Course: Software Project Lab – I (SPL-I)

Date: December 16, 2025
Course: Software Project Lab – I (SPL-I)
Proposed Project: *IUT CaféCache – Cafeteria E-Token & Wallet Management System*
Institution: Islamic University of Technology (IUT)

1. Introduction

Before finalizing the system design and implementation approach of *IUT CaféCache*, we conducted a study on several existing open-source console-based management systems available on GitHub. The objective of this study was **not to replicate existing systems**, but to understand how similar real-world problems are solved from a **backend and coding perspective**, particularly in environments without modern frameworks.

Since large-scale, pure console-based systems are relatively rare, we focused on projects that implement **core backend concepts** such as menu or inventory management, user authentication, transaction handling, file-based persistence, and role-based access control. These concepts directly align with the functional requirements of our proposed cafeteria system.

This document summarizes the projects studied, their key implemented features, and the relevance of their internal logic to the design of CaféCache.

2. Overview of Studied Projects

The following open-source projects were selected due to their conceptual similarity and practical backend implementations:

1. Restaurant Management System (C++)
2. Restaurant Billing System with Login (C)
3. Console-Based Restaurant Management System (C++)
4. Store / Inventory Management System (C++)

Although these projects differ in scope and language, they collectively demonstrate **standard backend workflows** that can be adapted to our system.

Restaurant Management System(C++)

Restaurant Management System (C++)

Repository: https://github.com/Suhana-Pendhari/restaurant-management-system

Core Implemented Features

- Menu management
- Order placement
- Inventory tracking
- Role separation (Customer / Owner)
- Sales reporting
- File-based data persistence

Summary

- The reference project is a **single-binary, console-based C++ application**, with most of its logic concentrated in a large `main.cpp` file (approximately 2000+ lines).
- Persistent storage is implemented using **plain text files**, including `menu.txt`, `orders.txt`, `customers.txt`, `employees.txt`, and `owner.txt`.
- The system already supports a variety of restaurant-related workflows such as menu management, customer registration, order placement, employee management, and date-based sales reporting.
- Reporting features are implemented by **parsing human-readable order logs** from `orders.txt` and aggregating data in memory.
- Several critical components required for CaféCache are missing, including:
    - Wallet / e-wallet system
    - Recharge request and approval workflow
    - Secure authentication
    - Token-based ordering

- Review and rating system
- Structured persistence (e.g., JSON or database)
- Modular code separation and transactional safety

High-Level Code Structure

- The application logic is encapsulated primarily within a single class named `Restaurant`, defined inside `main.cpp`.
- This class manages:
    1. Menu data
    2. Stock quantities
    3. Customer information
    4. Employee information
    5. Order processing
    6. Administrative dashboard and reporting
- Key in-memory data structures include:
    1. `vector<pair<string, float>> menu` — stores menu items and prices
    2. `vector<int> stock` — tracks available quantities
    3. `map<int, vector<pair<string, float>>> customerOrders` — temporary in-memory order tracking
    4. `map<int, string> customers` and `map<int, string> customerPhones` — customer data
    5. `map<int, pair<string, float>> employees` and `map<int, string> employeeWork` — employee data
    6. `map<string, float> dailySales` — used for sales aggregation

- Application startup flow:
    1. Load customers, employees, and menu data from files
    2. Display role-based menu (Customer or Owner)
    3. Execute selected operations
    4. Persist updated data back to text files

Key Functions and Implemented Behavior

Menu Management

- `loadMenuFromFile()` and `saveMenuToFile()` load and persist menu data to `menu.txt`.
- Owner-level operations include:
    - `addMenuItem()`
    - `deleteMenuItem()`
    - `updateStockQuantity()`
- Menu items are stored as space-separated values (item name, price, quantity).

Customer and Order Management

- `registerCustomer()` allows new customer creation and appends records to `customers.txt`.
- `loadCustomerData()` loads existing customers and initializes customer ID counters.
- `placeOrder()` implements an interactive order flow:
    - Displays menu

    - Accepts item selections and quantities
    - Updates stock in memory
    - Calculates total bill
    - Persists order details using `saveOrderToFile()`
- Orders are written to `orders.txt` in a **human-readable, block-style format**, including timestamps and item details.
- `viewCustomerDetails()` parses `orders.txt` and `customers.txt` to display a customer's order history.

Employee Management

- Employee data is loaded via `loadEmployeeData()`.
- CRUD operations include:
    - `addEmployee()`
    - `viewEmployeeDetails()`
    - `deleteEmployee()`
- Employee information is stored in `employees.txt`.

Dashboard and Reporting

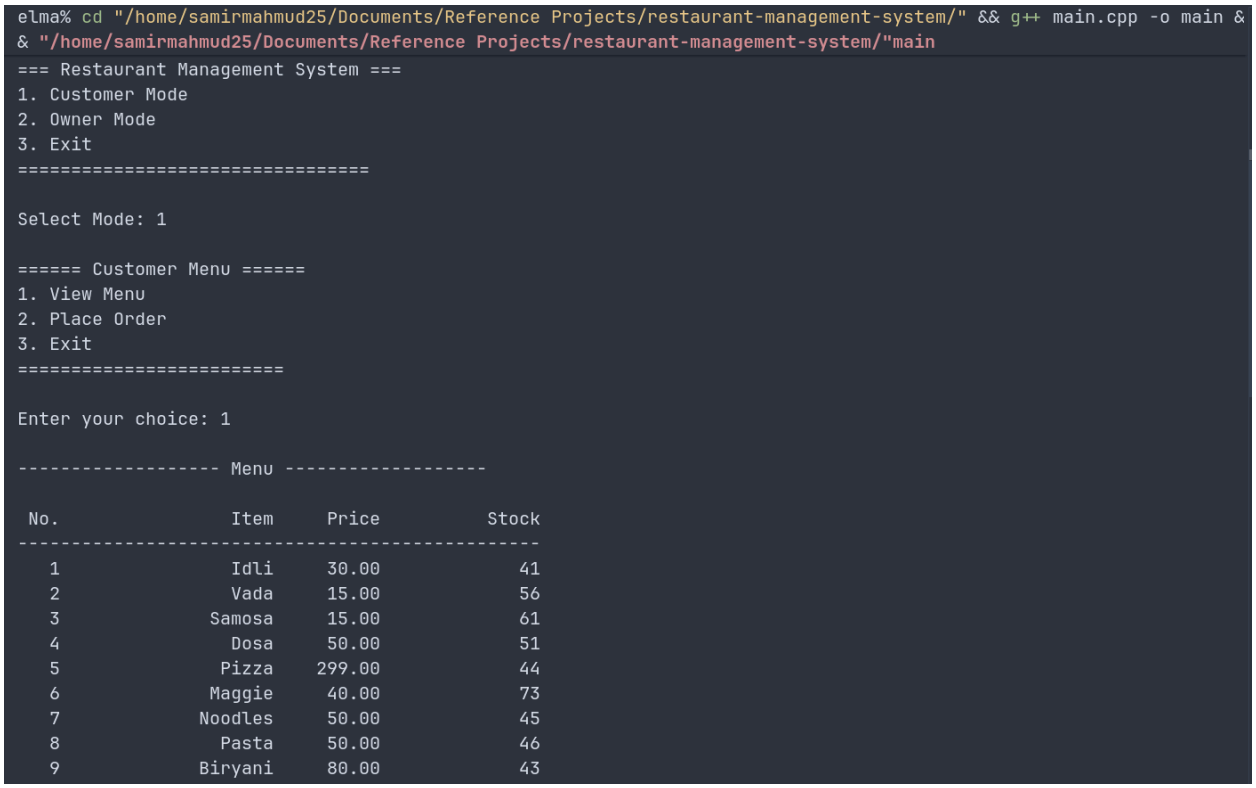
- Reporting features include:
    - Daily summary
    - Weekly summary
    - Monthly summary
    - Custom date reports

- These are implemented through functions such as:
    - `showTodaysDetails()`
    - `showLastWeeksSummary()`
    - `showLastMonthsSummary()`
- Reports are generated by parsing timestamps from `orders.txt` using `std::get_time` and aggregating totals in memory.

Owner Authentication

- The owner password is stored in plaintext inside `owner.txt`.
- `isOwner()` performs a direct string comparison to authenticate the admin.

Some Screenshots:



```
elma% cd "/home/samirmahmud25/Documents/Reference Projects/restaurant-management-system/" && g++ main.cpp -o main &
& "/home/samirmahmud25/Documents/Reference Projects/restaurant-management-system/"main
=== Restaurant Management System ===
1. Customer Mode
2. Owner Mode
3. Exit
================================

Select Mode: 1

====== Customer Menu ======
1. View Menu
2. Place Order
3. Exit
========================

Enter your choice: 1

------------------ Menu ------------------

 No.               Item     Price          Stock
-------------------------------------------------
   1               Idli     30.00             41
   2               Vada     15.00             56
   3             Samosa     15.00             61
   4               Dosa     50.00             51
   5              Pizza    299.00             44
   6             Maggie     40.00             73
   7            Noodles     50.00             45
   8              Pasta     50.00             46
   9            Biryani     80.00             43
```

```
elma% cd "/home/samirmahmud25/Documents/Reference Projects/restaurant-management-system/" && g++ main.cpp -o main &
& "/home/samirmahmud25/Documents/Reference Projects/restaurant-management-system/"main
====== Customer Menu ======
1. View Menu
2. Place Order
3. Exit
========================

Enter your choice: 2
Enter customer ID (0 for new customer): 0
Enter customer name: Samir
Enter customer phone number: 01889394454
Phone number must be exactly 10 digits. Please enter a valid phone number.
Enter customer phone number: 1234567890
Customer registered successfully!
Customer ID: 25
Welcome, Samir!

------------------ Menu ------------------

 No.               Item     Price          Stock
-------------------------------------------------
   1               Idli     30.00             41
   2               Vada     15.00             56
   3             Samosa     15.00             61
   4               Dosa     50.00             51
   5              Pizza    299.00             44
   6             Maggie     40.00             73
   7            Noodles     50.00             45
   8              Pasta     50.00             46
   9            Biryani     80.00             43
```

```
Enter item index to order (0 to finish): 1

Enter quantity: 1

Added 1 x Idli - Rs.30.00 to your order.


Enter item index to order (0 to finish): 0

===== Order Summary =====
Idli: Rs.30.00
------------------------
Total Bill: Rs.30.00

====== Customer Menu ======
1. View Menu
2. Place Order
3. Exit
==========================
```

Restaurant Billing System(C++)

Core Functional Modules (Coding Perspective)

1. User Registration & Login Module

Purpose:
Manages customer authentication and identification.

Implementation Approach:

- User credentials (username, password) are stored in a text file.
- Registration appends new user records to the file.
- Login validates user input by scanning stored credentials.

Coding Concepts Used:

- `struct` to represent user data
- `fopen()`, `fprintf()`, `fscanf()` for file persistence
- String comparison using `strcmp()`
- Input validation via loops and condition checks

Learning Outcome:

- Demonstrates how authentication systems work at a basic level without external libraries.
- Highlights the importance of identity management in transactional systems.

---

2. Menu Management Module

Purpose:
Displays available food items and prices to users.

Implementation Approach:

- Menu items are either hardcoded or loaded from a file.
- Each item is assigned an ID, name, and price.
- Menu is displayed using formatted console output.

Coding Concepts Used:

- Arrays or structures for item storage
- Loops for menu traversal
- Separation of display logic into dedicated functions

Relevance to CafeCache:

- Directly maps to cafeteria menu viewing functionality.
- Can be extended with categories, availability status, and stock control.

---

3. Order Placement & Billing Module

Purpose:
Handles the core transaction workflow of selecting items and generating bills.

Implementation Approach:

- Users select items by ID and quantity.
- Total bill is calculated dynamically.
- Order details are written to a file for record-keeping.

Coding Concepts Used:

- Arithmetic operations for bill calculation
- File append operations for order logs

- Control flow using `switch-case` or menu loops

Strengths:

- Clear procedural flow from selection → calculation → confirmation
- Easy to follow logic for beginners

Limitations:

- No transaction rollback mechanism
- No concurrency handling
- No abstraction between order creation and payment logic

---

4. Billing & Receipt Generation

Purpose:
Produces a formatted bill for the user.

Implementation Approach:

- Bill is printed on the console and optionally saved to a file.
- Contains item list, quantities, prices, and total amount.

Coding Concepts Used:

- Console formatting
- Structured output using loops
- Persistent logging via text files

Real-World Mapping:

- Mirrors real cafeteria billing receipts.

- Can be extended to support digital tokens or invoice numbers.

---

5. File Handling & Data Persistence

Purpose:
Stores user, order, and billing information persistently.

Implementation Approach:

- Uses plain text files as a lightweight database.
- Sequential file reading and writing.

Coding Concepts Used:

- `FILE*`, `fopen()`, `fclose()`
- Read/write modes (`"r"`, `"w"`, `"a"`)
- Basic error checking

Advantages:

- Simple and transparent
- Easy to debug

Drawbacks:

- No structured querying
- Vulnerable to data corruption
- Poor scalability

Key Strengths of the Reference Project

- Clear separation of logical functionalities using functions
- Realistic billing workflow
- Demonstrates foundational backend concepts:
    - Authentication
    - Transactions
    - Persistent storage
- Suitable for understanding **how real systems work internally**, without abstractions

---

Limitations Identified

1. No wallet or balance-based payment system
2. No admin approval or transaction verification
3. Plaintext password storage
4. No modular architecture
5. No reporting or analytics layer
6. No review or feedback mechanism

These limitations open opportunities for improvement in a more advanced system like **IUT CafeCache**.

Relevance to IUT CafeCache Project

This project provides a **solid foundational reference** for:

- Menu-driven billing logic

- User authentication concepts
- Order and bill lifecycle management
- File-based persistence strategies

The **CafeCache system extends this concept further** by:

- Introducing wallet-based transactions
- Adding token-based ordering
- Implementing admin approvals
- Supporting analytics and reporting
- Applying better modularization and data modeling

Key Learnings for Our Project

- How low-level billing systems are structured
- Importance of modular design from early stages
- Necessity of secure authentication
- Why transactional integrity matters
- How real-world systems evolve from simple designs

Conclusion

The Restaurant Billing System serves as a **valuable reference implementation** that demonstrates the core mechanics of a transactional application in C. While it lacks advanced features, it offers crucial insight into **backend logic, file-based persistence, and system workflows**, making it an ideal foundation for designing and extending a more robust system like **IUT CafeCache**.

Console

Console-Based Restaurant Management System

Repo:
https://github.com/shohan2032/Console-Based-Restaurant-Management-System-Project.git

Project Analysed: *Console-Based Restaurant Management System Project*
Technology: C++ (Console-based, single `main.cpp`)

Summary

- The project is implemented as a **single-file C++ console application** (`main.cpp`, approximately 650 lines).
- Persistent data storage relies entirely on **plain text files**, including `FoodMenu.txt`, `ownerList.txt`, and `customerPanel.txt`.
- Currently implemented functionalities include:
    - Owner authentication using `ownerList.txt`
    - Menu management (add, edit, delete food items)
    - Owner management (add, edit, delete owners)
    - Customer sign-up and sign-in stored in `customerPanel.txt`
    - Viewing menu items
    - Placing orders with quantity checks
    - Immediate stock updates and bill calculation
- Features **not implemented** relative to IUT CafeCache requirements include:
    - Wallet/e-wallet system
    - Recharge request and approval workflow
    - Persistent order history

    - Unique electronic order tokens
    - Sales reporting and analytics
    - Review and rating system
- Notable code quality issues include extensive global state usage, fixed-size arrays, plaintext password storage, fragile file parsing logic, platform-dependent system calls, and lack of modular design or testing.

Key Code Locations and Functional Flow

File Reading and Data Loading

- `ownerListRead()`
  Loads owner data from `ownerList.txt` into the global array `ownerList[]`.
- `foodListRead()`
  Reads food menu data from `FoodMenu.txt` into the global `foodlist[]` array.
- `customerPanelRead()`
  Loads registered customer credentials from `customerPanel.txt` into `customerList[]`.

These functions are executed during program initialization to populate in-memory data structures.

Menu Display

- `seeAllFood()`
  Prints the current food menu stored in memory to the console. This function forms the basis for both owner and customer interactions with menu data.

Owner Workflow (`class owner`)

- `ownerInfo()`
  Collects owner credentials (ID, name, contact number) from user input.

- `ownerChecking()`
  Verifies owner credentials against records stored in `ownerList.txt`.

- `ownerSelection()`
  Presents an interactive menu for owner-specific operations.

- Menu management:
  - `addItems()`
  - `editItems()`
  - `deleteItem()`

- Owner management:
  - `addOwner()`
  - `editOwner()`
  - `deleteOwner()`
  - `seeAllOwner()`

Editing or deleting records requires **rewriting the entire text file**, which introduces performance and data consistency risks.

Customer Workflow (`class customer`)

- `customercheck()`
  Handles both sign-in and sign-up logic. Credentials are stored in plaintext within `customerPanel.txt`.

- `customerSelection()`
  Provides a customer-facing menu for viewing food items and placing orders.

- `orderFood()`
  Handles interactive food ordering and invokes quantity checks.

Ordering and Stock Update

- `quantityChecking(name, quantity)`
  Verifies item availability, updates the global bill amount, and rewrites `FoodMenu.txt` to decrement stock quantities.

- The vector `vc` temporarily stores ordered items during a single execution.

- **No persistent order record** is created; once the program exits, all order data is lost.

Observed Data Formats

- **FoodMenu.txt**
  Contains menu entries in strict whitespace-separated format:
  `menuNumber foodName foodPrice foodQuantity`

- **ownerList.txt**
  Stores owner data in repeated blocks:
  `ID NAME NUMBER`

- **customerPanel.txt**
  Stores customer usernames and passwords in plaintext.

These formats are tightly coupled to parsing logic, making the system fragile in case of file corruption or format deviation.

Feature Mapping to IUT CafeCache Requirements

A. User Operations (Student / Teacher)

- **View Menu:**
  Implemented using `seeAllFood()`. Serves as a suitable baseline.

- **View Wallet Status:**
  Not implemented. No wallet model or balance tracking exists.

- **View Recharge Requests:**
  Not implemented.

- **Order Food with Electronic Token:**
  Partially implemented. While users can order food and receive a bill, there is:

  - No persistent order storage
  - No token generation
  - No payment abstraction (wallet or otherwise)

- **Purchase History:**
  Not implemented due to lack of order persistence.

B. Admin Operations (Provost Office / Staff)

- **Add / Update Menu:**
  Implemented with full-file rewrites.

- **Approve Recharge Requests:**
  Not implemented.

- **View Purchase Logs:**
  Not implemented.

- **Sales Reports (Daily/Weekly/Monthly):**
  Not implemented.

- **Price Management:**
  Implemented, but without price history tracking.

Code Quality, Portability, and Security Concerns

1. **Global State & Fixed-Size Arrays**
   Extensive reliance on global arrays (e.g., size 100 or 1000) limits scalability and maintainability. Dynamic containers such as `std::vector` are recommended.

2. **Platform-Specific Calls**
   Use of `system("cls")` and `system("color")` makes the program Windows-dependent.

3. **Plaintext Password Storage**
   Storing passwords directly in text files poses severe security risks.

4. **Fragile File Parsing**
   File operations assume perfect formatting; any inconsistency can crash or corrupt program logic.

5. **No Concurrency or Atomicity**
   Multiple concurrent executions can corrupt data files.

6. **Minimal Error Handling**
   File open and write failures are not consistently checked.

7. **Use of `goto`**
   Reduces readability and violates structured programming principles.

Review and Rating System — Design Considerations

Data to Store

- Review ID
- Customer ID
- Item ID
- Rating (1–5)

- Comment
- Timestamp
- Approval flag

Key Challenges

- Preventing fake reviews
- Ensuring only verified customers can review
- Moderation overhead
- Efficient aggregation of ratings

Proposed Solution

- Allow reviews only for items present in completed orders
- Store reviews in structured storage (SQLite or JSON)
- Maintain average ratings per item
- Include admin moderation tools

Refactoring and Implementation Roadmap for IUT CafeCache

1. **Modularization**
    - Separate models, services, persistence, and CLI layers.
2. **Structured Storage**
    - Prefer SQLite for transactional integrity and query support.

3. **Wallet and Recharge Workflow**
    - Implement atomic transactions for balance updates and approvals.
4. **Order Tokens**
    - Generate unique identifiers for each order.
5. **Security Improvements**
    - Hash passwords
    - Remove platform-dependent calls
6. **Testing**
    - Introduce unit and integration tests.

Minimal Viable Enhancements Toward CafeCache

- Persist orders in a structured format
- Introduce wallets and recharge approval
- Generate order tokens
- Add review-rating support
- Secure credential storage

Conclusion

This console-based restaurant management system provides a solid foundational example of menu handling, user interaction, and billing logic using C++. However, its limitations in persistence, security, modularity, and extensibility highlight why more structured design

practices are necessary for real-world systems. The insights gained from this project directly inform the architectural and implementation decisions required for building **IUT CafeCache** as a scalable, secure, and industry-aligned cafeteria management system.

Screenshots:

```
elma% cd "/home/samirmahmud25/Documents/Reference Projects/Console-Based-Restaurant-Management-System-Project/" &&
g++ main.cpp -o main && "/home/samirmahmud25/Documents/Reference Projects/Console-Based-Restaurant-Management-Syste
m-Project/"main
sh: line 1: cls: command not found
To enter as a owner click 1:
To enter as a customer click 2:
2
sh: line 1: cls: command not found
Enter 1 for signIN.
Enter 2 for signUP.
Enter 0 for exit.
2
sh: line 1: cls: command not found
Enter your email address:
mhsamir25@gmail.com
Enter your password:
1234
sh: line 1: color: command not found
..SignUp Successfully Done..
sh: line 1: cls: command not found
sh: line 1: color: command not found
====================
Welcome to Resturant
1.To See all food enter 1.
2.To Order food enter 2.
3.To exit enter 0.
1
sh: line 1: cls: command not found
sh: line 1: color: command not found
Menu Number:3
Food Name:dim
```

```
====================
Welcome to Resturant
1.To See all food enter 1.
2.To Order food enter 2.
3.To exit enter 0.
1
sh: line 1: cls: command not found
sh: line 1: color: command not found
Menu Number:3
Food Name:dim
Price:900
Stock Available:2

Menu Number:4
Food Name:am
Price:900
Stock Available:3

Menu Number:6
Food Name:lichu
Price:100
Stock Available:4

Menu Number:4
Food Name:apple
Price:230
Stock Available:3
```

Store

Store Management System

Repository Analysed: https://github.com/asp616848/Store-Management-system/tree/main
Technology: C++ (Console-based, multifile implementation)

Summary

- The project follows a **modular C++ design**, with clear separation of concerns across multiple source and header files.
- Core modules include:
    - Account management (`accounts.*`)
    - Inventory and product handling (`inventory.*`, `product.*`)
    - Store controller logic (`store.*`)
    - Program entry point (`main.cpp`)
- Persistent storage is implemented using **CSV files**:
    - `inventory.csv` for product data
    - `users.csv` for account data (with basic obfuscation)
- Implemented features that are directly relevant to IUT CafeCache include:
    - User and merchant account models
    - Per-user wallet balance
    - Inventory management with quantity tracking
    - Purchase flow with balance deduction and stock updates
    - Sales tracking per product
    - Aggregated reporting (top selling products, highest spending users)

- Missing or partial features relative to IUT CafeCache:
    - Order tokens
    - Recharge request and approval workflow
    - Persistent, queryable order history
    - Review and rating system
    - Strong authentication and security mechanisms
    - Time-based analytics and visual reports

Core Code Structure and Responsibilities

Entry Point

- **`multifile/main.cpp`**
    - Initializes the `Store` object
    - Allows the user to choose between account and inventory modes
    - Acts primarily as a launcher with minimal logic

Store Controller

- **`multifile/store.cpp`** / **`store.hpp`**
    - Central coordinator of system behavior
    - Handles:
        - Login

        - User and seller creation
        - CLI loops for accounts (`run()`) and inventory (`runInv()`)
        - Purchase flow (`MakeAPurchase()`)
        - Persistence via `saveToFile()` and `loadFromFile()`
        - Stock alerts and analytics (top sellers, top spenders)

The `Store` class effectively acts as the application layer, coordinating interactions between accounts and inventory.

Account Management

- **`multifile/accounts.cpp` / `accounts.hpp`**
    - Implements an account hierarchy:
        - `Account` (base)
        - `CustomerAccount`
        - `MerchantAccount`
    - Features:
        - Balance storage and updates
        - Expenditure tracking
        - Viewing balances (merchant-side)
- Persistence:
    - Account data stored in `users.csv`
    - Passwords are obfuscated using a Caesar cipher and numeric offset

---

Inventory and Product Management

- **`multifile/inventory.cpp`** / **`inventory.hpp`**
- **`multifile/product.cpp`** / **`product.hpp`**
    - Models product attributes:
        - ID, name, category, price, quantity, sales count
    - Inventory operations:
        - Add, remove, and find products
        - Load/save inventory via CSV

`inventory.csv` structure:

```
id,name,category,price,quantity,sales
```

    -

Feature Mapping to IUT CafeCache Requirements

A. User Operations

View Menu

- **Implemented**
- Inventory items are displayed via `printAllProducts()` and during purchase flow.
- Suitable as a base; can be extended with categories, filters, and pagination.

View Wallet Status

- **Implemented**
- `CustomerAccount` maintains a balance with getter and updater methods.
- CLI allows users to add funds directly.
- **Limitation:** No admin approval or recharge request tracking.

View Recharge Requests

- **Not Implemented**
- No request queue or approval workflow exists.

Order Food with Unique Electronic Token

- **Partially Implemented**
- `MakeAPurchase()`:
    - Validates quantity
    - Deducts inventory
    - Deducts user balance
    - Updates sales statistics
- **Missing:**
    - Unique order token
    - Persistent order record
    - Order status lifecycle

Purchase History

- **Not Implemented**
- Only aggregated expenditure is stored; no per-order history is available.

B. Admin Operations

Add / Update Menu

- **Implemented**
- Inventory supports add, remove, and update operations with CSV persistence.

Approve Recharge Requests

- **Not Implemented**

View Purchase Logs

- **Not Implemented**
- Only aggregate statistics exist.

Sales Reports (Daily / Weekly / Monthly)

- **Not Implemented**
- No timestamped order data is available for temporal analysis.

Manage Items and Prices

- **Implemented**
- Price changes supported, but no price history is preserved.

Conclusion

The Store Management System is a strong and practical reference project that already implements several core features required by **IUT CafeCache**, including inventory handling, wallet balance management, and purchase flows. However, to achieve full functional parity, the system must introduce structured order persistence, secure authentication, recharge approval workflows, and a review-rating mechanism. With moderate refactoring and the addition of structured storage (preferably SQLite), this project can evolve into a robust, industry-aligned cafeteria management backend.

Coding Perspective & Implementation Analysis

Coding Perspective & Implementation Analysis

Project: IUT CafeCache – Cafeteria Token & Wallet Management System

Purpose of This Analysis

This document outlines how the **IUT CafeCache system** can be implemented from a **pure coding perspective**, focusing on:

- Internal data structures
- Program flow
- User roles and interactions
- Persistence mechanisms
- Extensibility for future SPL-II enhancements

The goal is to demonstrate a **clear backend-oriented design**, suitable for a **console-based C++ application**, while keeping the system scalable and modular.

Overall System Architecture (Conceptual)

The system can be viewed as a **layered console application** with the following conceptual layers:

1. **User Interface Layer (CLI)**
    - Menu-driven console interactions
    - Handles input/output only

2. **Business Logic Layer**
    - Wallet management
    - Order processing
    - Token generation
    - Recharge approval logic
3. **Data Management Layer**
    - In-memory data structures (linked lists, vectors, maps)
    - File/database persistence

Each layer is kept logically separate to reduce coupling and improve maintainability.

User Roles & Responsibilities

1. Student / Teacher (End Users)

- View cafeteria menu
- View wallet balance
- Request wallet recharge
- Place food orders
- Receive electronic token
- View purchase history

2. Admin (Provost / Cafeteria Authority)

- Add/update/remove menu items

- Approve recharge requests
- View sales reports
- View order logs

Core Data Structures (Very Important)

We can use a **hybrid approach**:

- **Linked Lists** for dynamic, frequently changing entities
- **Vectors/Maps** where indexed access is needed
- **Files / DB** for persistence

---

1. User Data (Linked List)

```
struct User {
    int userId;
    string name;
    string role; // student / teacher / admin
    double walletBalance;
    User* next;
};
```

Why Linked List?

- Dynamic number of users
- Frequent insertions (new registrations)
- No need for random index-based access
- Demonstrates low-level memory handling

---

2. Menu Items (Linked List)

```
struct MenuItem {
    int itemId;
    string name;
    double price;
    int stock;
    MenuItem* next;
};
```

Why Linked List?

- Menu items can be added/removed dynamically
- No fixed size
- Admin operations map naturally to list insert/delete

---

3. Orders (Linked List of Orders)

```
struct Order {
    int orderId;
    int userId;
    string token;
    double totalAmount;
    string status; // PENDING, PAID, COMPLETED
    Order* next;
};
```

Each order may internally store:

- A list/vector of ordered items
- Timestamp

Why Linked List?

- Orders are continuously created
- Easy to append new orders
- Sequential traversal for logs and reports

---

4. Recharge Requests (Queue via Linked List)

```
struct RechargeRequest {
    int requestId;
    int userId;
    double amount;
    bool approved;
    RechargeRequest* next;
};
```

Why Linked List / Queue?

- Recharge requests follow FIFO nature
- Admin processes them sequentially
- Queue logic fits perfectly

Program Flow (High-Level Execution)

Step 1: Program Startup

- Load users from file → build linked list
- Load menu from file → build linked list
- Load wallet balances
- Load orders, recharge requests, reviews

Step 2: Login & Role Detection

```
1. Login
2. Exit
```

After login:

- If Student/Teacher → User Menu
- If Admin → Admin Menu

Step 3: User Menu Flow

```
1. View Menu
2. View Wallet
3. Request Recharge
4. Order Food
5. View Purchase History
6. Submit Review
7. Logout
```

Order Flow (Important):

1. User selects items
2. System checks stock
3. Calculates total
4. Deducts wallet balance
5. Generates unique token
6. Stores order in linked list
7. Persists order to file

Step 4: Admin Menu Flow

```
1. Add / Update Menu
2. Approve Recharge Requests
3. View Order Logs
4. View Sales Reports
5. Moderate Reviews
6. Logout
```

Token Generation Logic

Tokens can be generated using:

- Combination of orderId + timestamp
- Or random alphanumeric string

Example:

```
string generateToken() {
    return "IUT-" + to_string(rand() % 100000);
}
```

Error Handling & Validation

- Check wallet balance before order
- Check stock availability
- Prevent duplicate reviews
- Handle invalid input gracefully

Conclusion

From a coding perspective, **IUT CafeCache** can be implemented as a structured, modular console-based C++ application using **linked lists as core data structures**, supported by file-based persistence and clear separation of concerns. This approach demonstrates strong understanding of backend logic, data structures, and system design while remaining fully achievable within SPL-I constraints.