A design document for your Front End, giving the overall structure of your solution, showing the classes and methods in a UML class diagram and a brief (one sentence) description of the intention of each method and class in a table

Design

Architecture
- *clearly documents structure of solution*
- *explicitly describes intention of each class and method*
- *accurately reflects solution structure*
- *clearly shows where inputs and outputs fit in*

Completeness
- *evidently addresses all required functionality (solution has parts to address all required operations and results)*
- *has specified inputs, outputs and files (only!) (Front End takes in transactions on std input, produces responses on std output, has two input files (current user accounts file, available rental units file) and has one output file (daily transaction file))*

1. Purpose

This document describes the software design for the Front End of the Banking System. It outlines the overall architecture, major classes, data flow, and interfaces. The design ensures that all functional and non-functional requirements are addressed.

The Front End:

- Reads transactions from a standard input
- Reads bank account data from a current accounts file
- Produces responses on standard output
- Writes a daily transaction file at logout

So far no other inputs or outputs are used.

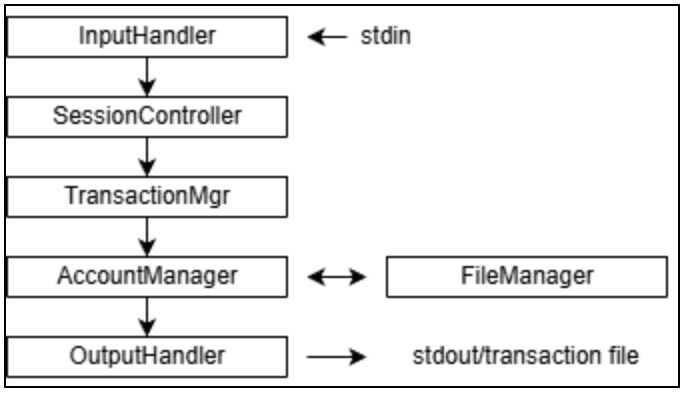
2. System Architecture

Architectural Style

The system follows a Layered/modular architecture with clearly separated responsibilities:

1. Presentation Layer - handles user input/output
2. Control Layer - Manages session flow and command execution
3. Domain Layer - Represents bank accounts and transactions
4. Persistence Layer - Handles file input and output

This separation improves maintainability, testability, and clarity



Data Flow

1. An Input handler reads a line from standard input
2. A session controller validates session state
3. A transaction manager interprets transaction codes
4. An account manager validates and updates account data
5. Classes use an account manager and file manager loads/saves files
6. An output handler displays responses and writes logs

3. Input and Output Interfaces

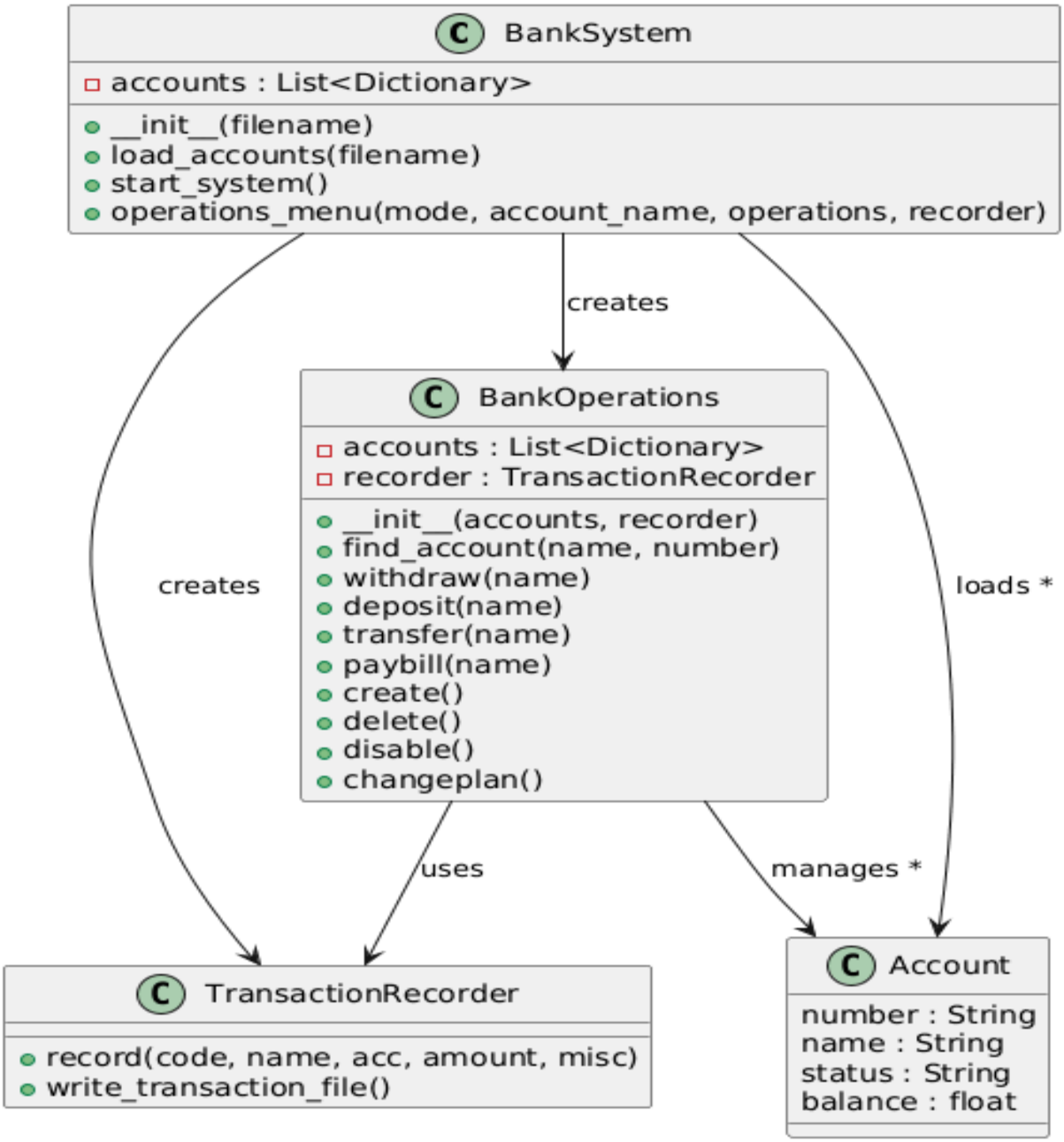
Inputs

Source	**Description**
stdin	Transaction stream
accounts file	Current bank accounts

Outputs

Source	**Description**
stdout	User feedback and errors
transaction file	Daily transaction log

4. UML Class Diagram



5. Class and Method Intentions

Class Responsibilities

Class	**Responsibility**
BankSystem	Controls the overall flow of the program by loading accounts, handling user login, and displaying menus.

BankOperations	Handles all banking transactions and updates account balances while recording each action.
TransactionRecorder	Records all completed transactions and saves them to a file at logout.
Account	Stores information about a user's bank account, including name, number, status, and balance.

Method Intentions

Class	**Method**	**Description**
bank_system	__init__	Initializer for accounts list; loads data from the specified file.
bank_system	load_accounts	Parses the input file using fixed-width slicing to populate the account list.
bank_system	start_system	The main session loop that handles initial user login and program termination.
bank_system	operations_menu	Displays available features and routes user input to specific bank operations.
transaction_recorder	__init__	Initializes an empty list to store formatted transaction strings during a session.
transaction_recorder	record	Formats transaction data into a standardized string with specific padding and widths.
transaction_recorder	write_transaction_file	Saves all records to a text file, appends the "00" termination code, and clears the list.
bank_operations	__init__	Links the logic to the current accounts list and the session's transaction recorder.
bank_operations	find_account	Searches the accounts list for

		a specific dictionary matching the provided name and number.
bank_operations	withdraw	Deducts a user-specified amount from an active account and logs it with code '01'.
bank_operations	deposit	Increases an active account's balance by a specified amount and logs it with code '04'.
bank_operations	transfer	Moves funds between two accounts owned by the same user and logs it with code '02'.
bank_operations	paybill	Deducts funds to pay a specific utility or service company and logs it with code '03'.
bank_operations	create	Admin only: Adds a new account to the system list and logs the creation with code '05'.
bank_operations	delete	Admin only: Removes an account from the system list and logs the deletion with code '06'.
bank_operations	disable	Admin only: Sets an account status to 'D' (Disabled) and logs the change with code '07'.
bank_operations	changeplan	Admin only: Updates the account's plan type and logs the change with code '08'.

6. Handling Required Functionality

Requirement	**Design Component**
Login/Logout	bank_system

Privilege Enforcement	bank_operations.validate()
Privilege Enforcement	bank_operations.withdraw()
Transfers	bank_operations.transfer()
Pay Bills	bank_operations.payBill()
Deposits	bank_operations.deposit()
Create/Delete	bank_operations.create(), delete()
Disable	bank_operations.disable()
Change Plan	bank_operations.changePlan()
File Format	transation_recorder.format(), FileManager
Error Handling	bank_operations, bank_system

7. File Handling

Input files

File	**Purpose**
Current Accounts File	Loads existing accounts

Output files

File	**Purpose**
Daily Transaction File	Stores formatted transactions

Only these files should be used by the system

8. Robustness and Error Handling

- All user input is validated before processing
- Invalid commands generate clear error messages
- No exception terminates the program unexpectedly
- Session continues after recoverable errors

9. Conclusion

This design:

- Clearly separates responsibilities

- Addresses all required transactions
- Defines explicit input/output boundaries
- Supports automated testing
- Is directly traceable to system requirement