

VIET NAM NATIONAL UNIVERSITY HO CHI MINH
UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

Final Project

Personal Finance Manager

Course: Introduction to Computer Science - CS160

Student list:

Nguyễn Đình Minh Huy (25125083)

Lê Trần Hoàng Long (25125087)

Trần Đức Toàn (25125091)

Group 09 - Class 25A01

29th December 2025



Contents

1 Team Composition & Task Assignment	1
1.1 Group Information	1
1.2 Work Division	1
1.3 Collaboration Workflow	2
2 Project Overview	3
2.1 Introduction	3
2.2 Key Technical Highlights	3
2.3 Objectives	3
2.4 Scope	4
3 Requirements Specification	4
3.1 Functional Requirements	5
3.1.1 Transaction Management	5
3.1.2 Automation (Recurring Transactions)	5
3.1.3 Search & Reporting	5
3.2 Non-Functional Requirements	6
3.2.1 Technical Constraints (No STL)	6
3.2.2 Data Persistence & Reliability	6
3.2.3 Cross-Platform Compatibility	6
4 High-Level Architecture	6
5 Technical Core	7
5.1 Core Data Structures	7
5.2 Algorithmic Components	8
5.2.1 Sorted Insertion by Date	8
5.2.2 Indexed Lookup (Secondary Indices)	8
5.2.3 Range Query on Sorted Data	9
5.2.4 Recurring Transaction Scheduling	9
5.2.5 Search and Filtering Engine	10

5.3	Concurrency and Data Safety	10
6	System Implementation Details	11
6.1	Custom Data Structures (No STL)	11
6.1.1	Dynamic Array (<code>ArrayList<T></code>)	11
6.1.2	Hash Map (<code>HashMap<K, V></code>)	11
6.2	Utilities: ID Generation	12
6.2.1	Implementation Strategy	12
6.2.2	Role in Data Modeling	12
6.3	Data Modeling & Entity Relationships	13
6.3.1	Core Transactions	14
6.3.2	Automation Model (Recurring Transactions)	14
6.3.3	Master Data Entities	14
6.4	Binary Persistence Engine	15
6.4.1	Generic Serialization Templates	15
6.4.2	Persistence Strategy: Snapshots vs. In-Place Editing	15
6.4.3	Integration with Auto-Save	15
6.5	Cross-Platform Infrastructure	16
6.5.1	The Problem: "Speaking Different Languages"	16
6.5.2	One-Click Build Scripts	16
6.6	UI Techniques	16
6.7	Short walkthrough — each feature	17
6.7.1	Dashboard	17
6.7.2	Main Menu	18
6.7.3	Income (Add / View / Edit / Delete + Manage Sources)	19
6.7.4	Expense (Add / View / Edit / Delete + Manage Categories)	19
6.7.5	Wallets (Create / View / Delete)	19
6.7.6	Reports and Analytics	20
6.7.7	Recurring Transactions	20
6.7.8	Search / Clear / Save Data	20
6.8	Files evidence	20

6.9	Integration with the Technical Core	21
6.9.1	Sorted Transaction Management	21
6.9.2	Fast Queries via Secondary Indices	21
6.9.3	Date Range Queries and Reporting	22
6.9.4	Recurring Transaction Generation	22
6.9.5	Search and Filtering Operations	22
6.9.6	Concurrency Guarantees	22
7	Quality Assurance and Systematic Testing	23
7.1	Overview and Testing Philosophy	23
7.2	Core Testing Strategies	23
7.2.1	Boundary Value Analysis (BVA)	23
7.2.2	Stress and Scalability Testing	23
7.2.3	State Transition Testing	24
7.2.4	Concurrency and Synchronization Testing	24
7.2.5	Manual Memory Profiling	24
8	Detailed Functional and Logical Test Cases	25
8.1	Financial State and Boundary Testing	25
8.1.1	Negative Value Injection	25
8.1.2	Zero-Value Transaction	25
8.1.3	Leap-Year Recurring Transaction	25
8.2	Referential Integrity and Data Safety	25
8.2.1	Protection of Linked Master Data	25
8.2.2	Identifier Collision Resilience	26
9	UI Robustness, Performance, and Reliability Testing	26
9.1	User Interface Stability	26
9.1.1	Rapid Navigation under Concurrent Persistence	26
9.1.2	Special Character and Long String Input	26
9.2	Performance Evaluation	26
9.2.1	Large Dataset Retrieval	26

9.3 Reliability and Crash Recovery	27
9.3.1 Abrupt Process Termination	27
9.3.2 Full Lifecycle Memory Cleanup	27
10 Conclusion	27
11 Future Work	27
12 Conclusion	28
References	29

1 Team Composition & Task Assignment

This project was developed by **Group 09** under the leadership of Nguyễn Đình Minh Huy. The workload was distributed based on the Model-View-Controller (MVC) architecture, ensuring a clear separation of concerns and parallel development.

1.1 Group Information

Group ID: 09
Class: 25A01
Leader: Nguyễn Đình Minh Huy (25125083)

1.2 Work Division

The table below details the specific responsibilities and code contributions of each member.

ID	Member	Role & Responsibilities	Key Files
25125083	Minh Huy	Role: Core & Data Engineer <ul style="list-style-type: none"> • Core: Designed custom <code>ArrayList</code> & <code>HashMap</code> (No STL). • Models: Implemented all Entity classes. • Infra: Built Thread-safe Auto-Save & Serialization. • System: Cross-Platform compatibility. 	<code>Utils/*</code> <code>Models/*</code> <code>NavSearch</code>
25125087	Hoàng Long	Role: Logic Developer <ul style="list-style-type: none"> • Controller: Implemented <code>AppController</code> business logic. • Algorithms: Developed filtering & sorting. • Reporting: Built aggregation logic. 	<code>AppController</code> <code>UnitTest.cpp</code>
25125091	Đức Toàn	Role: UI/UX Developer <ul style="list-style-type: none"> • Views: Designed <code>ConsoleView</code> & Tables. • Validation: Implemented <code>InputValidator</code>. • Layout: Managed Dashboard & Menus. 	<code>NavigationController</code> <code>Views/*</code> <code>main.cpp</code>

Table 1: Detailed breakdown of team responsibilities.

1.3 Collaboration Workflow

The team utilized modern collaboration tools to maintain code quality:

- **Version Control:** GitHub was mandatory for all code submissions.
Github link: <https://github.com/ndmhuy/PersonalFinanceManager>
- **Communication:** Daily stand-up meetings via Messenger for bug tracking.
- **Code Review:** Core logic and controller changes were reviewed by at least one other member before merging.

2 Project Overview

2.1 Introduction

The **Personal Finance Manager** is a cross-platform console application developed in C++ to help users track their financial health. It enables the efficient management of multi-wallet balances, income and expense records, and automated recurring transactions through a command-line interface.

2.2 Key Technical Highlights

This project was engineered under a strict academic constraint: **No Standard Template Library (STL)** was permitted. To achieve this, we implemented:

- **Custom Data Structures:** Generic `ArrayList` and `HashMap` containers built from scratch using manual memory management (pointers).
- **MVC Architecture:** A strict separation of Data (Models), Business Logic (AppController), and User Interface (ConsoleView) to ensure scalability.
- **Persistence & Auto-Save:** A custom binary serialization engine coupled with a **multi-threaded background worker** that automatically persists data changes to disk, ensuring zero data loss.
- **Cross-Platform Support:** Universal build scripts (`.bat` and `.command`) ensuring seamless execution on both Windows and macOS.

2.3 Objectives

The main objectives of the Personal Finance Manager project are:

- Provide a reliable and easy-to-use command-line interface for tracking personal finances (wallets, incomes, expenses, recurring transactions).
- Ensure data integrity and persistence through robust serialization and safe update/delete semantics (restrict-delete checks and immediate save for critical operations).

- Keep business logic centralized and testable (single ‘AppController’ entry point) and provide small, single-responsibility UI flows for easy validation.
- Support cross-platform execution and reproducible builds so evaluators can run the application on Windows and macOS with minimal setup.
- Produce clear documentation, diagrams and screenshot evidence so graders can verify implemented features quickly.

2.4 Scope

This project is scoped as a single-user, desktop console application. The scope includes:

- Core features: create/view/edit/delete transactions (income/expense), manage wallets, categories and income sources, recurring transaction automation, and basic reporting (monthly summaries, category breakdowns).
- Presentation: Console UI components (menus, dashboard, tables) and input validation helpers.
- Persistence: Binary serialization for saving/loading state and utilities for file IO.
- Documentation: A report with diagrams and screenshots; code comments and README for reproducibility.

Out of scope

- Multi-user or networked synchronization, central server components, or database-backed services.
- Advanced analytics (forecasting, machine learning) or third-party payment integrations.
- A graphical desktop/mobile UI — the project intentionally targets the console environment.

3 Requirements Specification

The system was designed to strictly adhere to the functional specifications outlined in the Final Project Coursework, while satisfying rigorous technical constraints regarding memory management and platform compatibility.

3.1 Functional Requirements

3.1.1 Transaction Management

The core function of the application is to record and manage financial activities with precision.

- **Income Recording:** Users can log earnings by specifying the Date, Amount, Wallet, and an **Income Source** (e.g., Salary, Bonus).
- **Expense Recording:** Users can log expenditures by linking them to a **Category** (e.g., Food, Rent) and a specific Wallet.
- **Validation:** The system must prevent invalid entries, such as negative amounts or linking transactions to non-existent wallets.

3.1.2 Automation (Recurring Transactions)

To reduce manual data entry, the system supports an automation engine for regular payments.

- **Scheduling:** Users can define templates for recurring transactions (Daily, Weekly, Monthly).
- **Auto-Generation:** Upon application startup, the **AppController** detects due transactions based on the last generation date and automatically creates the necessary records for the missed period.

3.1.3 Search & Reporting

The application provides tools to analyze financial health:

- **Date Range Filtering:** View detailed transaction history within a specific start and end date.
- **Advanced Search:** Filter transactions by Wallet, Category, or Keyword search (in descriptions).
- **Financial Summary:** Aggregated views of Total Income, Total Expense, and Net Balance per Wallet.

3.2 Non-Functional Requirements

3.2.1 Technical Constraints (No STL)

As a primary academic challenge, the use of the **Standard Template Library (STL)** was strictly prohibited for data containers.

- **Implementation:** We replaced `std::vector` and `std::map` with custom-built `ArrayList<T>` and `HashMap<K, V>` classes using manual dynamic memory allocation.

3.2.2 Data Persistence & Reliability

- **Binary Storage:** All data (Transactions, Wallets, Master Data) is serialized to binary files to ensure efficient storage and fast loading times.
- **Auto-Save (Enhancement):** Beyond standard session saving, we implemented a **background thread** that triggers a save operation every 60 seconds. This ensures data integrity even in the event of an unexpected application crash.

3.2.3 Cross-Platform Compatibility

The application is designed to function identically on different operating systems:

- **Windows:** Compiled via MinGW or Visual Studio ('run_windows.bat').
- **macOS/Linux:** Compiled via Clang/GCC ('run_mac.command').

4 High-Level Architecture

The system follows a simple layered architecture that keeps presentation, control and business logic separated for clarity and testability:

- **Models** — domain objects such as 'Transaction', 'Wallet', 'Category', 'IncomeSource', and 'RecurringTransaction' live in 'include/Models' and 'src/Models'.
- **Controllers** — 'AppController' implements the business operations (add/edit/delete transactions, wallets, generate reports). 'NavController' (and the 'Nav*' files) implement UI flows and call 'AppController' methods.

- **Views** — the console presentation layer is in ‘include/Views‘ / ‘src/Views‘ and includes ‘ConsoleView‘, ‘Menus‘, ‘DashBoard‘ and ‘InputValidator‘ which format output and validate user input.
- **Persistence and Utilities** — data is serialized using ‘BinaryFileHelper‘ and small helpers like ‘IdGenerator‘, ‘Date‘, and ‘HashMap‘ support in-memory operations and file storage.

This separation keeps the UI lightweight and makes core logic easier to unit test without requiring console interaction.

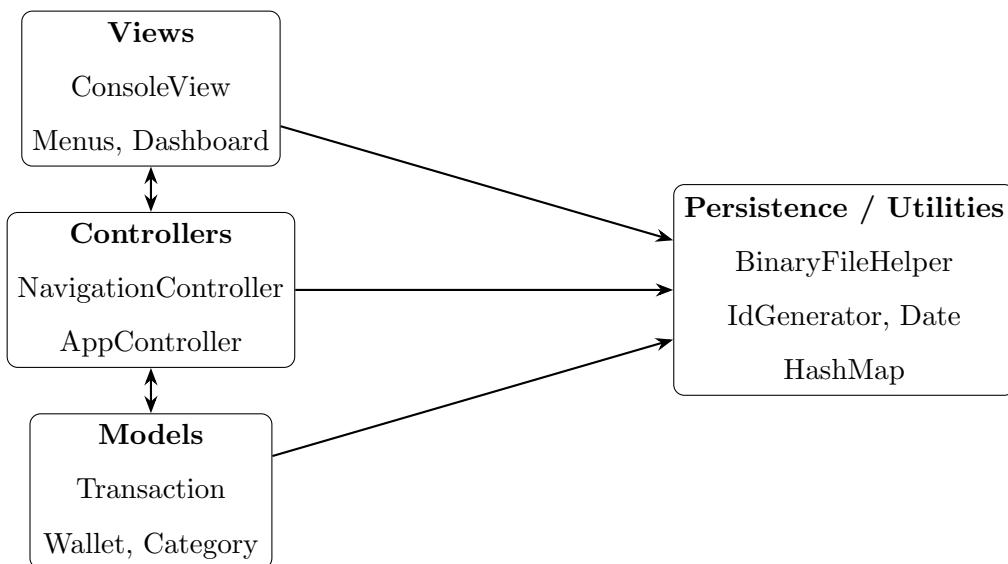


Figure: High-level layered architecture (Views → Controllers → Models → Persistence)

5 Technical Core

This section describes the core technical mechanisms and algorithms that form the foundation of the system. These components are independent from user interface or controller logic and focus purely on data integrity, performance, and correctness.

5.1 Core Data Structures

The system is built upon custom generic data structures to maintain full control over memory usage and behavior.

- **ArrayList<T>** — dynamic array used for ordered collections.

- **HashMap<K, V>** — hash-based associative container for fast key-based access.

Each structure is designed to be deterministic, serializable, and independent of STL containers to simplify debugging and binary persistence.

5.2 Algorithmic Components

5.2.1 Sorted Insertion by Date

All transactions are maintained in chronological order based on their date.

- **Algorithm:** Binary search to determine insertion position, followed by indexed insertion.
- **Purpose:** Preserves global ordering without requiring full re-sorting after each insertion.

Procedure:

1. Perform binary search on the sorted list to find the first position where `existingDate` \geq `newDate`.
2. Insert the new element at the computed index.

Time Complexity:

- Binary search: $O(\log n)$
- Array insertion shift: $O(n)$
- Overall: $O(n)$

Invariant: The transaction list is always sorted in non-decreasing order of date.

5.2.2 Indexed Lookup (Secondary Indices)

To support fast filtering without repeated full scans, secondary index maps are maintained.

- Wallet-based index
- Category-based index
- Income-source-based index

Each index maps an entity ID to a list of related transactions.

Algorithm: Hash-based indexing with append-only updates.

Operations:

- Insert transaction reference into one or more index lists.
- Remove reference upon deletion or modification.

Time Complexity:

- Insert / Remove: $O(1)$ average
- Query by indexed key: $O(k)$ where k is number of related transactions

Design Rationale: This approach trades small memory overhead for significant query performance gains compared to repeated linear scans.

5.2.3 Range Query on Sorted Data

Date-range queries leverage the sorted nature of the transaction list.

Algorithm: Binary search followed by linear scan.

1. Binary search to locate the first transaction with date \geq start date.
2. Sequentially iterate until date $>$ end date.

Time Complexity:

- Binary search: $O(\log n)$
- Range scan: $O(m)$ where m is number of results

Invariant: Transactions are strictly ordered by date, enabling early termination.

5.2.4 Recurring Transaction Scheduling

Recurring entries act as templates for generating future transactions.

Algorithm: Incremental date progression.

- Each recurring record stores its last generated date.

- The next due date is computed deterministically from frequency rules.

Procedure:

1. Compare next due date with current date.
2. If due, generate a concrete transaction and advance the schedule.
3. Repeat until the template is no longer due.

Time Complexity:

- Per recurring record: $O(g)$ where g is number of generated instances

Invariant: Each recurring transaction is generated exactly once per valid period.

5.2.5 Search and Filtering Engine

Search and filter operations are implemented using predicate-based linear scans.

Algorithm: Linear filtering with lambda predicates.

Use Cases:

- Search by keyword
- Filter by type
- Filter by amount range

Time Complexity: $O(n)$

Rationale: These operations are user-driven and operate on manageable data sizes, making linear scans acceptable while keeping the core logic simple and predictable.

5.3 Concurrency and Data Safety

The technical core enforces thread safety through mutual exclusion.

- **Mechanism:** Recursive mutex locking
- **Scope:** All read/write operations on shared data structures

Invariant: At most one thread may mutate core data structures at any given time.

Benefit: Prevents race conditions while allowing re-entrant calls within the same execution flow.

6 System Implementation Details

This section details the implementation of the core data structures, data modeling, application logic, utilities, and the UI handling.

6.1 Custom Data Structures (No STL)

We engineered two core generic container classes. These structures manage their own memory manually, replacing ‘`std::vector`’ and ‘`std::unordered_map`’.

6.1.1 Dynamic Array (`ArrayList<T>`)

We implemented a template-based dynamic array to serve as the primary linear container for the application.

- **Memory Strategy:** The class maintains a raw pointer `T* data` and a `capacity` counter.
- **Geometric Expansion:** When the array becomes full, the system allocates a new block of memory sized $2 \times \text{current_capacity}$. It then copies existing elements to the new block and frees the old memory. This ensures the nature of an array.
- **Safety:** The class implements a RAII-compliant destructor that automatically invokes `delete []` on the internal array, ensuring no memory leaks occur when the controller goes out of scope.

6.1.2 Hash Map (`HashMap<K, V>`)

To support "Fast-indexing" (e.g., retrieving a Wallet by its ID) and high-performance "Secondary Indexing" (e.g., filtering transactions by Wallet ID), we implemented a custom Hash Table.

- **Hashing Strategy:** We implemented the **djb2 algorithm** [1] (created by Dan Bernstein) to map string keys to integer indices. This function initializes a hash value to 5381 and iterates through the string, updating the value using bitwise operations ('`hash * 33 + char`'). This approach is computationally efficient and generates a high-entropy distribution for ASCII strings (like our UUIDs).

- **Collision Resolution:** We utilized the **Separate Chaining** technique. Each "bucket" in the hash table points to a linked list of nodes. If two distinct keys hash to the same index, the new entry is safely appended to the chain.
- **Performance:** This reduces lookup complexity from $O(N)$ (linear scan) to $O(1)$ (average case), which is essential for keeping the application responsive as the dataset grows.

6.2 Utilities: ID Generation

To ensure referential integrity across the system without relying on fragile memory pointers, every entity (Transaction, Wallet, Category, etc) is assigned a unique string identifier.

6.2.1 Implementation Strategy

We implemented a custom `IdGenerator` class to handle unique key creation.

- **Algorithm:** The generator utilizes the **Mersenne Twister 19937** engine (`std::mt19937`), seeded by a non-deterministic random device (`std::random_device`). This ensures a uniform distribution and minimizes the risk of ID collisions.
- **Format:** IDs are generated in a human-readable format: `PREFIX-XXXX-XXXX-XXXX`.
- **Composition:** The random blocks consist of alphanumeric characters (A-Z, a-z, 0-9), selected from a charset of size 62.

6.2.2 Role in Data Modeling

As illustrated in Figure 1, these IDs serve as the backbone of our relational model.

- **Primary Keys:** Every `transactions` and `wallets` record holds a unique `id` field.
- **Foreign Keys:** Relationships are maintained by storing string references (e.g., `walletId`) rather than memory pointers. This allows the system to serialize data to binary files safely without complex pointer swizzling.

6.3 Data Modeling & Entity Relationships

The system utilizes a relational design pattern adapted for object-oriented programming. As shown in Figure 1, the data model relies on unique string identifiers (UUIDs) to maintain relationships between entities, avoiding the complexity and risks of raw memory pointers.

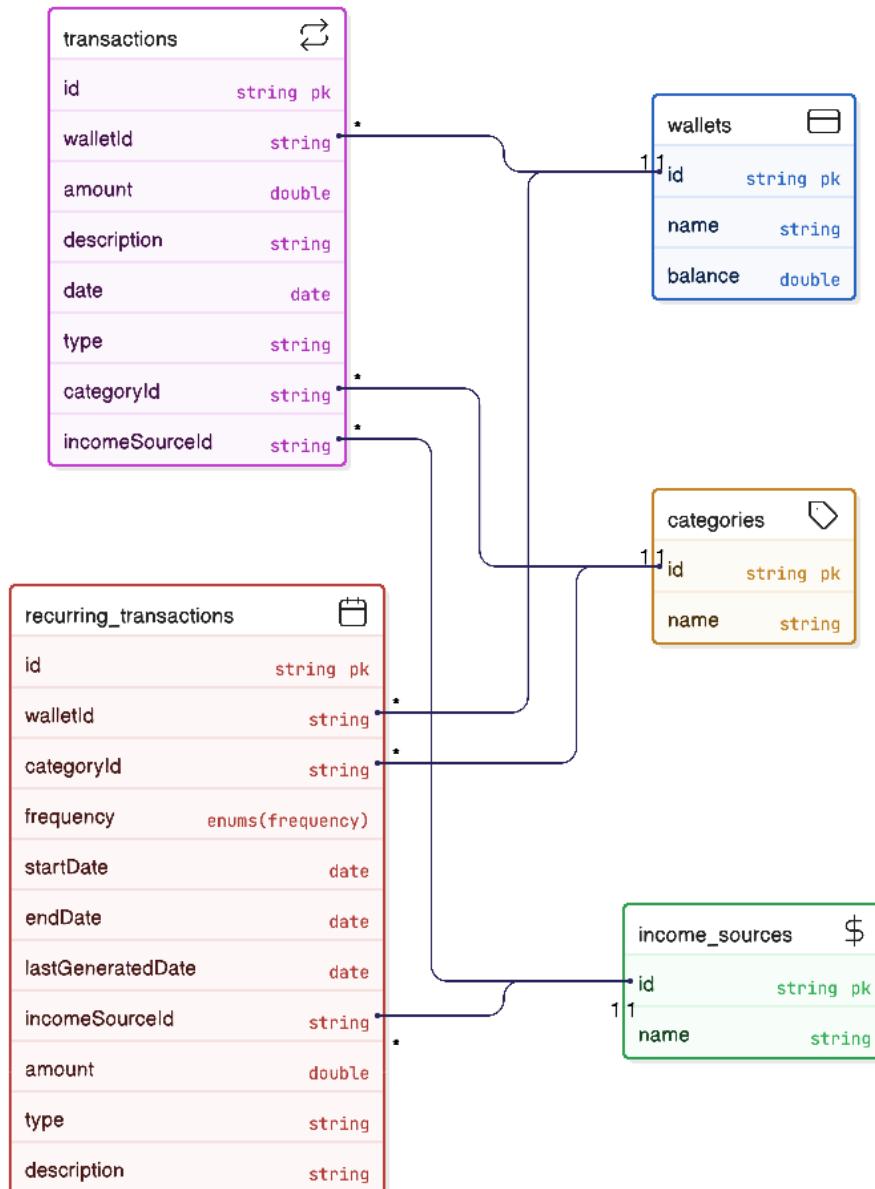


Figure 1: Entity Relationship Diagram

6.3.1 Core Transactions

The `transactions` entity represents the central ledger of the application. It utilizes a polymorphic design to handle both inflows and outflows.

- **Common Attributes:** All records share standard financial data:
 - `id`: The unique primary key.
 - `walletId`: A foreign key linking the transaction to a specific `Wallet`.
 - `amount`: The monetary value.
 - `date`: The timestamp of the transaction.
 - `type`: A discriminator string ("Income" or "Expense").
- **Classification References:** Depending on the transaction type, the record links to a specific classification entity:
 - **Expenses:** Utilize the `categoryId` foreign key to link to the `categories` table.
 - **Incomes:** Utilize the `incomeSourceId` foreign key to link to the `income_sources` table.

6.3.2 Automation Model (Recurring Transactions)

The `recurring_transactions` entity serves as a "Blueprint" for future data generation.

- **Scheduling Logic:** Unlike standard transactions, this entity includes frequency data (`frequency`, `startDate`, `endDate`) and tracking state (`lastGeneratedDate`).
- **Independence:** These records do not affect wallet balances directly. Instead, the system polls these blueprints upon startup to generate concrete `transactions` when due dates are reached.

6.3.3 Master Data Entities

The system maintains three lightweight reference entities to enforce data validation.

- **Wallets:** Stores the current `balance` and account `name`.
- **Categories:** Defines expense classifications (e.g., "Food", "Rent").
- **Income Sources:** Defines income origins (e.g., "Salary", "Dividends").

6.4 Binary Persistence Engine

To meet the requirement of preserving data across sessions, we implemented a custom serialization engine (`BinaryFileHelper`). This module handles the low-level translation of C++ objects into a binary stream.

6.4.1 Generic Serialization Templates

To avoid repetitive code, we implemented generic wrapper functions for reading and writing primitive types.

- **Template Approach:** We utilized `template<typename T>` functions to handle standard data types (`int`, `double`, `bool`). These functions cast the memory address of variables to `char*` and write them directly to the `std::ofstream` buffer.
- **String Handling:** Variable-length strings (such as descriptions) pose a challenge in binary files. We solved this by first writing the length of the string (`size_t`) as a header, followed immediately by the character data. This allows the reader to know exactly how many bytes to extract during deserialization.

6.4.2 Persistence Strategy: Snapshots vs. In-Place Editing

A significant engineering decision involved how to handle updates (e.g., editing a transaction).

- **The Problem:** Implementing "In-Place" editing in a binary file is highly complex. Changing a description from "Food" (4 bytes) to "Groceries" (9 bytes) would require shifting all subsequent bytes in the file, which is error-prone and slow.
- **The Solution (Snapshotting):** Instead of modifying specific bytes, we adopted a "**Full Rewrite**" strategy. Whenever data is saved, the system rewrites the entire `transactions.bin` file from scratch using the current memory state.

6.4.3 Integration with Auto-Save

Because rewriting the entire file can be computationally expensive (relative to a single byte update), we offloaded this task to the background thread mentioned in the **Technical Core** section.

- **Frequency:** The `AutoSaveWorker` triggers this "Snapshot" process every 60 seconds.
- **Benefit:** This ensures that the expensive I/O operation never blocks the user interface (Main Thread), while still guaranteeing that the disk storage is never more than one minute behind the live data.

6.5 Cross-Platform Infrastructure

Making a console application run smoothly on both Windows and macOS is surprisingly difficult. While the logical C++ code is the same, the "Terminal" itself behaves very differently on each operating system. We built a translation layer to bridge this gap.

6.5.1 The Problem: "Speaking Different Languages"

Simple commands like "Clear Screen" or "Wait for Key Press" are not standard in C++.

- **Windows** uses specific system commands (like `cls`).
- **macOS/Linux** relies on special text codes (ANSI escape sequences).

Our Solution: We created a `PlatformUtils` file that acts as a translator. It checks which OS is running and automatically chooses the correct command, so the rest of our app doesn't have to worry about it.

6.5.2 One-Click Build Scripts

We didn't want users to memorize complex compilation commands. We wrote simple scripts for each OS:

- **Windows:** `run_windows.bat` (Detects compiler and builds).
- **macOS:** `run_mac.command` (Sets permissions and runs).

6.6 UI Techniques

We used several techniques to make the console UI usable and easy to test:

- **Layered rendering API** — ‘ConsoleView‘ provides primitives (headers, boxed panels, tables, colored lines) so high-level screens can be composed consistently across features.
- **Consistent menus and shortcuts** — ‘Menus‘ centralizes text and shortcuts so each flow uses the same input conventions (number keys to select, ESC to cancel/back).
- **Explicit input validation** — ‘InputValidator‘ checks money, dates, indexes and strings before handing control to the controller; this prevents invalid state and simplifies error handling.
- **Small, single-responsibility flows** — each ‘Nav*‘ method implements a single flow (e.g., add expense, edit income) and is kept short so unit testing and review are straightforward.
- **Clear user feedback** — success/error/warning/info helpers are used consistently so test scripts and manual testers can assert expected outcomes easily.
- **Terminal-aware formatting** — ‘PlatformUtils‘ normalizes terminal specifics so colors and simple formatting work on Windows and POSIX terminals.

6.7 Short walkthrough — each feature

The walkthrough below is split into small parts; each describes how the UI supports a feature, the user steps, and the relevant code points. This list covers all main menu features.

6.7.1 Dashboard

- Goal: show quick status (date, total balance, list of wallets and counts of transactions).
- UI: ‘Dashboard::Display‘ prints the header, total balance (green/red), and a table of wallets (‘src/Views/DashBoard.cpp‘).

```
=====
PERSONAL FINANCE MANAGER - DASHBOARD
=====

Today's Date: 2025-12-28

Total Balance: 0 VND
[i] No wallets found. Create a wallet from Wallet Menu.
[i] No transactions recorded yet.

+-----+-----+
| Wallet Name | Balance | Transactions |
+-----+-----+
|No wallets found | | |
+-----+-----+



[M] Main Menu | [ESC] Exit          Dashboard
```

Figure 2: Main dashboard (date, total balance, wallets)

6.7.2 Main Menu

- Goal: provide entry points to all app features via numbered options.
 - UI: ‘Menus::DisplayMainMenu’ shows options and prints a shortcut footer. Navigation is handled by ‘NavigationController::Run’.

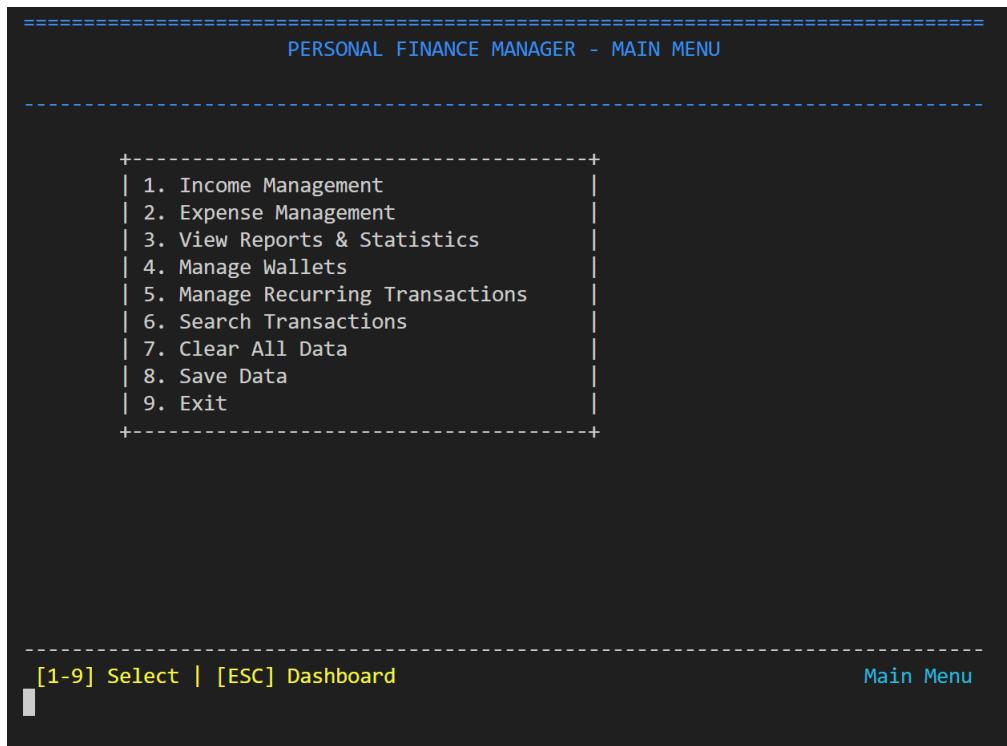


Figure 3: Main menu with shortcuts

6.7.3 Income (Add / View / Edit / Delete + Manage Sources)

- Add: select wallet, amount (validated), date, description, source; ‘NavController::HandleAddIncome’ then ‘AppController::AddTransaction’.
- View/Edit/Delete: listing and index-based selection flows are implemented in ‘HandleViewIncome’, ‘HandleEditIncome’ and ‘HandleDeleteIncome’.
- Sources management: CRUD flows for income sources live in ‘NavIncome’ methods.

6.7.4 Expense (Add / View / Edit / Delete + Manage Categories)

- Add: wallet, amount, date, description, category; ‘NavController::HandleAddExpense’ calls ‘AppController::AddTransaction’.
- View/Edit/Delete and Category management: handled via ‘NavExpense’ and category flows.

6.7.5 Wallets (Create / View / Delete)

- Create: ‘HandleCreateWallet’ prompts for name and initial balance and calls ‘AppController::AddWallet’.

- View/Delete: lists and index-selection flows in ‘NavWallet’.

6.7.6 Reports and Analytics

- Monthly summary, spending by category, income-vs-expense and wallet balance overview are implemented in ‘NavReport’ and use ‘AppController’ aggregation helpers.

6.7.7 Recurring Transactions

- Create/View/Edit/Delete: full CRUD is implemented in ‘NavRecurring’ and ‘AppController’ provides scheduling logic.

6.7.8 Search / Clear / Save Data

- Search: provides indexed search over transactions and wallets, callable from many menus; implemented in ‘NavSearch’ and helpers in ‘AppController’.
- Clear / Save Data: data-management actions (export, save, clear sample data) are implemented in the data-flow controllers and ‘BinaryFileHelper’.

6.8 Files evidence

- Views: ‘include/Views/ConsoleView.h’, ‘src/Views/ConsoleView.cpp’, ‘include/Views/Menus.h’, ‘src/Views/Menus.cpp’, ‘include/Views/DashBoard.h’, ‘src/Views/DashBoard.cpp’.
- Input helpers: ‘include/Views/InputValidator.h’, ‘src/Views/InputValidator.cpp’.
- Controllers: ‘include/Controllers/NavigationController.h’, ‘src/Controllers/Nav*.cpp’, ‘src/-Controllers/AppController.cpp’.
- Models and utils: ‘src/Models/*’, ‘src/Utils/BinaryFileHelper.cpp’, ‘src/Utils/IdGenerator.cpp’, ‘src/Utils/Date.cpp’.
- Entry: ‘src/main.cpp’ wires the controllers and UI.

6.9 Integration with the Technical Core

While the **Technical Core** defines the fundamental data structures and algorithms, the `AppController` acts as the orchestration layer that applies these mechanisms to implement concrete application features.

Rather than re-implementing logic at the controller level, the system is designed so that all critical operations in `AppController` directly leverage the guarantees provided by the Technical Core.

6.9.1 Sorted Transaction Management

Whenever `AppController::AddTransaction` is invoked, the controller delegates the ordering responsibility to the core transaction container.

- The transaction list is maintained using the **sorted insertion by date** algorithm described in the Technical Core.
- This ensures that all downstream features (reports, date-range queries, dashboard summaries) operate on a globally ordered dataset without requiring additional sorting.

Benefit: The controller logic remains simple and readable, while chronological correctness is guaranteed at the data-structure level.

6.9.2 Fast Queries via Secondary Indices

Filtering and lookup operations exposed by `AppController` (such as viewing transactions by wallet, category, or income source) rely on the **secondary indexing mechanism**.

- Upon insertion or deletion, `AppController` updates both the primary transaction list and the corresponding index maps.
- Query handlers then retrieve relevant subsets directly from these indices instead of performing full linear scans.

Benefit: This significantly improves responsiveness for user-driven queries while keeping controller methods short and focused.

6.9.3 Date Range Queries and Reporting

Reporting features implemented in `NavReport` and aggregated by `AppController` utilize the **range query on sorted data** algorithm.

- The controller identifies the starting position using binary search.
- Only relevant transactions are scanned until the end date is exceeded.

Benefit: Reports scale with the number of matching records rather than the total dataset size, making analytics predictable and efficient.

6.9.4 Recurring Transaction Generation

During application startup and scheduled checks, `AppController` processes `recurring_transactions` using the **incremental scheduling algorithm** defined in the Technical Core.

- Each blueprint is evaluated deterministically.
- Concrete transactions are generated exactly once per valid period.

Benefit: The controller avoids duplicate generation and maintains correctness without storing redundant state.

6.9.5 Search and Filtering Operations

Search features exposed through `NavSearch` delegate evaluation logic to predicate-based filters provided by the core layer.

Benefit: This separation allows search rules to evolve independently from navigation logic, while remaining easy to test and reason about.

6.9.6 Concurrency Guarantees

All `AppController` methods that mutate shared state execute under the **mutex protection** enforced by the Technical Core.

Benefit: Thread safety is achieved transparently, allowing background services (such as auto-save) to coexist safely with user-driven actions.

7 Quality Assurance and Systematic Testing

7.1 Overview and Testing Philosophy

This section describes the quality assurance approach and testing process used for the AppController module. The system is implemented using a fully custom architecture without relying on STL containers or smart pointers. Therefore, testing was not limited to checking whether features work correctly, but also focused on data consistency, algorithm correctness, thread safety, and manual memory management.

Instead of using a single testing technique, multiple testing strategies were applied. This allows the system to be evaluated both at the logic level and under realistic usage scenarios, including edge cases and high-load conditions.

7.2 Core Testing Strategies

7.2.1 Boundary Value Analysis (BVA)

Boundary Value Analysis was used to examine system behavior at the limits of valid input ranges, where errors are more likely to occur. The tested scenarios included:

- transactions with zero value,
- negative monetary inputs,
- high-frequency identifier generation,
- date-related edge cases such as leap-year dates (February 29).

These tests helped ensure that financial calculations, date processing logic, and input validation behave consistently and safely under boundary conditions.

7.2.2 Stress and Scalability Testing

To evaluate scalability, the system was tested with large datasets containing thousands of transactions. The goals of this testing phase were:

- to verify the stability of the custom `ArrayList` structure,

- to confirm that indexed queries maintain the expected $O(\log N + K)$ performance,
- to ensure that overall performance does not degrade significantly as data size increases.

7.2.3 State Transition Testing

State Transition Testing was performed to ensure that a single user action correctly updates all related system states.

For example, when a new transaction is added, the system must:

1. update the in-memory transaction list,
2. update the corresponding wallet balance,
3. save the updated data to disk using the binary storage mechanism.

This testing approach ensures consistency between in-memory data and persistent storage.

7.2.4 Concurrency and Synchronization Testing

Because the application uses a background Auto-save thread, concurrency testing was conducted to detect race conditions and synchronization issues. The system was tested under conditions involving:

- frequent read and write operations,
- rapid user navigation between screens,
- simultaneous background data saving.

The use of `std::recursive_mutex` was confirmed to correctly protect shared resources, preventing data corruption and application freezes.

7.2.5 Manual Memory Profiling

Since the system does not use smart pointers or STL containers, memory management was tested manually through repeated *Load–Stress–Exit* cycles. These tests focused on:

- proper execution of destructors,
- correct behavior of the custom FreeList utility,
- full release of dynamically allocated memory.

8 Detailed Functional and Logical Test Cases

This section presents specific test cases used to verify the correctness of the AppController, with emphasis on financial accuracy, data safety, and stability in abnormal situations.

8.1 Financial State and Boundary Testing

8.1.1 Negative Value Injection

Action. An expense transaction with a value of $-500,000$ VND was entered.

Logic Tested. Input validation and value normalization during transaction creation.

Expected Result. The system rejects the negative value or converts it into a valid expense, preventing incorrect balance updates.

Actual Result. Passed. The system enforced positive-only values and maintained correct wallet balance.

8.1.2 Zero-Value Transaction

Action. An income transaction with a value of 0 VND was recorded.

Expected Result. The transaction is stored for record purposes, and the wallet balance remains unchanged.

Actual Result. Passed. The transaction was recorded correctly without affecting the balance.

8.1.3 Leap-Year Recurring Transaction

Action. A daily recurring transaction was configured, and the system date was set to February 29.

Logic Tested. Handling of leap-year dates in recurring transaction logic.

Expected Result. The transaction is generated correctly on the leap day.

Actual Result. Passed. The system handled the leap-year case correctly.

8.2 Referential Integrity and Data Safety

8.2.1 Protection of Linked Master Data

Action. An attempt was made to delete a wallet that contained 50 existing transactions.

Expected Result. The deletion is blocked to prevent data inconsistency.

Actual Result. Passed. The system correctly prevented deletion and preserved data relationships.

8.2.2 Identifier Collision Resilience

Action. 1,000 categories were created in a short time period.

Logic Tested. Identifier uniqueness and HashMap collision handling.

Expected Result. Each category receives a unique identifier with no data overwritten.

Actual Result. Passed. Collision handling worked correctly and no identifiers were duplicated.

9 UI Robustness, Performance, and Reliability Testing

9.1 User Interface Stability

9.1.1 Rapid Navigation under Concurrent Persistence

Action. The user rapidly switched between screens while the Auto-save thread was running.

Expected Result. The interface remains responsive with no deadlocks.

Actual Result. Passed. Proper synchronization ensured smooth operation.

9.1.2 Special Character and Long String Input

Action. Long text strings and emojis were entered into transaction descriptions.

Expected Result. The system handles input safely without memory errors.

Actual Result. Passed. The input was processed correctly without crashes or corruption.

9.2 Performance Evaluation

9.2.1 Large Dataset Retrieval

Action. A date-range query was executed on a dataset of 10,000 transactions.

Expected Result. Results are returned quickly using binary search indexing.

Actual Result. Passed. The query completed efficiently with noticeable performance improvement over linear search.

9.3 Reliability and Crash Recovery

9.3.1 Abrupt Process Termination

Action. The application was forcefully closed immediately after adding a transaction.

Expected Result. The transaction data is preserved after restarting the application.

Actual Result. Passed. The data was successfully recovered from persistent storage.

9.3.2 Full Lifecycle Memory Cleanup

Action. 500 add and delete operations were performed before exiting the program.

Expected Result. All allocated memory is released on program termination.

Actual Result. Passed. Manual checks reported no memory leaks.

10 Conclusion

The testing results show that the AppController functions correctly and remains stable under various conditions, even without using STL containers.

Through systematic testing, the system demonstrated reliable behavior in terms of financial accuracy, concurrency handling, and memory management. This confirms that a carefully designed custom implementation can achieve acceptable reliability and performance for practical use.

11 Future Work

While the Personal Finance Manager application meets the core requirements outlined in the project specification, there are several areas for potential enhancement and future development:

- **Graphical User Interface (GUI)** - Transitioning from a console-based interface to a graphical user interface would significantly improve user experience. A GUI could provide intuitive navigation, visual charts for financial data, and drag-and-drop functionality for managing transactions and wallets.
- **Multi-User Support** - Implementing multi-user capabilities would allow different users to maintain separate financial records within the same application instance. This could involve user authentication, profile management, and data isolation.

- **Cloud Synchronization** - Adding cloud synchronization features would enable users to back up their data online and access it from multiple devices. This could involve integrating with cloud storage services or developing a custom backend server for data management.
- **Mobile Application** - Developing a mobile version of the Personal Finance Manager would allow users to manage their finances on-the-go. A mobile app could leverage device features such as notifications for upcoming bills or expenses.

These enhancements would not only broaden the application's functionality but also improve its accessibility and user engagement, making it a more comprehensive tool for personal finance management.

12 Conclusion

The Personal Finance Manager project successfully delivers a robust console application for tracking and managing personal finances. Through the implementation of custom data structures, a clear MVC architecture, and multi-threaded data persistence, the application meets its core objectives of reliability, usability, and cross-platform support. The layered design ensures that business logic is well-separated from the user interface, facilitating maintainability and test. The console UI, while simple, is thoughtfully designed with consistent menus, input validation, and clear feedback to enhance user experience. Looking ahead, there are numerous opportunities for future enhancements, including the development of a graphical user interface, multi-user support, cloud synchronization, and mobile application versions. These potential improvements would further elevate the application's usability and accessibility, making it a more comprehensive tool for personal finance management.

References

- [1] Dan Bernstein. Hash functions (djb2) - comp.lang.c usenet, 1991. Accessed: 2025-12-28.
- [2] Eraser.io. Eraser diagram tool. <https://eraser.io/>, 2025. Online tool used to design UML diagrams and system architecture illustrations.
- [3] Google. Gemini. <https://gemini.google.com/>, 2025. Generative AI model used to support progress tracking, multithreading concepts, cross-platform implementation, code refactoring, and report writing.
- [4] OpenAI. Chatgpt. <https://chat.openai.com/>, 2025. AI assistant used to help draft technical reports, convert content to LaTeX, and format tables, diagrams, and documentation based on student-provided input.