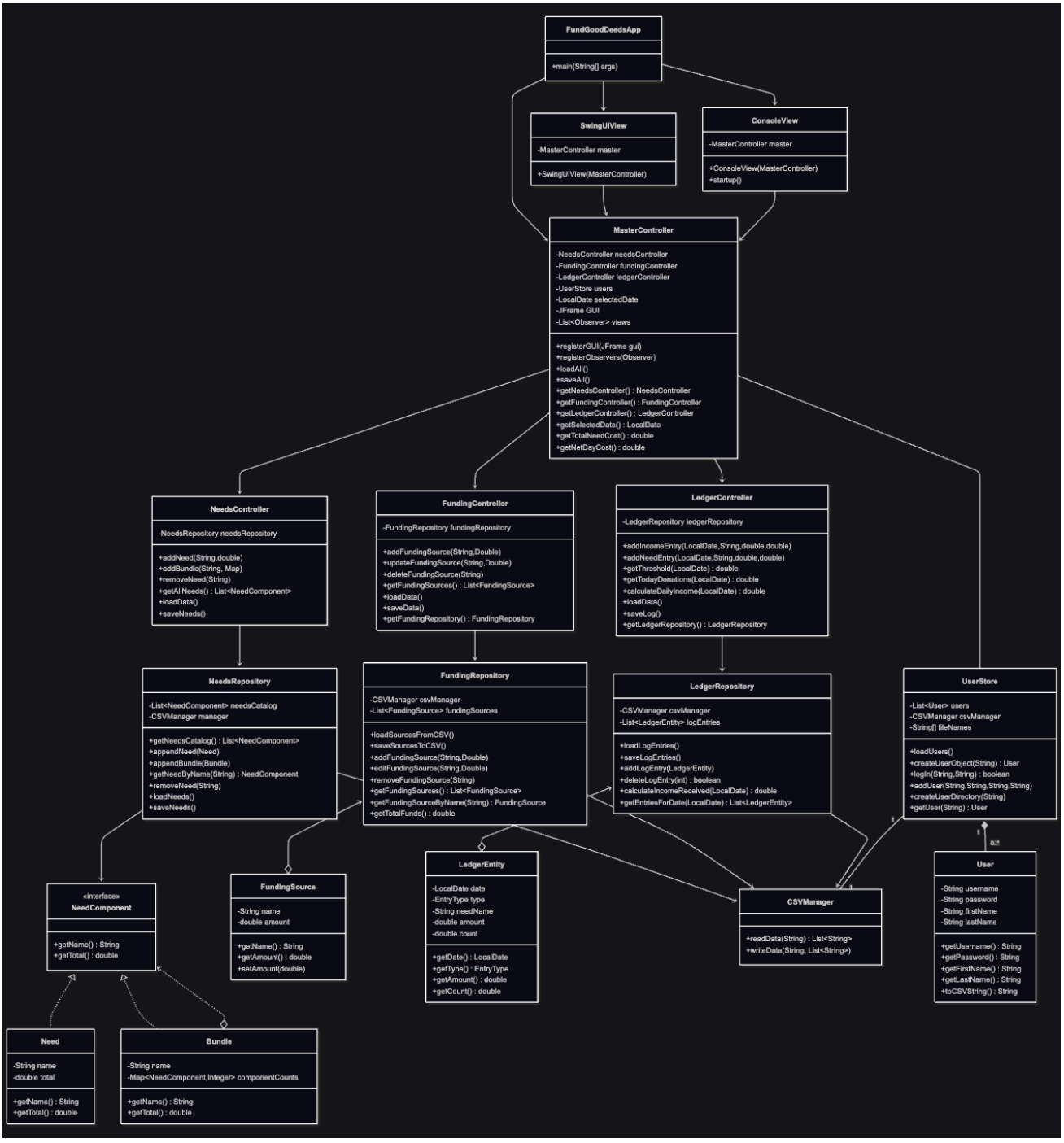
Full System Architecture (MVC)



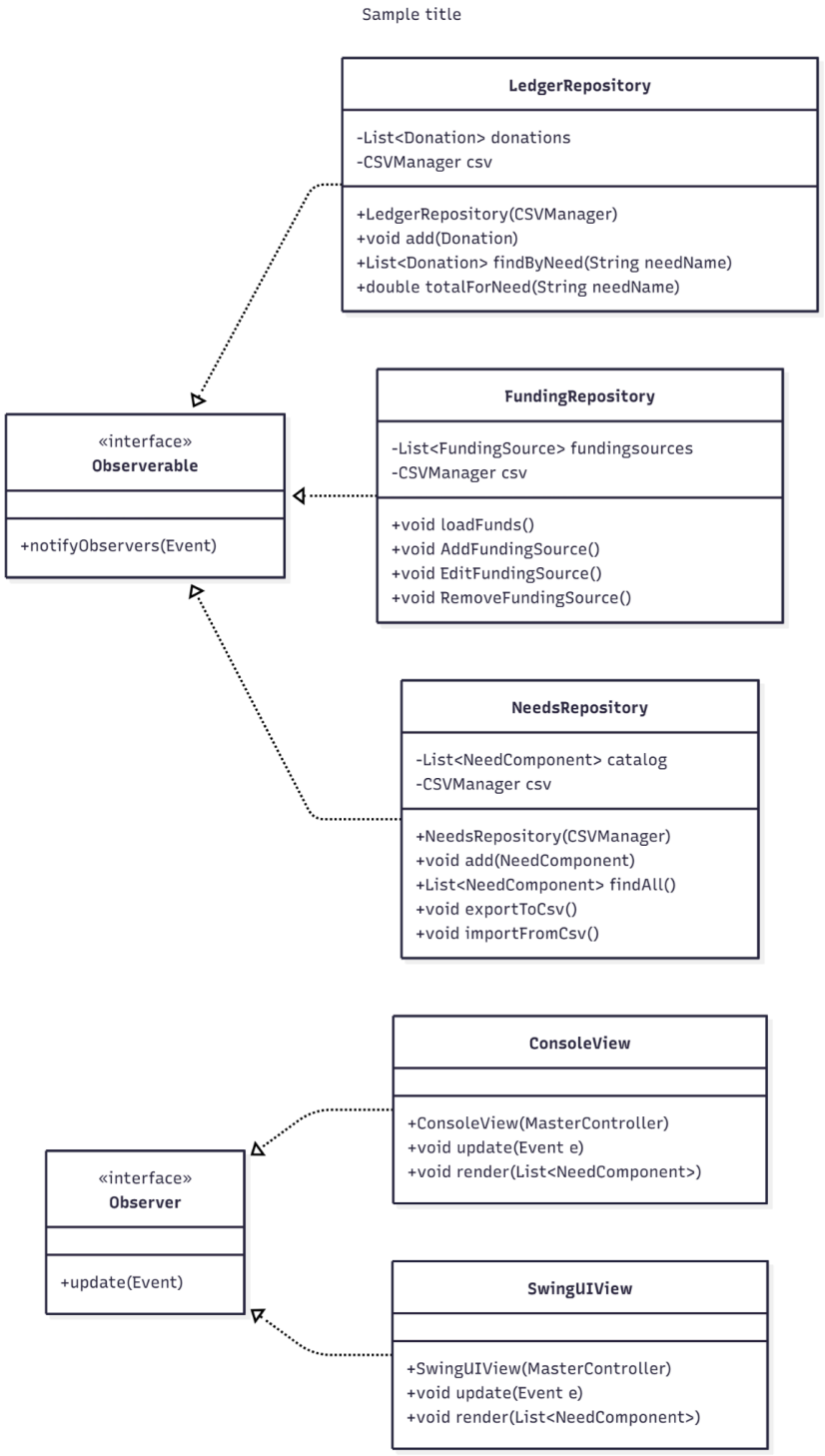
V2 Detailed Architecture


FundGoodDeedsApp
+main(String[] args)
SwingUIView
-MasterController master
+SwingUIView(MasterController)
ConsoleView
-MasterController master
+ConsoleView(MasterController)
+startup()
MasterController
-NeedsController needsController
-FundingController fundingController
-LedgerController ledgerController
-List<Observer> views
+registerObservers(Observer)
+loadAll()
+saveAll()
+getNeedsController() : NeedsController
+getFundingController() : FundingController
+getLedgerController() : LedgerController
+getSelectedDate() : LocalDate
+getTotalNeedCost() : double
+getNetDayCost() : double
NeedsController
-NeedsRepository needsRepository
+addNeed(String,double)
+addBundle(String, Map)
+removeNeed(String)
+getAllNeeds() : List<NeedComponent>
+loadData()
+saveNeeds()
FundingController
-FundingRepository fundingRepository
+addFundingSource(String,Double)
+updateFundingSource(String,Double)
+deleteFundingSource(String)
+getFundingSources() : List<FundingSource>
+loadData()
+saveData()
+getFundingRepository() : FundingRepository
LedgerController
-LedgerRepository ledgerRepository
+addIncomeEntry(LocalDate,String,double,double)
+addNeedEntry(LocalDate,String,double,double)
+getThreshold(LocalDate) : double
+getTodayDonations(LocalDate) : double
+calculateDailyIncome(LocalDate) : double
+loadData()
+saveLog()
+getLedgerRepository() : LedgerRepository
NeedsRepository
-List<NeedComponent> needsCatalog
-CSVManager manager
+getNeedsCatalog() : List<NeedComponent>
+appendNeed(Need)
+appendBundle(Bundle)
+getNeedByName(String) : NeedComponent
+removeNeed(String)
+loadNeeds()
+saveNeeds()
FundingRepository
-CSVManager csvManager
-List<FundingSource> fundingSources
+loadSourcesFromCSV()
+saveSourcesToCSV()
+addFundingSource(String,Double)
+editFundingSource(String,Double)
+removeFundingSource(String)
+getFundingSources() : List<FundingSource>
+getFundingSourceByName(String) : FundingSource
+getTotalFunds() : double
LedgerRepository
-CSVManager csvManager
-List<LedgerEntity> logEntries
+loadLogEntries()
+saveLogEntries()
+addLogEntry(LedgerEntity)
+deleteLogEntry(int) : boolean
+calculateIncomeReceived(LocalDate) : double
+getEntriesForDate(LocalDate) : List<LedgerEntity>
«interface»
NeedComponent
+getName() : String
+getTotal() : double
FundingSource
-String name
-double amount
+getName() : String
+getAmount() : double
+setAmount(double)
CSVManager
+readData(String) : List<String>
+writeData(String, List<String>)
LedgerEntity
-LocalDate date
-EntryType type
-String needName
-double amount
-double count
+getDate() : LocalDate
+getType() : EntryType
+getAmount() : double
+getCount() : double
Need
-String name
-double total
+getName() : String
+getTotal() : double
Bundle
-String name
-Map<NeedComponent,Integer> componentCounts
+getName() : String
+getTotal() : double

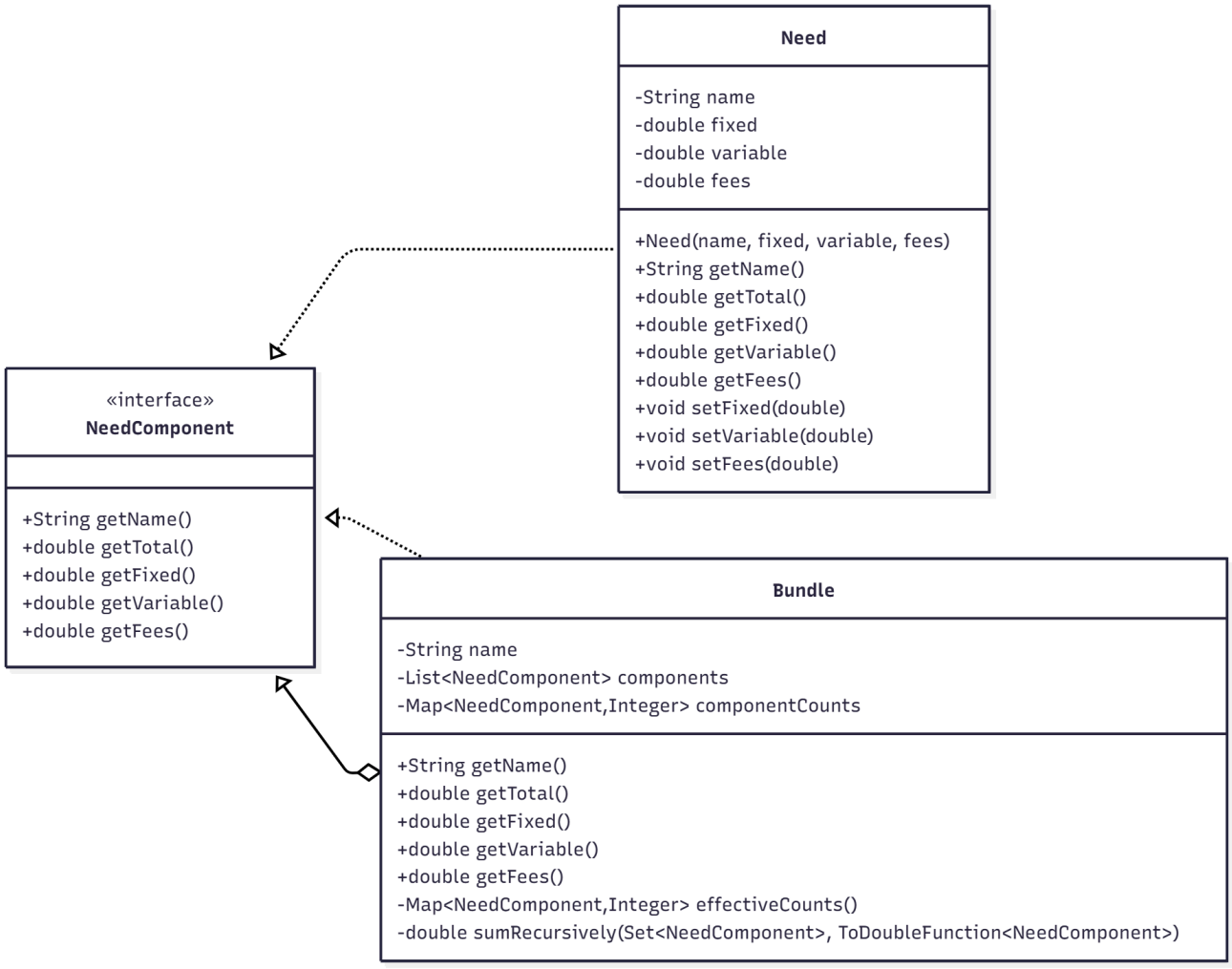
V2.1 - Revised Architecture with Users



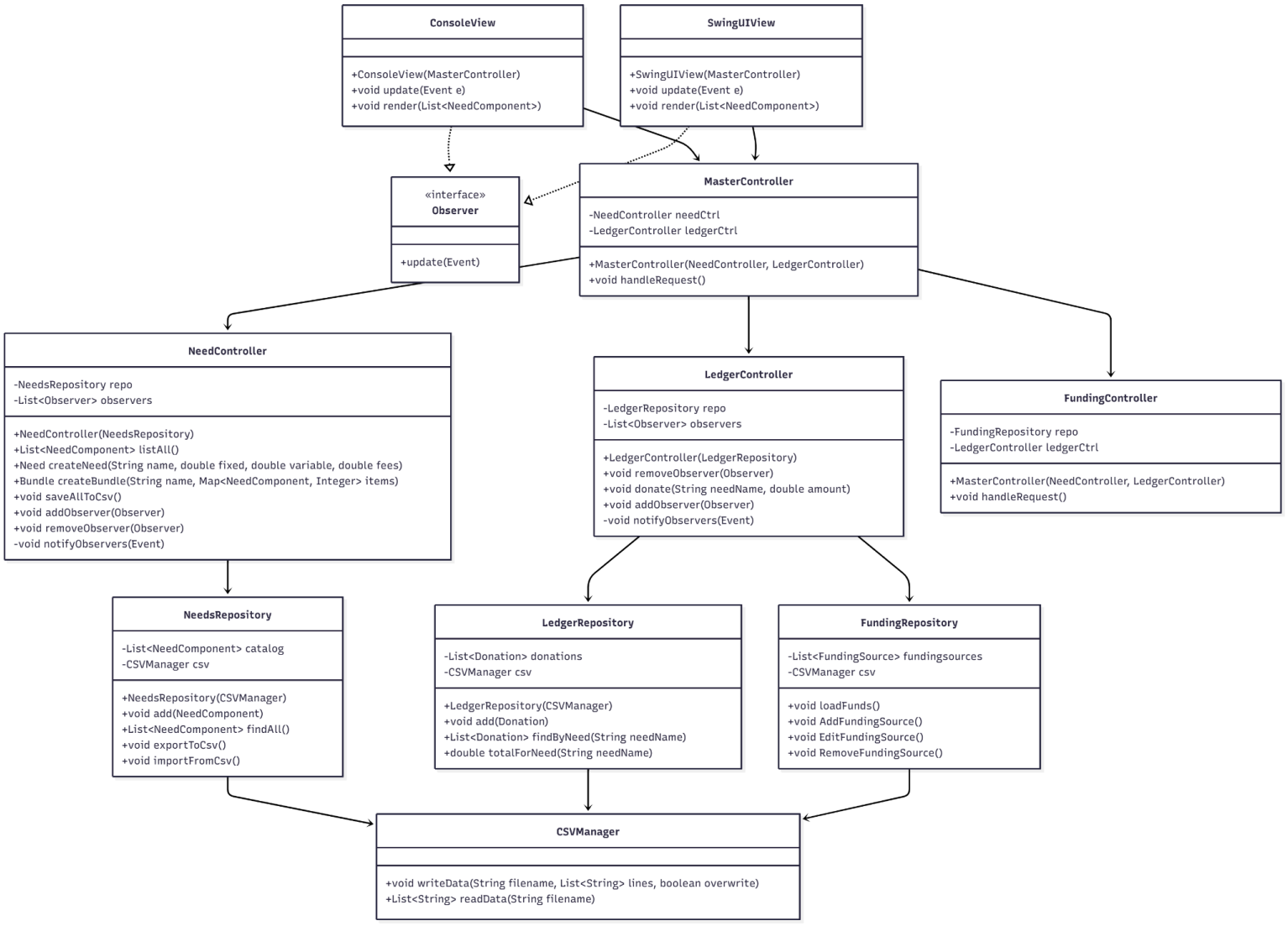
V2 Observer Pattern



V2 Composite Pattern (same as V1)



V2 - Views and Master Controller Coordination



V2

- **Need Management Subsystem**
  - Manages creation, retrieval, and status of Needs.
- **Bundle Management Subsystem**
  - Groups related Needs using the Composite pattern.
- **Ledger Subsystem**
  - Records expenses, calculates totals, and ensures transaction integrity.
- **Funding Subsystem**
  - Manages the definition, storage, and recording of financial income sources and dates.
- **UI / Controller Subsystem**
  - Manages user interactions, events, and display updates.
- **Users**
  - Allows creation and storage of user information per user. Users can signup, login and begin tracking finances. The data of one user is stored in a individual folder allowing us to have data persistence on users and security where other users can not access someone else's data.

Updated V2 System Architecture (with MasterController)

Version 2 introduces an expanded MVC architecture connected through the new MasterController. This controller coordinates all business logic and storage operations across the following subsystems:

- **Need Management Subsystem**
- **Bundle Management Subsystem**
- **Ledger Subsystem**
- **Funding Subsystem**
- **UI / Controller Subsystem**
- **Users**

The MasterController stands between the UI and subsystem controllers, simplifying orchestration, enforcing date constraints, and enabling clean dependency injection. Each subsystem communicates through strongly bounded interfaces and repositories, ensuring scalable growth and clear separation of concerns.

FundGoodDeedsApp acts as the default package and composition root. It Instantiates the Master Controller, all repositories, and both views, then wires them together using dependency injection before handling control to the UI loop.

The application now supports multiple authenticated users. Each user has a unique account with a username encrypted hashed password, managed by a **UserStore component.** After authentication, all needs, Funding, and Ledger Data is transparently visible to that user via the **CSVManager.** The Repositories never see raw passwords and never access another user's data. They only read and write CSV files inside the authenticated users directory. This preserves the existing CSV- based data persistence while enforcing user isolation.

Subsystems

Subsystem Overview (Updated for V2.0)

The system is decomposed into 6 major subsystems, each with clearly defined responsibilities and interactions:

1. **Need Management Subsystem**
   Handles creation, editing, and retrieval of basic needs. Implements Composite-compatible Need objects. It also enforces rules that bundles must reference valid needs.
2. **Bundle Management Subsystem**
   Groups need recursively via the Composite Pattern using Bundle. Supports hierarchical cost computation.
3. **Ledger Subsystem**
   Maintains daily logs of fulfilled needs, funding sources used, goals, and funds. Make's sure all financial entries follow the new date rules we added in V2, including the 7 day restriction.
4. **Funding Subsystem**
   Stores and manages definable financial sources, such as Paycheck or Student Loan, tracking the dollar value of each unit received. Provides lookup and validation logic for daily ledger income entries. This replaced all donation logic from V1.
5. **UI / Controller Subsystem**
   Contains ConsoleView, all controllers, and the new MasterController. Acts as the coordination interface between user actions and subsystem operations.
6. **User Subsystem**
   Provide secure Login signup and per data user scoping

a. Need Management

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

Class NeedComponent

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

Class NeedRepository	
Responsibilities	Handles loading, retrieving, and saving of Needs/Bundle data to the CSV file.
Collaborators (inheritance)	NeedsController, LedgerController

Class CSVManager	
Responsibilities	Handles the reading and writing of CSV data to the CSV file.
Collaborators (inheritance)	NeedsRepository, LedgerRepository

b. Bundle Management

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

Class NeedRepository	
Responsibilities	Handles loading, retrieving, and saving of Needs/Bundle data to the CSV file.
Collaborators (inheritance)	NeedsController, LedgerController, CSVManager

Class CSVManager	
Responsibilities	Handles the reading and writing of CSV data to the CSV file.

Collaborators (inheritance)	NeedsRepository, LedgerRepository, FundingRepository

c. Ledger subsystem

Class AbstractLedgerEntity	
Responsibilities	An abstraction layer for all Ledger entries to ensure all concrete entries share common methods.

Class LedgerEntity	
Responsibilities	Records all funding entries and computes totals.
Collaborators (uses)	Need, Bundle, AbstractLedgerEntity

Class LedgerRepository	
Responsibilities	Handles loading, retrieving, and saving of LedgerEntity data into the CSV file.
Collaborators (inheritance)	LedgerController, CSVManager

Daily Ledger Entry Types (Updated for V2.0)

The ledger uses a unified CSV format (log.csv) containing multiple types of financial entries. This expands beyond V1, which only tracked basic donation events.

Prefix	Type	Format	Description
f	Funds	yyyy,mm,dd,f,amount	Sets available funds for the day
t	Threshold	yyyy,mm,dd,t,amount	Sets the financial threshold for the day

n	Need Fulfillment	yyyy,mm,dd,n,needName,count	Records instances of fulfilled needs
i	Income from Funding Source	yyyy,mm,dd,i,sourceName,units	Records units of income received from a funding source

The LedgerRepository parses all four entry types, maintains day-by-day calculations, and applies Version 2 rules such as:

- Selected date may not be older than 7 days
- Threshold on the current date can only be changed if no needs or income entries exist
- Funds and threshold changes may not modify past dates

Class CSVManager	
Responsibilities	Handles the reading and writing of CSV data to the CSV file.
Collaborators (inheritance)	NeedsRepository, LedgerRepository, FundingRepository

d. Funding Subsystem

Class FundingSource	
Responsibilities	Represents a single definable source of income (name, amount per unit). Tracks the base value for one unit of funding.

Class FundingRepository	
Responsibilities	Handles loading, retrieving, and saving of FundingSource data into the CSV file. Ensures data integrity and access control for funding sources.
Collaborators (inheritance)	FundingController

Class CSVManager	
Responsibilities	Handles the reading and writing of CSV data to the CSV file.
Collaborators (inheritance)	NeedsRepository, LedgerRepository, FundingRepository

Class Day	
Responsibilities	Abstraction for collecting information for a single day (i.e expenses, income, previous day, next day)
Collaborators (inheritance)	LedgerRepo, FundingRepo, MasterController

Funding Subsystem

This sub system replaced all donation logic from V1 after realizing funding sources were more realistic when talking in terms of budgeting.

Class FundingSource

Represents a definable income stream with:

- name
- amountPerUnit — how much money one “unit” of the funding source provides

Examples:

- Paycheck (1 unit = 1800.00)
- Hourly Work (1 unit = 22.50)
- Student Loan Disbursement (1 unit = 5000.00)

Class FundingSourceRepository

Handles loading, retrieving, and saving funding source data from funding.csv.
Responsibilities include:

- Parsing rows of the form: i,name,amountPerUnit
- Ensuring unique funding source names
- Supporting lookups by name for ledger calculations
- Maintaining synchronized in-memory state and notifying observers

Class FundingSourceController

Coordinates all operations related to funding sources.
Responsibilities include:

- Adding/editing funding sources
- Delivering read-only lists to the UI
- Enforcing naming rules and data validation
- Notifying views when funding data is updated

e. UI/Controller Subsystem

Class FundGoodDeedsApp	
Responsibilities	Entry Point, initialize UI and subsystems.
Collaborators (uses)	NeedController, LedgerController, ConsoleView

Class NeedController	
Responsibilities	Handles add/update/view Need and Bundle actions.
Collaborators	NeedRepository, NeedComponent, ConsoleView

Class LedgerController	
Responsibilities	Handles add/update/view LedgerEntity actions.
Collaborators	LedgerRepository, ConsoleView

Class FundingController	
Responsibilities	Handles add/update/view FundingSource actions.
Collaborators (uses)	FundingRepository, LedgerController, ConsoleView

Class MasterController	
Responsibilities	Handles all incoming requests from the view to call a specific controller
Collaborators (uses)	FundingRepository, LedgerController, FundingController, SwingUI View, ConsoleView

Class ConsoleView	
Responsibilities	Observes data model changes to refresh the interface.
Collaborators	Master Controller

Class SwingUIView	
Responsibilities	Observes data model changes to refresh the Swing interface.
Collaborators	Master Controller

Class LoginPanel	
Responsibilities	Gather username and password. Call mastercontroller.loginSuccessful(). Provide sign up support and error messaging.
Collaborators	UserFrame, MasterController

Class UserFrame	
Responsibilities	Encapsulate login display. Trigger Main UI launch on success
Collaborators (uses)	LoginPanel, FundGoodDeedsApp

f. User Subsystem

Class User	
Responsibilities	Represent a authenticated user, store user name , name fields, and password hashing. Convert itself to and from csv format
Collaborators (uses)	userStore

Class UserStore

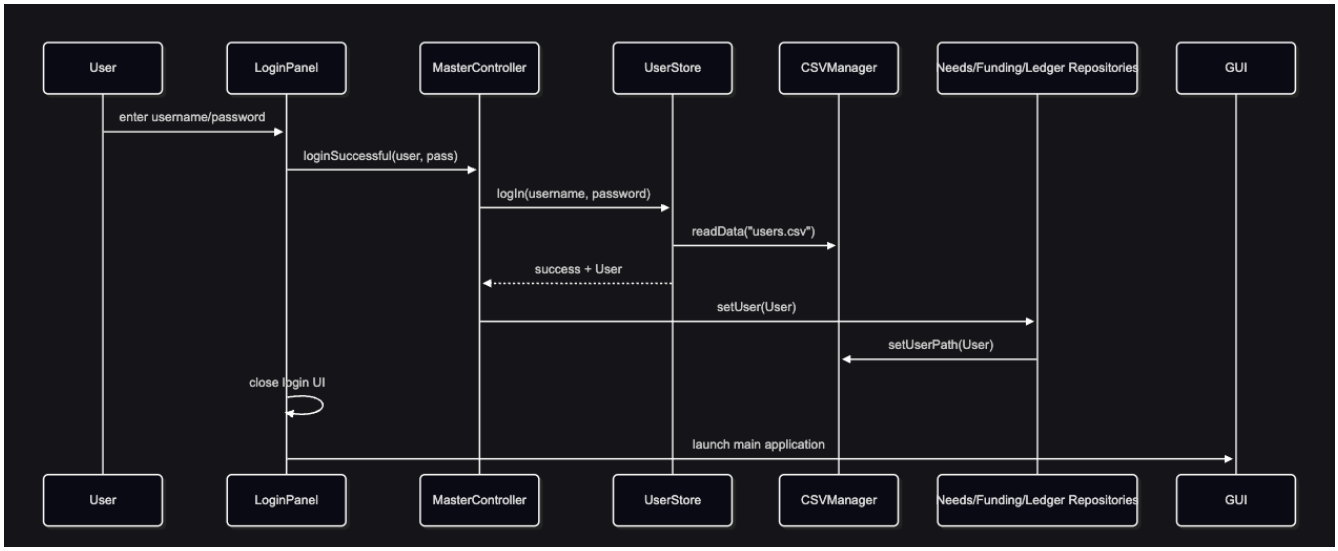
Responsibilities	Load and save data for all users from user.csv. Hash passwords when creating users. Verify Login Credentials. Create Per user data directories. Provide lookup and uniqueness enforcement.
Collaborators	CSVManager, User, MasterController

Class LoginPanel	
Responsibilities	Gather username and password. Call mastercontroller.loginSuccesful(). Provide sign up support and error messaging.
Collaborators	UserFrame, MasterController

Class UserFrame	
Responsibilities	Encapsulate login display. Trigger Main UI launch on success
Collaborators (uses)	LoginPanel, FundGoodDeedsApp

Sequence Diagrams

g. Login and Authentication.

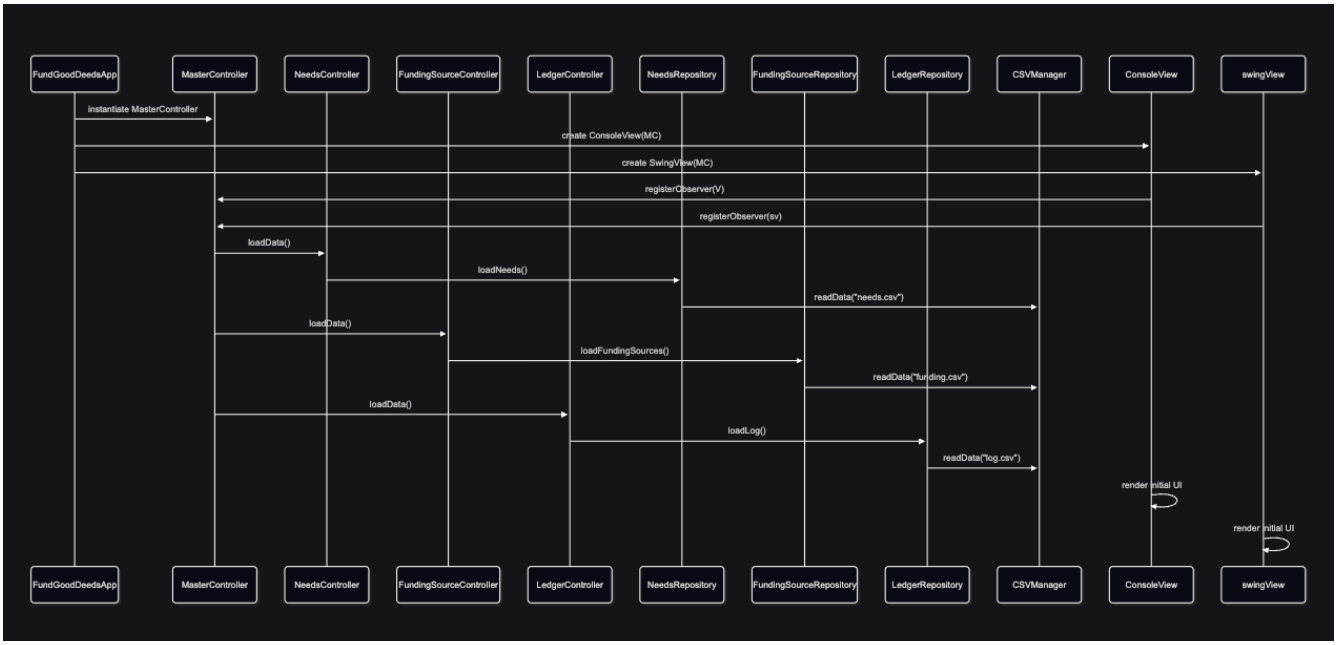


This is the full workflow before the user data is displayed or loaded. The user first enters the credentials into the **LoginPanel**. Which forwards the login to **MasterController.** The Controller delegates authentication to the **UserStore**, which validates the username and password against hashed entries in **users.csv**

If authentication succeeds the **MasterController** receives the authenticated **User** object and shares context with the **repositories**, through their controllers. Each repository updates its internal state to point the **CSVManager** toward the authenticated users dedicated data directory, ensuring that all subsequent reads and writes are fully user isolated.

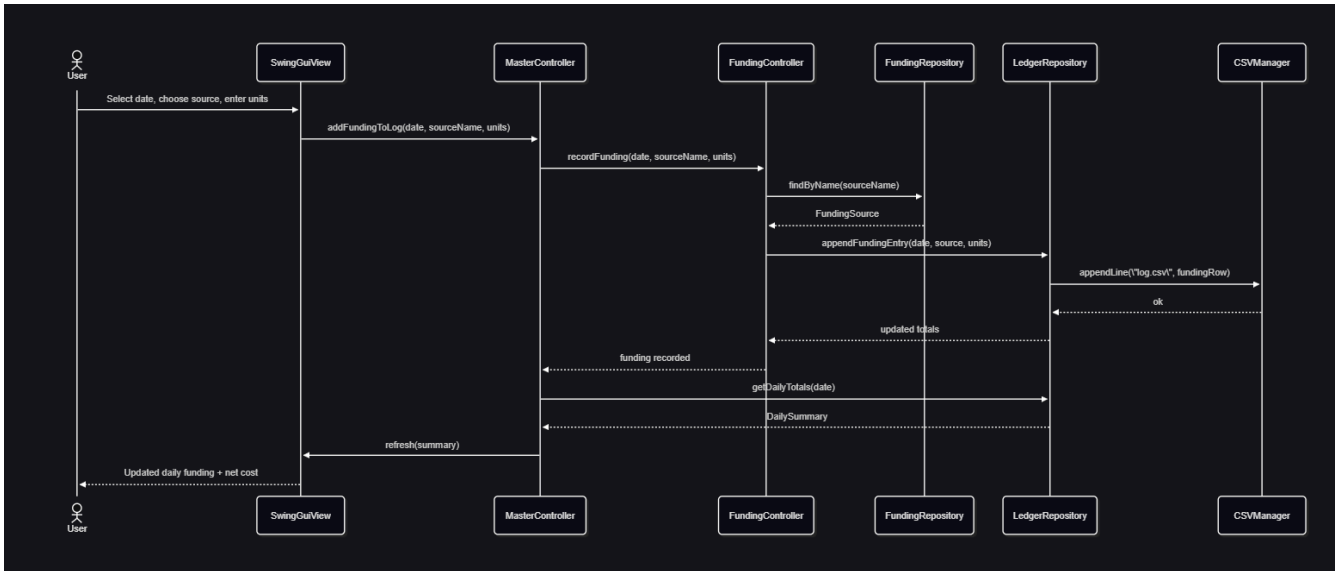
Once the system is correctly scoped to the authenticated user the **login UI** closes and controls the transition to the main **SwingUIView**. Enabling the user to interact with the personal needs, funds, and ledger data. This sequence establishes a secure boundary, no subsystem initializes or loads domain data until authentication is complete and the user context is fully wired into the architecture.

h. Startup Process



On application startup, the user launches the program, which initializes the **MasterController**, the core **GUI** components, and the **SwingUI**. The **MasterController** is responsible for **instantiating the remaining domain-specific controllers**. **Each controller then loads its respective system data** into the appropriate repository abstractions. Such as those managing needs, bundles, incomes, and dates. Once this initialization phase completes, the user can interact with the fully populated model through the UI, ensuring that all domain objects are ready for manipulation as the session begins.

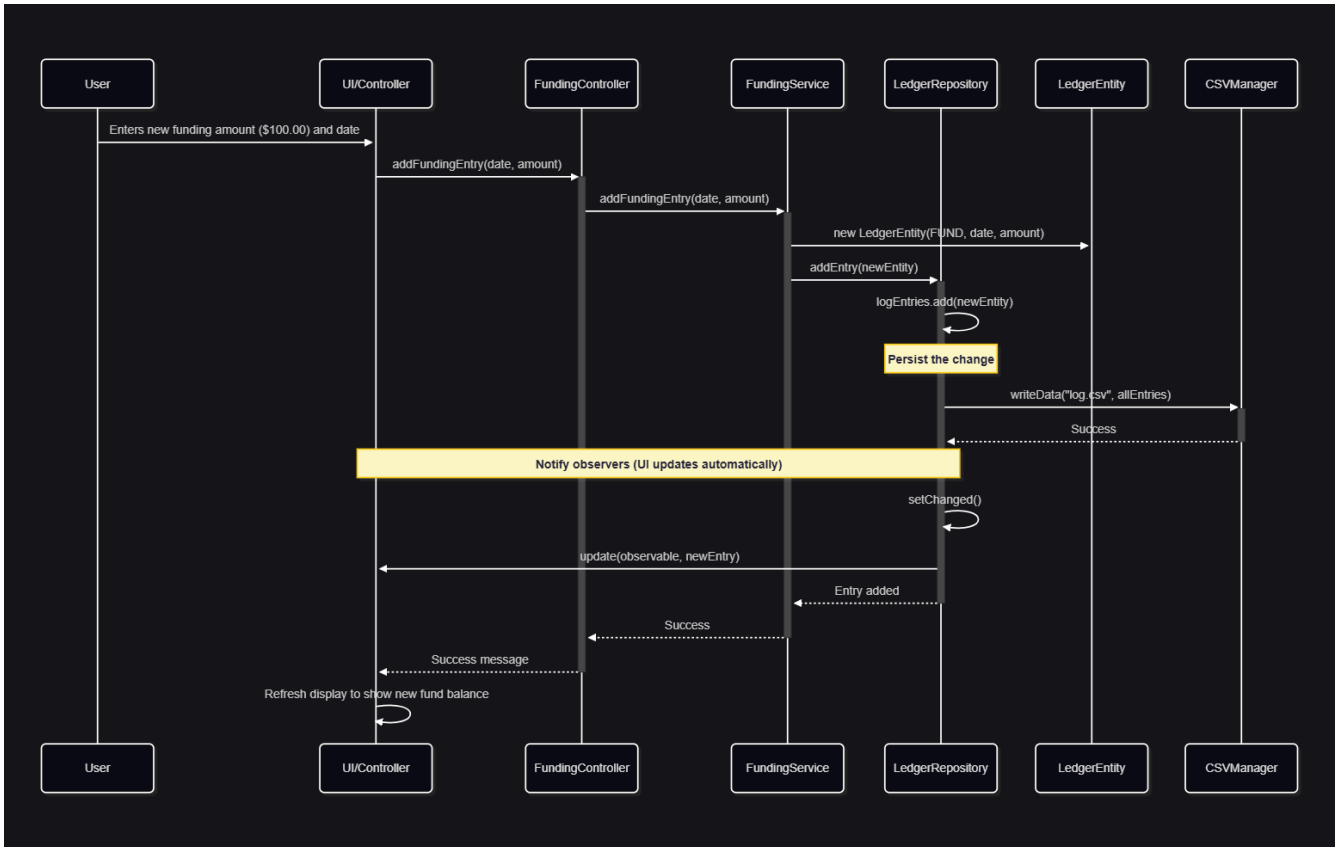
i. Adding an income source



In this scenario, the user is already authenticated and active within the system. The view layer prompts the user to select a data item, which is then forwarded to the **MasterController**. Based on the request context, the **MasterController** delegates the operation to the **FundingController**, which is responsible for resolving the original income source and applying the necessary updates.

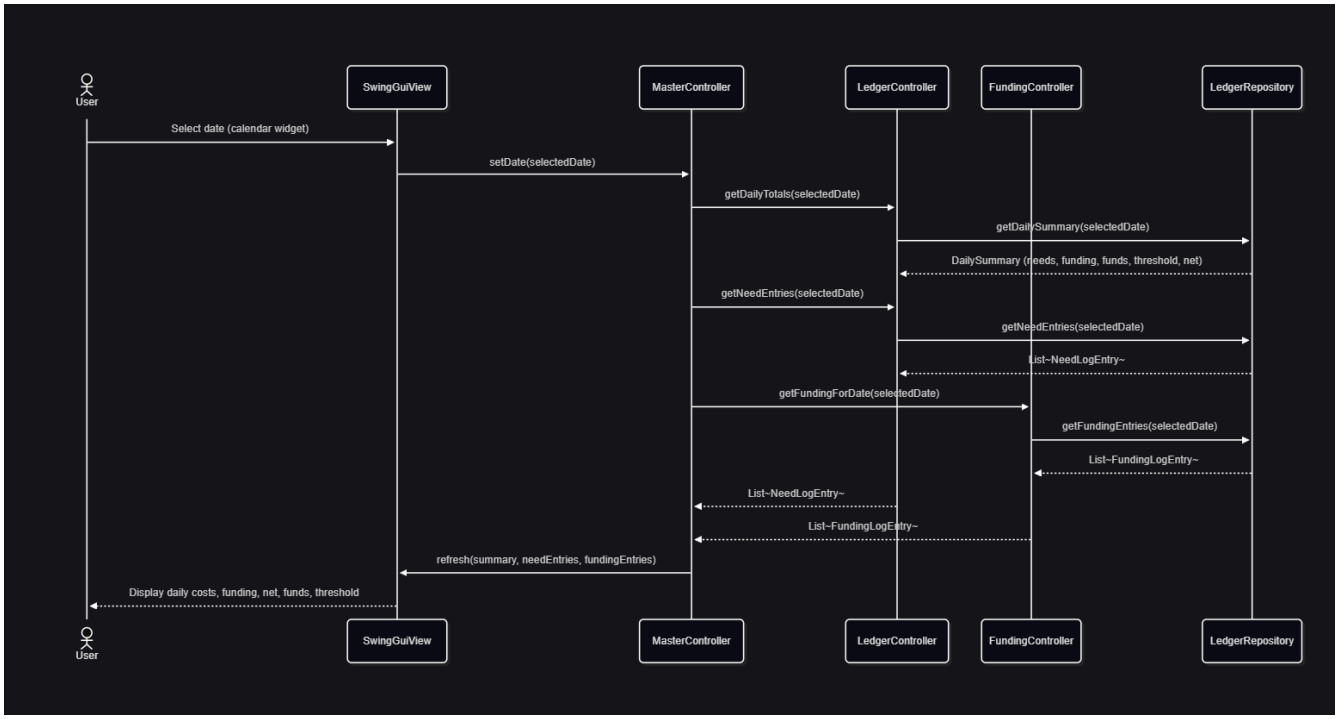
The **FundingController** encapsulates the domain logic related to funding operations and maintains a reference to the **LedgerRepository**. After performing the modification, it persists the updated income source and amounts to the repository. This ensures that application state remains consistent and that the **updated values are available** on subsequent application launches.

j. New Funding amount



When the user enters a funding amount, the UI layer; whether **CLI, GUI**, or the **MasterController** (represented collectively as **UI/Controller** in the diagram for brevity); invokes LedgerController.addEntry(date, need, amount). The **LedgerController** creates a new **LedgerEntity** with type = "FUND" and delegates persistence to the LedgerRepository, which writes the entry through the CSVManager. After the append operation completes, the controller notifies the view layer so the UI can display a confirmation message and refresh any computed totals.

k. Date Selector (core feature)



At any point, the **user can view information for the current date** (including income and expenses) and can n**avigate backward by one week or forward** without limit. In the illustrated workflow, the user **selects a date through the GUI**, which forwards the selected value to the **MasterController**. The **MasterController** delegates the request to the relevant controllers, invoking **getDailyTotals**, **getNeedEntries**, and **getFundingForDate** on the **LedgerController** and **FundingController**. These controllers, in turn, query the **LedgerRepository** to retrieve the underlying records, enabling the UI to render accurate, date-specific financial data.

Pattern Usage

l. Observer Pattern

Observer Pattern	
Observer(s)	ConsoleView(view layer), ReportView (optional)
Observable(s)	NeedRepository, LedgerRepository, FundingRepository,
Notification Method	notifyObservers(), update()
Event Sources	Need status changes, Funding recorded

m. Composite Pattern

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

n. Strategy Pattern

Strategy Pattern	
Receiver	Master Controller
Client (to create commands)	FundGoodDeedsApp
Invoker(s)	ConsoleUI, SwingUIView
Command	

o. MVC

MVC	
Model	Need, Bundle, LedgerEntity, FundingSource, LedgerRepository, NeedRepository, FundingRepository
View	ConsoleView, SwingUIView
Controller	NeedController, LedgerController, FundingController, MasterController
Glue/Contracts	Simple view models, using NeedComponent interface

p. Dependency Injection / Inversion (DI/IoC)

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

q. Repository Pattern

Repository Pattern	
Repository	NeedRepository, LedgerRepository, FundingRepository (in-memory R1; DB later)
Aggregate Root	NeedComponent, LedgerEntity, FundingSource
Clients	NeedController

System Rationale (V1)

The system is organized using the Model–View–Controller (MVC) architecture, the Composite pattern, the Observer pattern, the Repository pattern, and Dependency Injection, with clear separation of responsibilities among classes and concurrent updates across various interfaces.

Benefits, a huge separation of concerns. The Model layer includes classes such as NeedComponent, Need, Bundle, NeedsRepository, LedgerEntity, 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 the creation and validation of Need 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.

System Rationale (V2)

Version 2 replaces donation-based financial tracking with a more general funding income model. Instead of entering charitable donations, users now define funding sources (Paycheck, Hourly Job, Loan Disbursement, etc.) and log units of income received each day. This supports real-world budgeting scenarios and allows the system to calculate daily net cost (expenses minus income).

The addition integrates naturally into the Composite, MVC, Repository, and Observer patterns and is coordinated by the MasterController to ensure consistency across Need, Ledger, and Funding subsystems. Switching to funding sources made the whole system easier to reason about and matched what users actually do in real life when doing budgeting, banking or any financial tracking applications.

The authenticate user subsystem introduced in v2.1 provides true multi-user support without abandoning the CSV-based architecture. By isolating each users data into its own directories CSVManager.SetUserPath(), we maintain fully independent financial information. This is essential for privacy, modularity, and future enhancements. Such as admin roles or cloud synchronization. User Store centralizes password hashing, validation, and directory setup, keeping controllers and repos focused only on business logic.

Important Dates (V1 and V2):

10/13/2025 – Architectural Realignment to MVC
Shifted the project from a loosely structured controller-driven prototype into a strict **Model–View–Controller** architecture. Controllers now orchestrate workflows, views only handle user interaction, and models enforce business rules. This realignment enabled cleaner responsibility boundaries and made future enhancements predictable.

10/14/2025 – Observer-Based UI Synchronization
Introduced the **Observer Pattern** to keep UI components automatically synchronized with model changes. Need, Ledger, and Funding repositories now notify observers on updates, removing the need for manual refresh triggers and ensuring the UI always reflects accurate system state.

10/15/2025 – Repository Pattern for Persistent Data
Refactored persistence logic into standalone Repository classes for:

- NeedsRepository

- FundingSourceRepository
- LedgerRepository

CSV parsing and validation are now isolated and reusable, making controller logic simpler and storage-agnostic.

10/16/2025 – Dependency Injection & Composition Root

Implemented **Dependency Inversion** across the architecture.
All controllers now consume repository abstractions.
MasterController became the new **composition root**, wiring repositories, controllers, and the ConsoleView in a predictable, test-friendly structure.

10/17/2025 – Ledger Redesign for Full Financial Modeling

Replaced the previous donation-based model with a **fully financial daily ledger**, defining:

- fulfilled needs (expenses)
- funding source income
- daily funds
- daily thresholds
- net-cost computation

Added ledger entry types f, g, n, and i and centralized all daily calculations.

10/18/2025 – Funding Subsystem Introduction

Developed the **Funding Subsystem** with:

- FundingSource model
- FundingSourceRepository
- FundingSourceController

This replaced donation workflows entirely, introducing flexible income modeling (paychecks, hourly work, loans, etc.).

10/19/2025 – Composite Pattern Finalization for Bundles

Finalized the Composite structure (NeedComponent → Need / Bundle).
Improved recursive cost computation, bundle composition rules, and validation of forward references in needs.csv.
Guaranteed that bundles resolve deterministically and without circular definitions.

10/20/2025 – CSV Schema Standardization

Standardized Version 2 CSV formats:

- **needs.csv** – Need + Bundle definitions
- **funding.csv** – Funding source definitions

- **log.csv** – Daily funds, thresholds, needs, and income entries

Defined fixed column orders, row prefixes, and delimiter rules to stabilize parsing and future unit testing.

10/21/2025 – MasterController Design & Integration

Introduced the **MasterController** to consolidate subsystem orchestration:

- Handles selected date
- Enforces V2 date constraints (7-day rule, threshold restrictions)
- Coordinates loading/saving across all CSV files
- Provides one access point for UI actions
  This eliminated cross-controller coupling and made program execution deterministic.

10/22/2025 – Systemwide Error Handling Pass

Added robust validation for malformed CSV rows and inconsistent data states. Repositories now fail gracefully, skip corrupt entries, and notify the view with clear error messages. This increased stability during partial or messy data loads.

10/23/2025 – Controller Role Refinement

Clarified controller boundaries:

- Repositories = persistence only
- Controllers = rule enforcement and business logic
- Views = input/output only

Rewrote several controller methods to prevent business rules from leaking into repositories.

10/24/2025 – Financial Metrics & Daily Summary Hooks

Added helper methods for daily calculation pipelines, including:

- total cost
- total funding income
- net cost
- active funds
- active threshold

These support the ConsoleView's daily summary without duplicating math anywhere in the UI layer.

11/01/2025 – Unified Ledger Entry Model
Refined LedgerEntity to represent *any* row in log.csv and formalized EntryType enumeration. This eliminated prior redundancy between need- and funding-tracking logic.

11/03/2025 – V1 Diagram Lock & V2 Diagram Expansion
Completed cleanup of all V1 diagrams and produced V2-aligned diagrams with:

- MasterController
- Funding subsystem
- new CSV schemas
- updated composite interactions
- new sequence flow diagrams

These diagrams set the ground truth for the Version 2 design document.

11/05/2025 – Full V2 Integration & Console Stabilization
Connected the ConsoleView to all controllers via MasterController, enforcing proper observer updates and guaranteed consistency between need management, funding sources, and the daily ledger. This marked the first fully functional V2 skeleton.

11/06/2025 – Selected Date Enforcement & Ledger Rule Engine
Implemented core Version 2 calendar rules:

- No selecting dates older than 7 days
- Threshold modification locked if daily activity exists
- Past day financial fields cannot be altered
- Daily funds inherit the most recent prior value

Added a rule engine inside LedgerController to centralize validation logic.

11/07/2025 – Funding Source Income Pipeline
Finalized the income workflow with:

- User selection of funding source
- Unit multiplier conversion
- Ledger income entry creation
- Automatic totals recalculation
- Observer-driven UI refresh

This replaced the last remnants of donation-driven V1 behavior.

11/08/2025 – Daily Summary Aggregation Layer
Added a presentation-focused financial summary generator that consolidates:

- fulfilled need costs
- income totals
- net cost
- active threshold
- active funds

This provided a stable, reusable object for both UI and testing.

11/09/2025 – Composite Edge Case Corrections
Patched several corner cases in Bundle cost evaluation, including:

- recursive bundle-in-bundle lookups
- forward-reference validation
- empty bundle handling
- prevention of circular definitions

All invalid bundle definitions now trigger repository warnings instead of breaking load logic.

11/10/2025 – CSV Robustness & Sanitization Pass
Upgraded CSVManager to sanitize whitespace, trim invalid symbols, filter partial rows, and normalize malformed entries. Improved error messages for corrupted CSV states. Logged recoverable issues without stopping the system.

11/11/2025 – Controller-to-View Abstraction Cleanup
Refactored multiple controller return values into small view-model-like structures to simplify ConsoleView rendering. Reduced view logic duplication and removed any lingering business rule leakage from ConsoleView.

11/12/2025 – Repository Unit Test Scaffolding
Created mock-based unit test scaffolding for NeedsRepository, FundingSourceRepository, and LedgerRepository. Isolated CSV behavior for repeatable tests and formalized the expected behavior for each row type.

11/13/2025 – V2 Class Diagram Expansion
Expanded class diagrams to reflect:

- MasterController orchestration
- FundingSourceRepository
- FundingSourceController
- new LedgerEntity entry types
- refined Composite structure

Prepared these for insertion into the design document.

11/14/2025 – CLI Interaction Flow Finalization
Refined ConsoleView menus, adding consistent labeling, input validation, navigation shortcuts, and structured prompts. Ensured all funding, need, and threshold workflows follow a unified user experience.

11/15/2025 – UI Observer Behavior Polishing
Resolved race conditions and update timing issues between multiple observers. Ensured that Need, Ledger, and Funding updates propagate in the correct order when multiple subsystems are modified during a single user session.

11/16/2025 – User Subsystem Placeholder Stabilization
Integrated placeholder User and Admin classes with minimal fields to satisfy subsystem completeness. Ensured they do not interfere with V2 functionality but serve as a foundation for possible V3 authentication.

11/17/2025 – Documentation Refactor & Pattern Clarification
Rewrote internal and design documentation to clarify the use of MVC, Composite, Observer, Repository, and DI patterns. Updated terminology to remove all donation references and ensure complete alignment with funding-based modeling.

11/18/2025 – V2 Design Document Assembly
Compiled all diagrams, subsystem definitions, sequence flows, and updated schemas into the unified V2 design document. Ensured formatting uniformity, added MasterController integration details, and finalized all textual edits for submission-ready presentation.

11/19/2025 - Swing UI Layout
Cleaned up and drew out initial ui and aligned terms and console menus. Reduced clutter and introduced idea of panels.

11/20/25 - Master controller
Refactored console commands to route through a master controller instead of hitting controllers directly. This removed leftover V1 coupling and made command handling easy in cli and gui.

11/22/25 - Funding + ledger integration test
Reviewed how income and funding repo entries flow into the ledger repo, verified unites of each funding source are converted correctly into dollar amounts.

11/24/25 - observer notification ordering fix
Adjusted order of notify observers calls in the repositories to avoid bad ui changes when need ledger and funding updates happen. Verified both views always see snapshot upon save and reload.

11/29/25 - Swing ui wiring and look overhaul
Hooked up dates and reload button. Adding basing functionality to the view to coordinate with the working command line. Then took it to the next level with JOptionPanes, and JTables on panels to change what we are doing in the action area.

11/30/25 - Hook it all up and begin the demo and present.

Finally we came to heads and got it all logging and running now to develop presentation and demo and test for all bugs

12/2/25

Present and get chopped down by customers. We will be back, we will be better. Bugs were not all found. Diagrams need rework in some areas. Overall good work by team, and good presentation. Solid semester with a great group to work with.

12/3/25 - Implement users and fix bugs

We took on the task of creating user login, as well as fixing the bug within in needs and bundles that left needs inside of bundles after deletion. A mere mistake in the first version was an easy fix thanks to our great teammate Pat for heading this up as the other 3 members were all busy with other classes and he had free time.

12/3/25 - Final Design doc doctoring

Final tweaks and final additions of diagrams to this design doc are done, we now show the user flow, as well as updating any and all diagrams that were out of date upon presentation. Thanks for the opportunity to do this a day after presenting we were able to make our whole project better and more solid.