# Personal Finance Manager

Project by thumma jai sri santhosh kumar

· 23 November 2025

· Income, expense, savings, and accident-dues management

**THUMMA JAI SRI SANTHOSH KUMAR 25BAI10045**

# Personal Finance Manager — Desktop App

Tkinter-based UI to track income, expenses, savings and pay accident dues

1. User-friendly Tkinter desktop interface for personal finance
2. Track income, expenses, and savings in one place
3. Automatic savings calculation from transactions
4. Manage and pay accident dues impacting balances
5. Quick operations: add transactions, calculate savings, pay debts

# Clear, lightweight personal finance tracking

Automatic savings, current balances, and immediate 'accident' dues—simple interface, minimal complexity

**User need:** lightweight, easy-to-use tool to record personal finances

1

**Automatic savings:** calculate based on income or balances

2

**Accidents:** unexpected expenses that create immediate dues

3

**Tracking gaps:** need clear view of current balances, savings, and accident debts

4

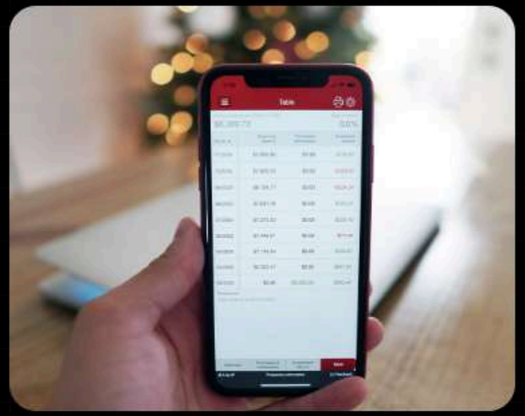
**Value:** simple interface to maintain clearer financial overview with minimal complexity

5

# Key Functional Requirements

Clear, user-focused behaviors for balance, savings, accident dues, and visibility

1



## Income & Balance Management

- Add income to **increment balance** immediately
- Record expenses to **decrease balance**
- UI shows **current balance** prominently

2

## Monthly Savings Configuration

- Set **monthly savings** as a percentage of current balance
- Auto-calculate savings from balance each month
- Display **total savings** separately in UI

3

## Accident Expense Simulation

- Generate random accident expenses bounded by **current balance**
- Track these as separate **accident dues**
- UI highlights unpaid accident dues clearly

4

## Accident Dues Payment Flow

- Pay accident dues using **savings first**
- If savings insufficient, deduct remaining from **balance**
- Update balance and savings immediately after payment

# Non-functional Requirements for Windows Tkinter App

Platform, usability, performance, reliability, and maintainability priorities

**Platform:** Target Windows desktop using **Tkinter** for compatibility and fast development

**Usability:** Minimalistic UI to enhance clarity and avoid overwhelming users

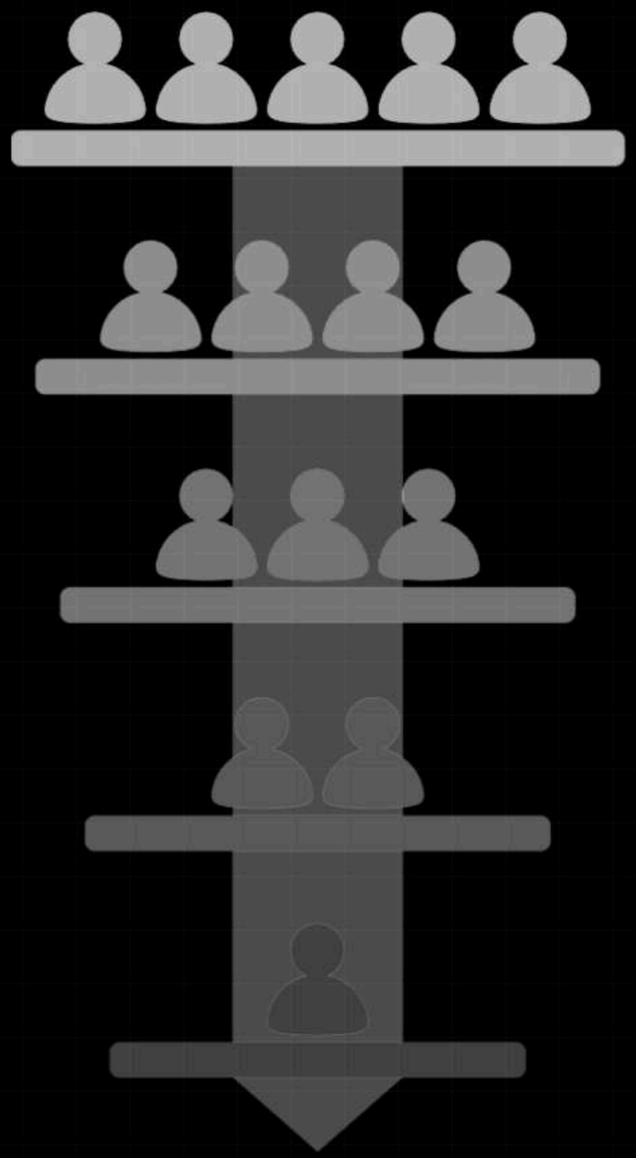
**Performance:** Immediate UI responsiveness during user actions

**Reliability:** Basic integer input validation to prevent errors

**Maintainability:** Currently a single Python prototype file; recommend modularization for easier enhancements and debugging

# System Architecture Overview

Single-process desktop app broken into clear UI, Business Logic, and Persistence layers for scalable maintenance



1

### UI (Tkinter)

Tkinter widgets: Entry, Button, Label — handles user interaction and display.

2

### Business Logic

Functions managing balance, savings, accident dues; Controller handles events like add_income, add_expense.

3

### Persistence

Current in-memory storage with planned optional JSON or SQLite persistence.

4

### Controller

Event handler connecting View actions to Model updates.

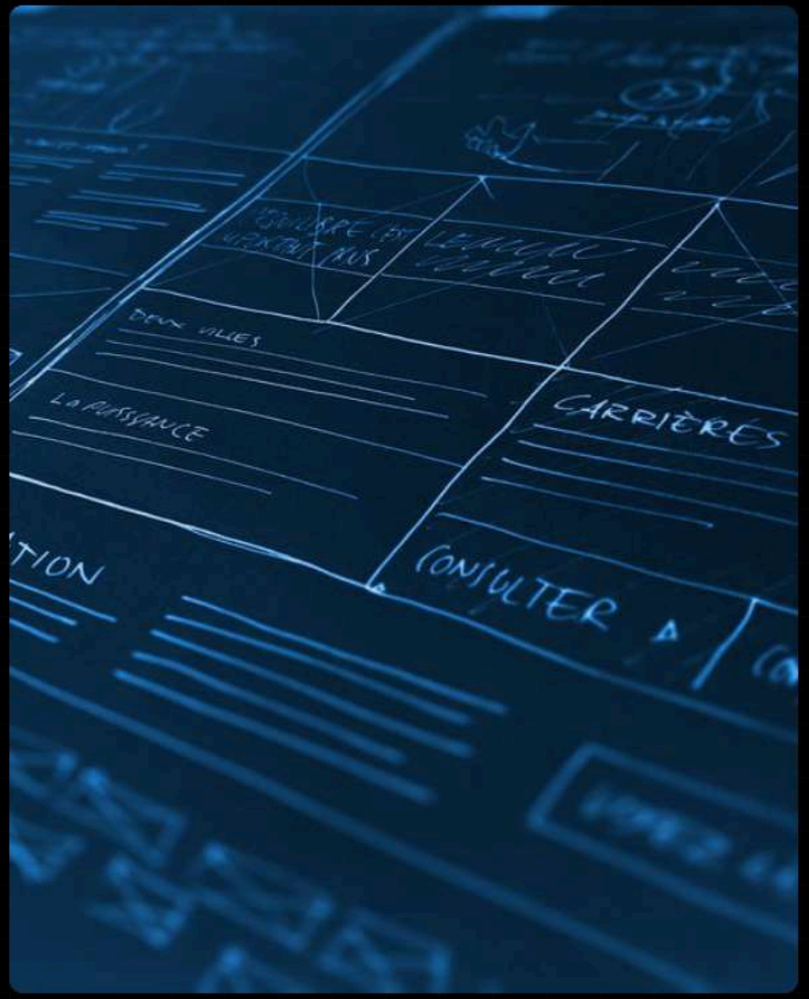
5

### Model

Global variables: bal, saving, accident represent application state.

# Design Diagrams Overview

Concise summary of actors, flows, and components for each UML diagram




## Use Case Diagram

Actors: User

Use cases: add income/expense, calculate savings, trigger accidents, pay dues

Focus: user actions and system responsibilities

## Workflow Diagram

Flow: sequential user inputs → app responses from start to finish

Shows: decision points, input validation, end states

Value: maps end-to-end user journey

## Sequence & Class/Component

Sequence: runtime interactions among User, GUI, Controller, Model

Class/Component: GUI invokes controller functions; controller updates model; GUI reads model

ER suggestions: recommended for future persistent storage

# Design Decisions and Rationale

Key choices, why they were made, and their implications

**Tkinter** chosen for rapid desktop prototyping — standard library, zero external dependencies, accelerates development cycles

**Single-file** prototype for simplicity — faster iteration but may create future maintainability challenges

**In-memory state** management simplifies logic; trade-off: data loss on application exit

**Random accident costs** bounded by current balance to ensure realistic, meaningful financial stress tests

Implication: fast iteration and realistic simulation vs. future refactor needs and persistence gaps

# Implementation Details — Core Mechanics & UI

Clear mapping of variables, functions, UI elements, and improvement opportunities

## Core variables

- **bal** — current balance (global)
- **saving** — accumulated savings (global)
- **accident** — accident dues (global)

## Primary functions

- **add_income()** — increase balance from input
- **add_expense()** — decrease balance
- **calculate_savings()** — move configured percentage to savings
- **random_event()** — generate accident expense up to current balance

## Accident payment logic

**pay_accident_dues()** — pay from savings first, then balance

## UI elements

- Labels and Entry widgets for inputs
- Buttons to trigger add_income, add_expense, calculate_savings
- Direct interaction via widgets

## Suggested improvements

- Input validation for user entries
- Persistency (save/load state)
- Modularization to reduce globals

# UI Evidence: Workflow Snapshots

Sequential screenshots validating app states during a typical user session

Main window at startup with **empty balance** to show initial state

After **income addition** showing updated balance and transaction entry

After **expense addition** showing reduced balance and expense record

Post **savings calculation** reflecting allocated savings and remaining funds

After **accident trigger** and **dues payment** illustrating liability handling

Purpose: validate **UI status transitions** and functional workflows

# Testing Approach: Manual + Automated

Validate finance flows, simulate edge cases, and implement pytest unit coverage

Manual validation of **income addition** and **balance updates**

Test handling of **expenses > balance**; design decision pending

Verify **savings calculation** at various percentages

Simulate multiple **accident events** to verify dues accumulation

Test paying dues when **savings insufficient** to confirm balance deduction

Automated unit tests: modularize into **FinanceState dataclass** with pytest per functionality

Edge cases: **non-integer inputs, negative values, zero balance, repeated accidents**

# Key Technical Challenges

Practical obstacles that limit robustness, usability, and maintainability

**Global state management** across multiple UI handlers causing inconsistent state and harder debugging

**No persistence**, resulting in data loss when the app closes and reduced usability

**Missing input validation** makes the prototype vulnerable to crashes with non-numeric entries

**UI scaling & layout limits** in Tkinter complicate creating intuitive, attractive interfaces

Awareness of these issues guides future work on **robustness** and **maintainability**

# Practical Takeaways for Developers

How to build small, robust personal-finance tools fast

Rapid prototyping with **Tkinter** is efficient for small-scale personal finance tools

Separate **UI** and **business logic** to improve testability and maintainability

Persist state using **JSON** or **SQLite** to enhance usability

Implement defensive **input validation** to prevent crashes and improve robustness

Balance development **simplicity** with foundational software engineering principles

Summary: fast prototypes + modular design + persistent state + validation = usable, maintainable tools

# Future Enhancements to Increase Robustness & Scalability

Planned features that improve reliability, usability, and global reach

- Add **persistent storage** (SQLite or JSON) to prevent data loss
- Improve **input validation** and comprehensive **error handling** for robustness
- Provide **transaction history** view with **CSV export** for record-keeping
- Add **monthly reports & charts** using Matplotlib for better analysis
- Support **internationalization** and **multi-currency** for broader applicability
- Refactor to **MVC architecture** or migrate to a **web app framework** to scale

# Key References for Development and Documentation

Trusted resources for Tkinter, PlantUML, and SQLite

1

**Python 3 Tkinter — Official GUI library reference: https://docs.python.org/3/library/tkinter.html**

2

**PlantUML — Diagram creation tool and syntax guide: https://plantuml.com/**

3

**SQLite — Lightweight embedded database for persistence: https://www.sqlite.org/index.html**

4

**These resources support development, documentation, and future enhancements**