OBJECT ORIENTED PROGRAMMING (CS-212)

RAILSYNC
A RAILWAY MANAGEMENT SYSTEM USING OOPS

UNDER THE GUIDANCE OF Prof. Umakanta Majhi

SUBMITTED BY -
Team Member 1: Agniv Kashyap
Scholar Id: 2312180
Team Member 2: Sanjogita Bhagowati
Scholar Id: 2312087
Group No: 23
Department: Computer Science and Engineering (Section A)
Semester: 4th (Batch: 2023 - 2027)

1. Title

RAILSYNC: Railway Management System Using Object-Oriented Programming in C++

A streamlined command-line railway reservation system built with C++ OOP principles for efficient ticketing, admin control and e-catering management.

2. Abstract

The Railway Management System is designed to simulate core functionalities of a real-world railway service using Object-Oriented Programming (OOP) concepts in C++. This project allows both administrators and users to interact with the system. Admins can monitor bookings, add or remove trains, view the train database and food orders. In contrast, users can book train tickets, manage their reservations and place e-catering orders.

This project showcases how OOP principles like encapsulation, inheritance, and polymorphism can be applied to build structured, scalable and modular applications. The system is lightweight, file-based and easily extendable, making it useful for academic and beginner-level real-world modelling.

3. Problem Statement

Managing railway operations, such as ticket bookings, updating the train database and catering services, involves handling large amounts of dynamic data and user interactions. Traditional systems often face challenges in terms of scalability, modularity, and maintainability, especially when built without structured design approaches.

The specific problem addressed in this project is:
"How can we design a simple, modular, and extensible Railway Management System using Object-Oriented Programming (OOP) principles in C++ that allows efficient handling of bookings, customer details, managing the train database and food orders through file-based data storage?"

This project aims to model real-world railway services in an academic context by using OOP techniques to create a structured, scalable and interactive system.

4. Objectives

The primary objectives of the Railway Management System project are:

- To simulate core railway operations like ticket booking, train management and e-catering services using C++.

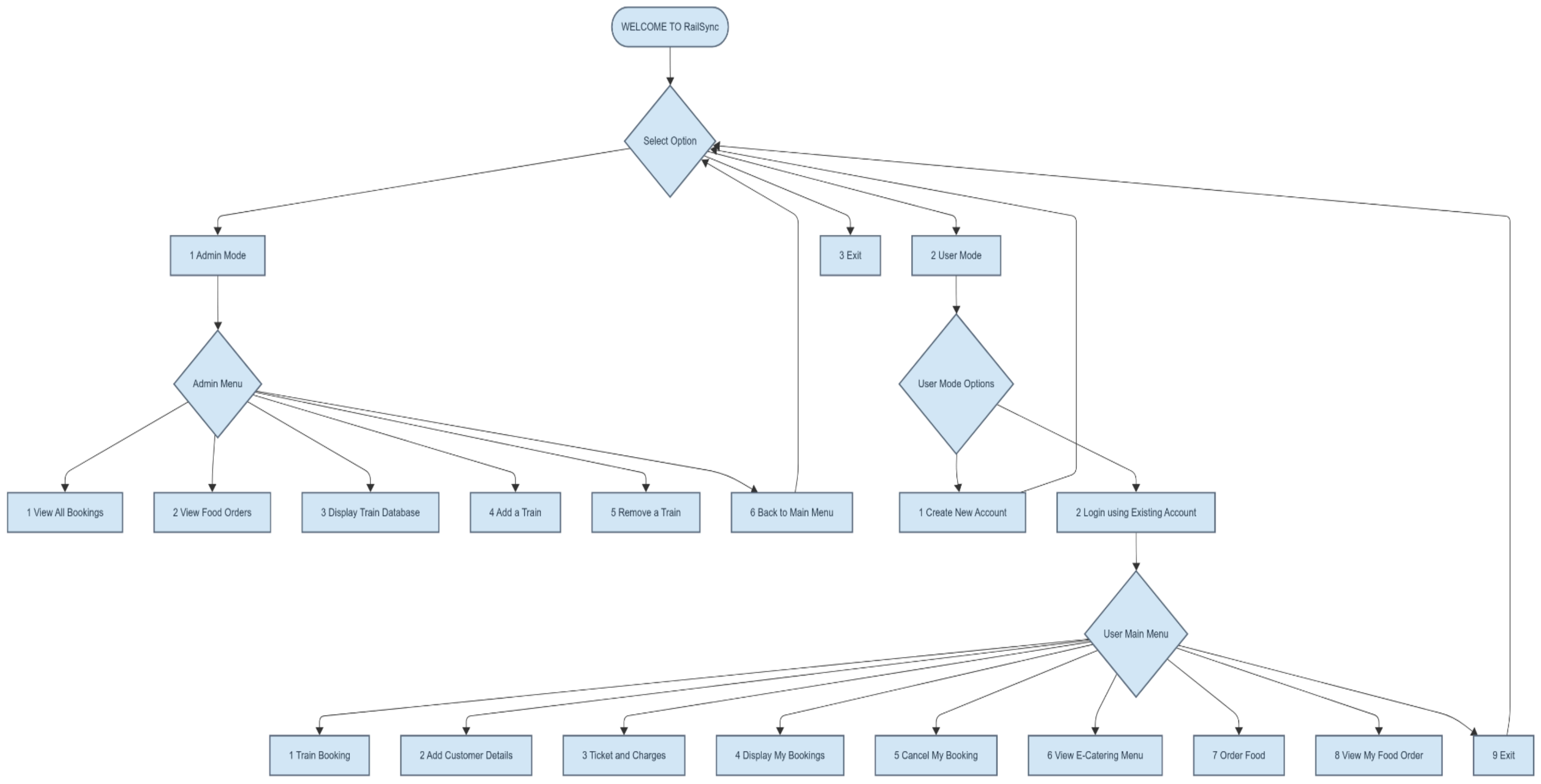
- To implement Object-Oriented Programming (OOP) concepts such as encapsulation, inheritance, polymorphism and abstraction for better code organisation and scalability.

- To design a user-friendly menu-driven interface that supports separate functionalities for administrators and users.

- To build a lightweight, modular, and easily extendable system that serves as a learning model for real-world applications.

- Using techniques to validate the system's robustness with exception handling, input validation and secure password management.

5. Workflow Diagram



The RailSync system is structured around two major modes of operation: Admin Mode and User Mode and a third option to exit the application.

- Main Menu:
  Upon launching RailSync, users are greeted with three options:
  1. Admin Mode
  2. User Mode
  3. Exit

Admin Mode Workflow:

Selecting Admin Mode redirects the administrator to the Admin Menu, offering the following functionalities:

- **View All Bookings**: Displays all existing train ticket bookings.
- **View Food Orders**: Lists all e-catering orders placed by passengers.
- **Display Train Database**: Shows the database of all trains available.
- **Add a Train**: Allows the admin to add a new train to the database.
- **Remove a Train**: Deletes a train entry from the system.
- **Back to Main Menu**: Returns to the main selection screen.

User Mode Workflow:

In User Mode, users have two options:

- **Create New Account**:
  New users can register by creating a RailSync account. After the account creation, the system redirects back to the main menu.
- **Logging in using Existing Account**:
  Registered users can log in to access additional services, including:
  - **Train Booking**: Reserve tickets for available trains.
  - **Add Customer Details:** Add traveller information.
  - **Ticket and Charges**: View ticket pricing and breakdown of charges.
  - **Display My Bookings**: Check all personal bookings made.
  - **Cancel My Booking**: Cancel an existing ticket.
  - **View E-Catering Menu**: Browse available food options.
  - **Order Food**: Place food orders associated with train travel.
  - **View My Food Order**: View all the food orders that have been placed.
  - **Back to Main Menu**: Returns to the main selection screen.

System Navigation:

- Every submenu provides a way to return to the Main Menu, ensuring smooth navigation.
- After major operations like account creation, booking, or ordering food, the user is either directed back to the Main Menu or remains within their session until they choose to exit.

6. Requirements

6.1 Software Requirements:

- **Operating System:** Windows / Linux / MacOS (Any system capable of running C++ code)
- **Compiler:** GCC (g++) / Clang
- Integrated Development Environment (IDE): Visual Studio Code / Any other suitable IDE

6.2 File Requirements:

- stations.txt : Stores the list of station names
- trains.txt: Maintains train details which include **T**rain number, Train name, Start station, End station, Departure time, Arrival time, Journey hours, AC3 price, AC2 price, AC1 price, AC3 seats available, AC2 seats available, AC1 seats available
- users.txt: User account information, username and password
- tickets.txt: Stores booked ticket records with PNR, user information, train and seat information.

6.3 Libraries and STL usage:

- #include<iostream>: For input and output (cin, cout, cerr)
- #include <fstream>: File handling (train data, bookings, etc.)
- #include <string>: String manipulation and storage
- #include <vector>: Dynamic array to store train data, food items, etc.
- #include <stdexcept>: Exception handling with invalid_argument, etc.
- #include <iomanip>: Output formatting (e.g., setw, setprecision)
- #include <sstream>: Parsing strings into individual data fields
- #include <cstdlib>: For system-level functions like system("cls") or exit()
- #include<algorithm>: Search, sorting, filtering functionalities

7. ALGORITHMS

7.1 Ticket Booking and Cancellation Logic:

File Reference: ticket.cpp, registration.cpp

Functionality:
Ticket booking and cancellation are managed through:

- **Booking Process:**
  - Capturing customer details like name, age, gender, and mobile number.
  - Displaying available trains and classes, followed by seat assignment based on availability.
  - Generating a unique PNR and saving booking information to a file (ticket.txt).
- **Cancellation Process:**
  - Searching for a booking using Ticket ID.
  - Cancelling the booking and releasing the reserved seat.
  - Updating ticket records and seat availability.
  - Handling invalid PNRs gracefully.

Algorithm Highlights:

- Unique PNR generation ensures easy referencing and management.
- Robust ticket lookup and error handling during cancellation to prevent system crashes.
- Dynamic update of records (both in-memory and file-based) ensures consistency between booking and seat availability.

OOP Concepts Used:

- **Encapsulation:** Customer and booking details are encapsulated within structured classes.

- **Abstraction:** Internal mechanisms like seat mapping and ticket file updates are hidden from the user.
- **Robustness:** Exception handling is applied for invalid input (e.g., wrong PNRs, invalid booking details).

7.2 Train Search and Filtering Logic

File Reference: database.cpp
Functionality:
Train search and station selection are handled through:

- Menus displaying all stations via displayStationsMenu().
- User inputs are **validated** (e.g., cin.fail()) before proceeding.
- Functions allow selection of boarding and destination stations (selectDestinationStation()), excluding invalid choices.

Algorithm Highlights:

- Robust station filtering: prevents the same station as both source and destination.
- User interaction is guided and error-proof.
- Train search (when paired with class Management) includes filters based on **availability**, **source**, **destination and duration.**

OOP Concepts Used:

- Abstraction: Data loading and filtering logic are hidden from user-facing functions.
- Modularity: Each logical block (menu, input, file read) is separated.

7.3 PNR Generation Logic:

File Reference: details.cpp

Functionality:

The logic for generating a **Passenger Name Record (PNR)** ensures each ticket has a unique identifier:

- A **10-digit PNR** is created using the rand() function, seeded by time(0) for randomness.
- Before confirming a new PNR, the code checks for existing entries in **tickets.txt** using the **pnrExists()** function.
- The ifstream reads the file line-by-line, and a delimiter (|) is used to parse the data and compare existing PNRs.

Algorithm Highlights:

- Prevents duplicate PNRs via linear file scan.
- Uses simple random number generation.

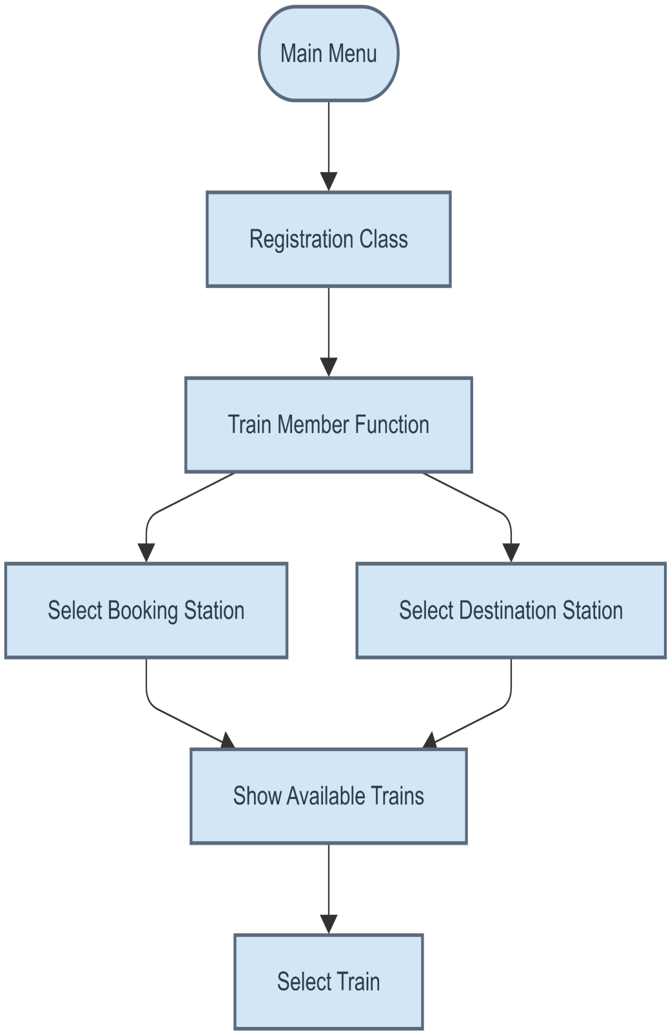
OOP Concepts Used:

- Encapsulation: PNR logic is inside the Details class.
- Static members store passenger and station data across the program.

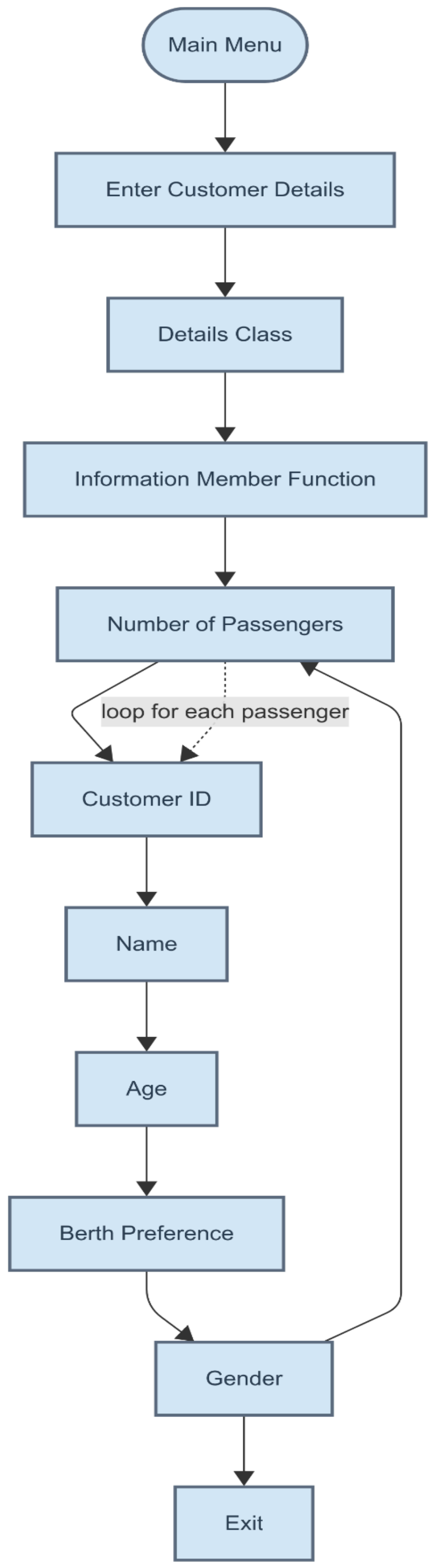
8. Technical Diagrams

Below are some diagrams that help illustrate how different features work in the project.

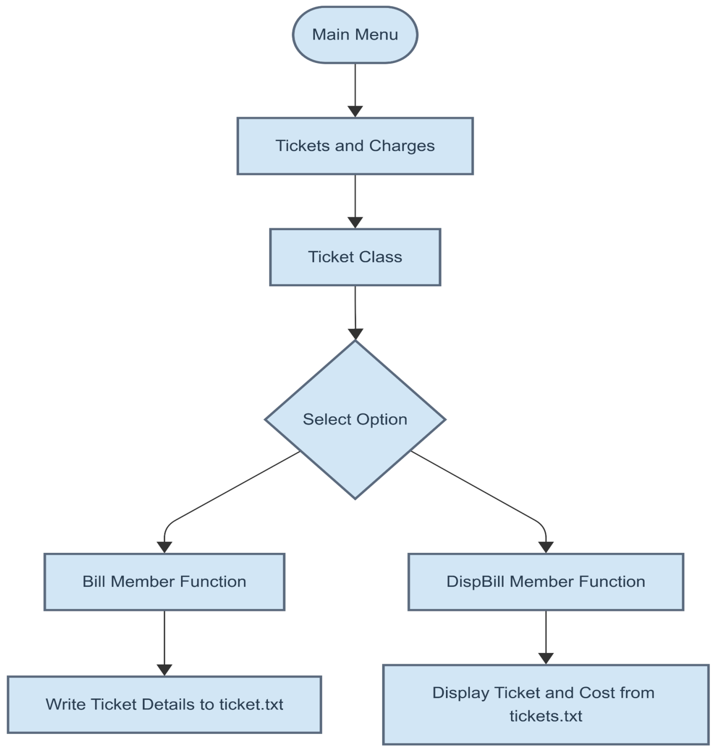
8.1 Train Booking (User Menu Component)



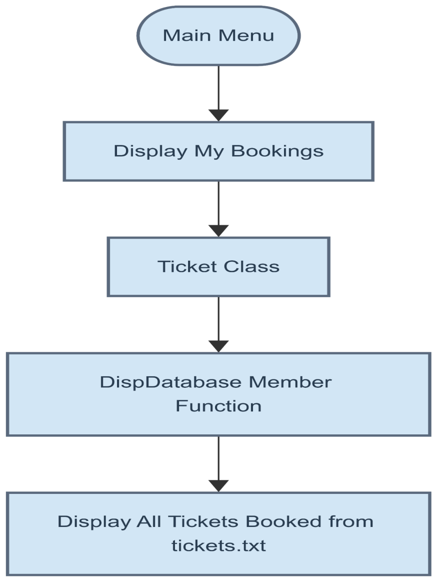
8.2 Add Customer Details (User Menu Component)



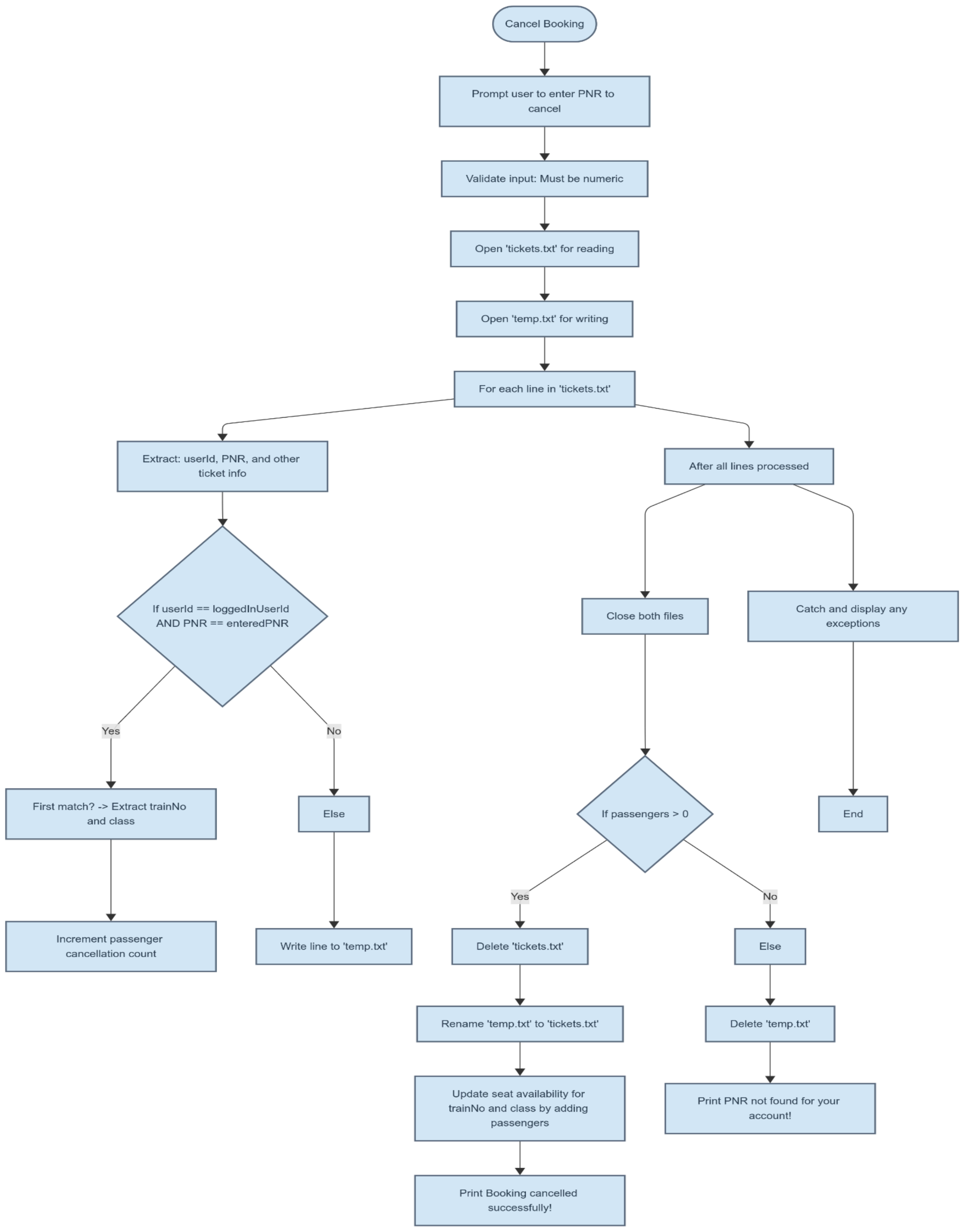
8.3 Tickets and Charges(User Menu Component)



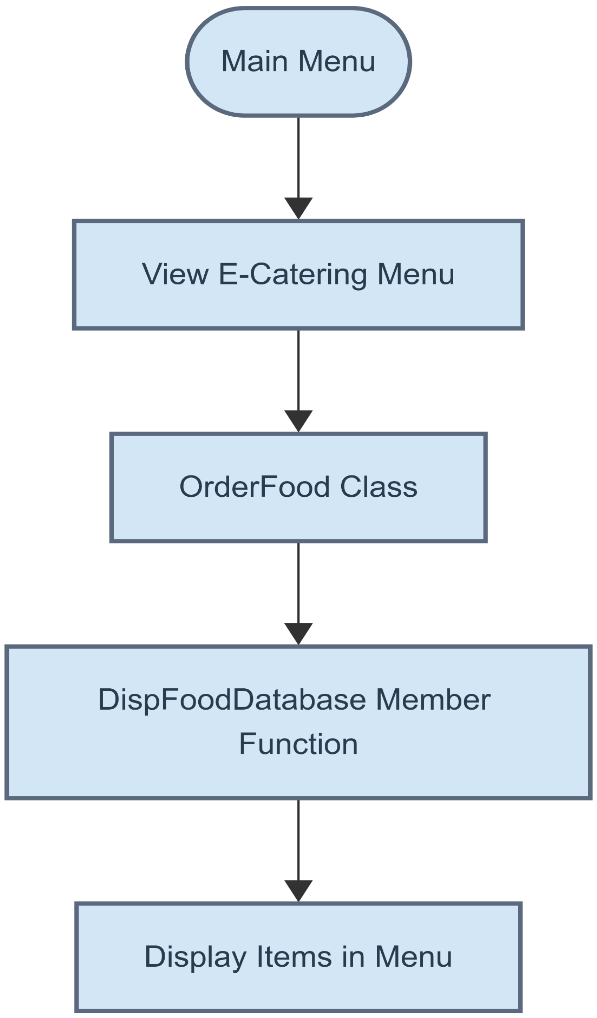
8.4 Display My Bookings(User Menu Component)


Main Menu
Display My Bookings
Ticket Class
DispDatabase Member Function
Display All Tickets Booked from tickets.txt

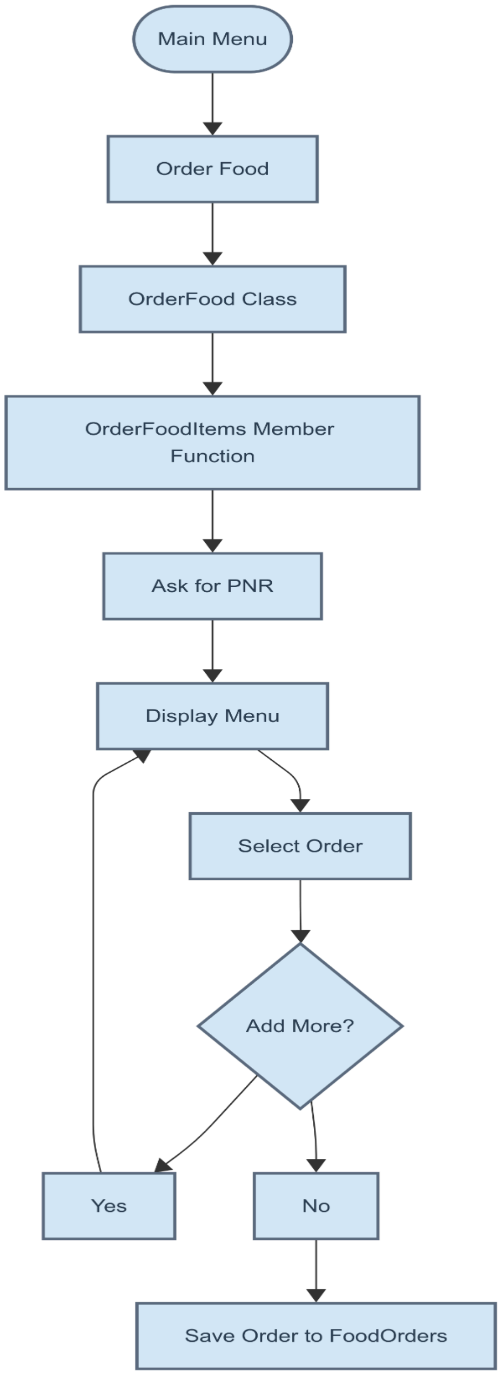
8.5 Delete My Bookings(User Menu Component)


Cancel Booking
Prompt user to enter PNR to cancel
Validate input: Must be numeric
Open 'tickets.txt' for reading
Open 'temp.txt' for writing
For each line in 'tickets.txt'
Extract: userId, PNR, and other ticket info
After all lines processed
If userId == loggedInUserId AND PNR == enteredPNR
Close both files
Catch and display any exceptions
Yes
No
First match? -> Extract trainNo and class
Else
If passengers > 0
End
Increment passenger cancellation count
Write line to 'temp.txt'
Yes
No
Delete 'tickets.txt'
Else
Rename 'temp.txt' to 'tickets.txt'
Delete 'temp.txt'
Update seat availability for trainNo and class by adding passengers
Print PNR not found for your account!
Print Booking cancelled successfully!

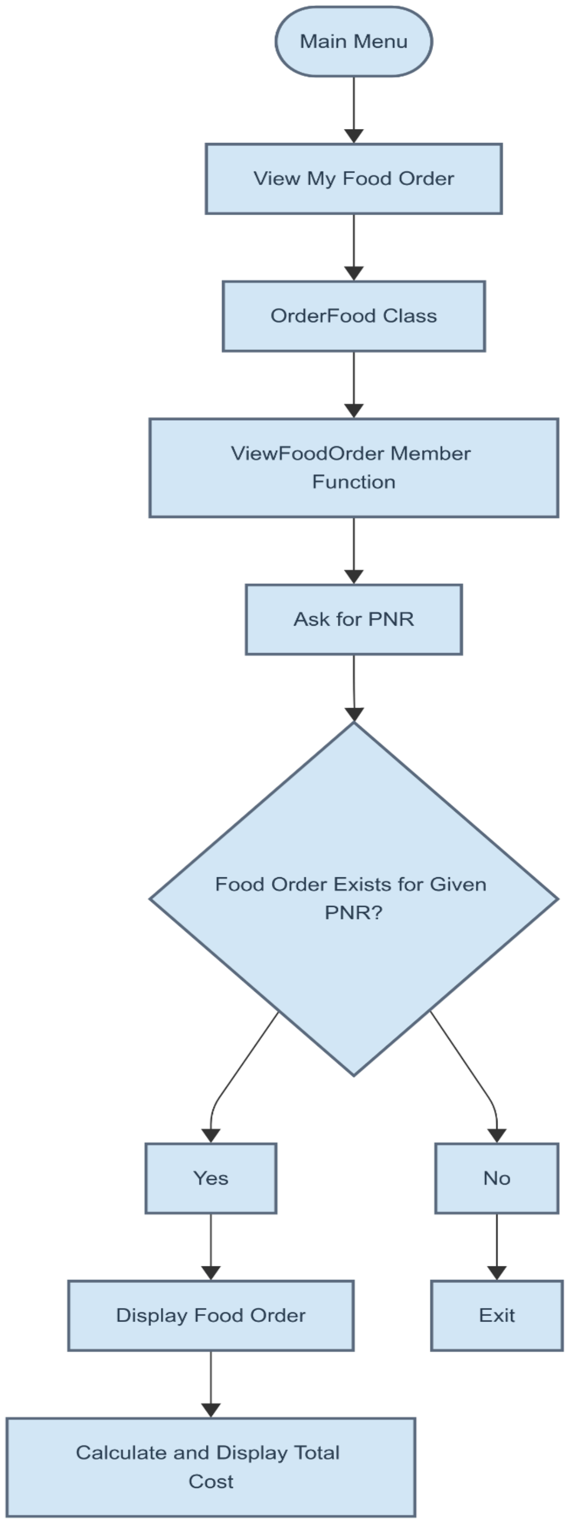
8.6 View E-Catering Menu(User Menu Component)



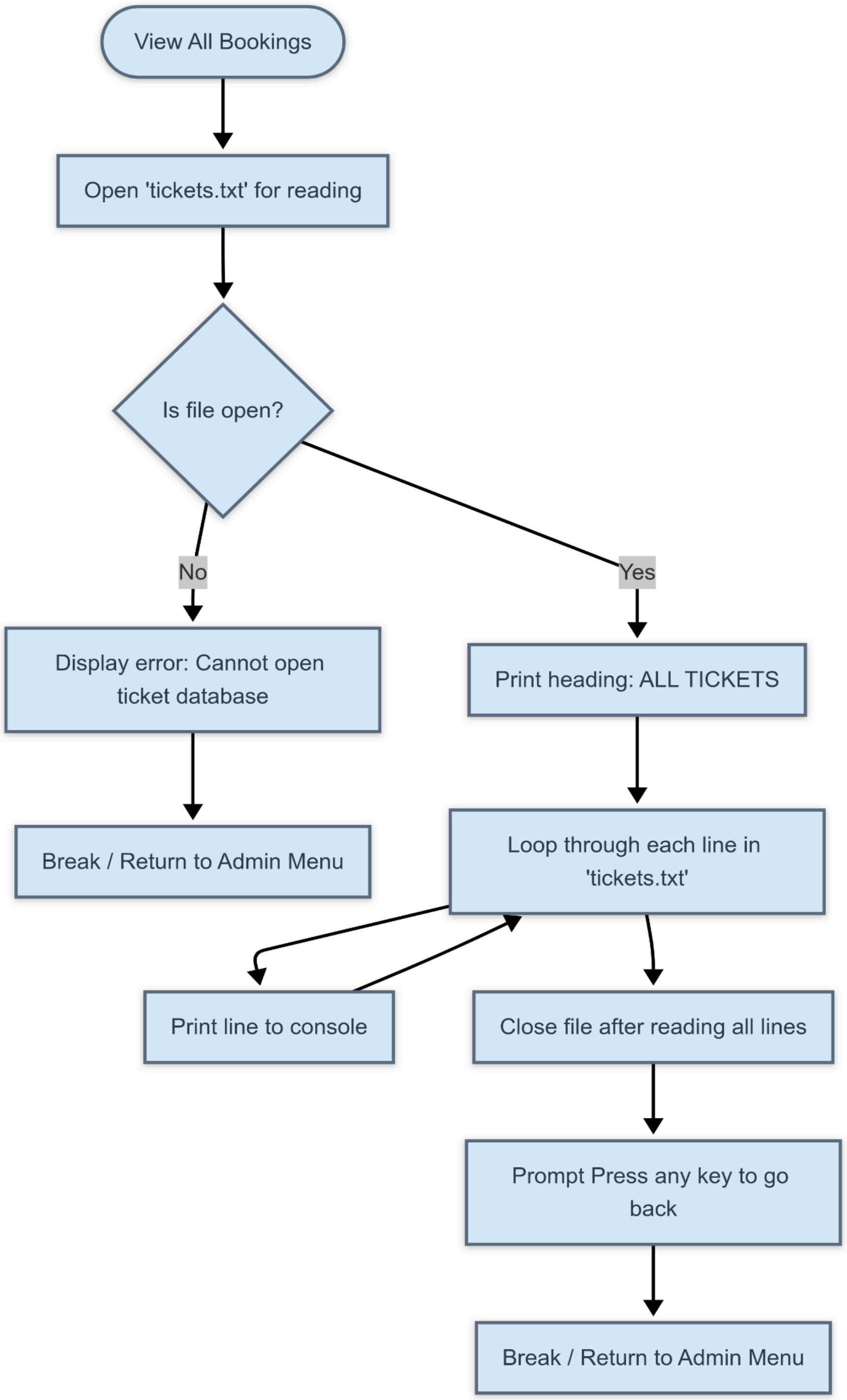
8.7 Order Food(User Menu Component)



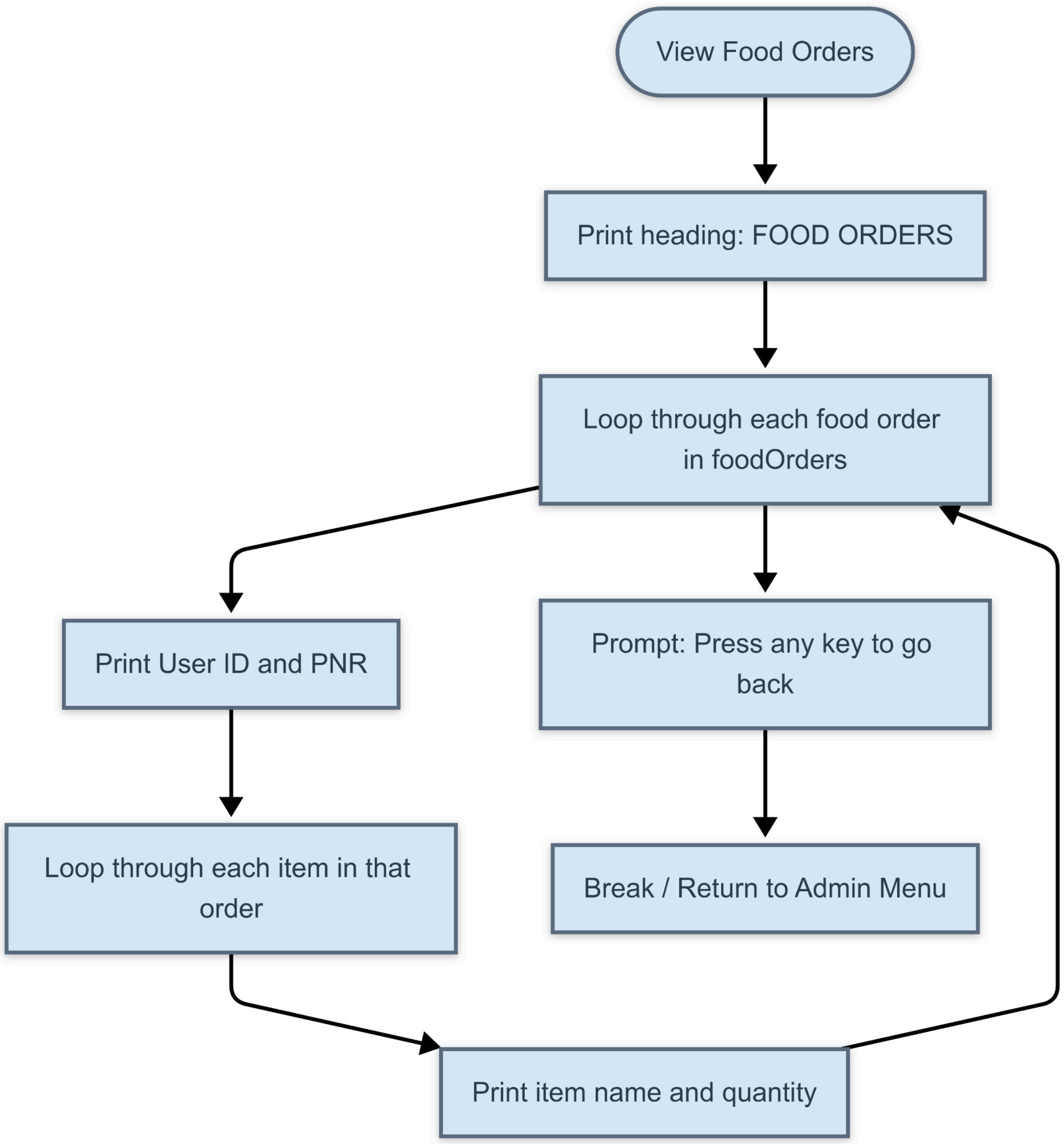
8.8 View Food Order(User Menu Component)


Main Menu
View My Food Order
OrderFood Class
ViewFoodOrder Member Function
Ask for PNR
Food Order Exists for Given PNR?
Yes
No
Display Food Order
Exit
Calculate and Display Total Cost

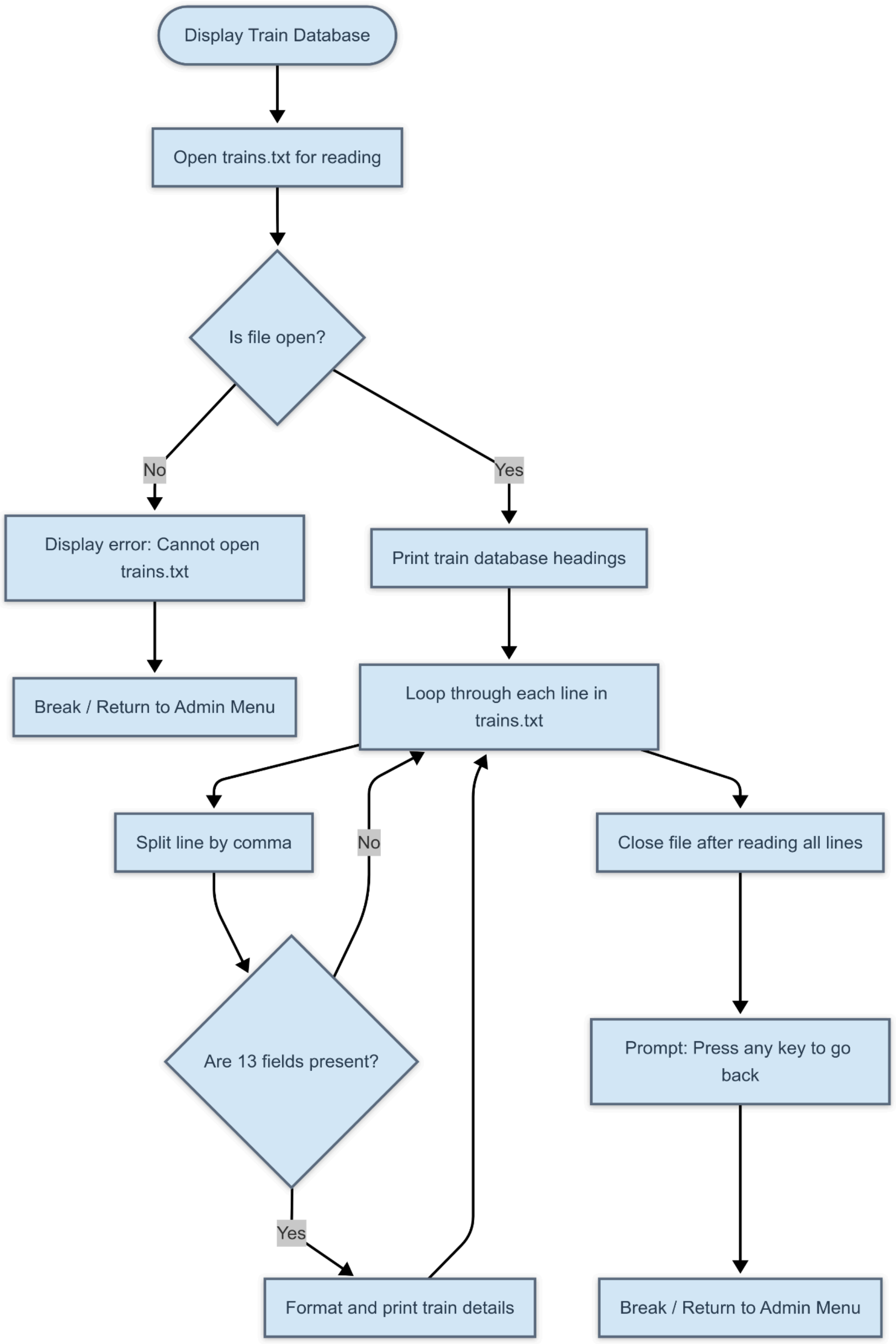
8.9 View All Bookings (Admin Menu Component)


View All Bookings
Open 'tickets.txt' for reading
Is file open?
No
Yes
Display error: Cannot open ticket database
Print heading: ALL TICKETS
Break / Return to Admin Menu
Loop through each line in 'tickets.txt'
Print line to console
Close file after reading all lines
Prompt Press any key to go back
Break / Return to Admin Menu

8.10 View Food Orders (Admin Menu Component)

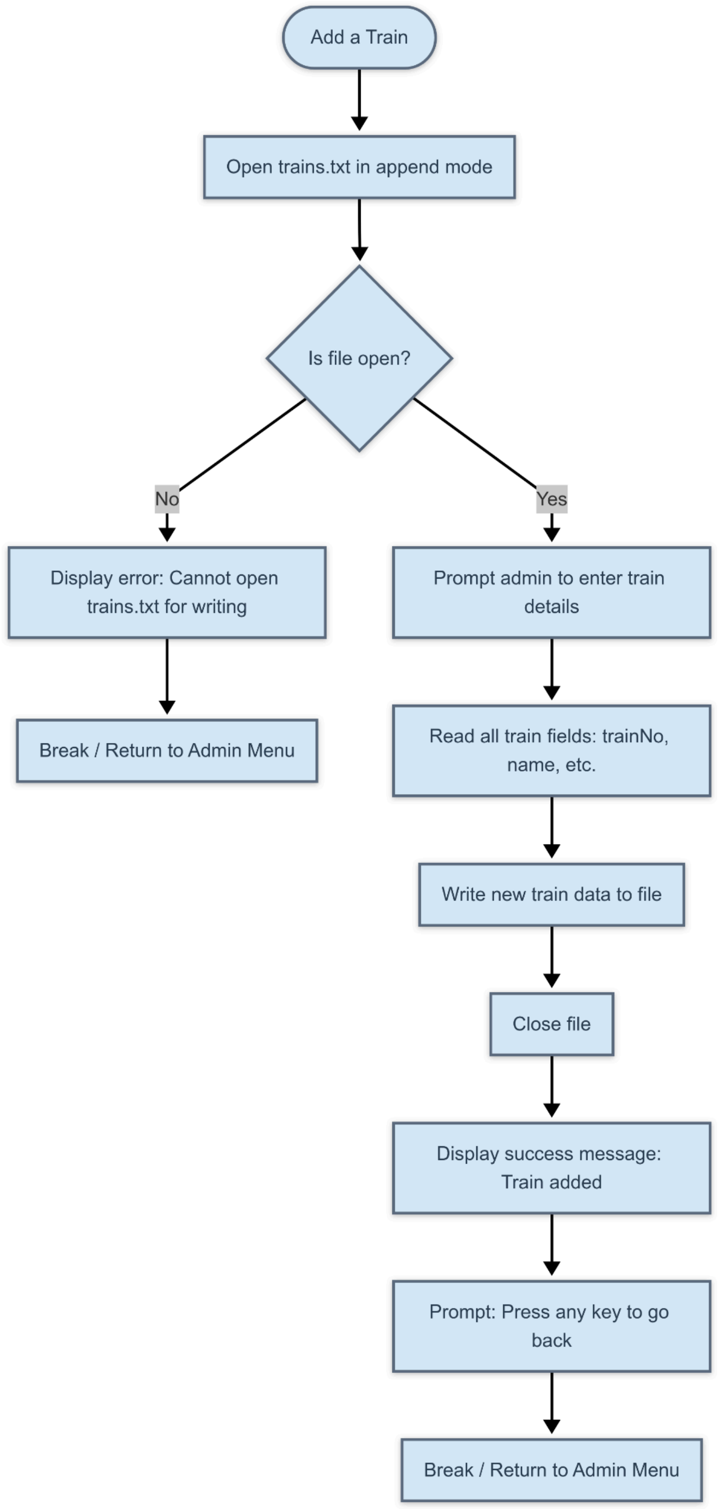

View Food Orders
Print heading: FOOD ORDERS
Loop through each food order in foodOrders
Print User ID and PNR
Prompt: Press any key to go back
Loop through each item in that order
Break / Return to Admin Menu
Print item name and quantity

8.11 Display Train Database(Admin Menu Component)

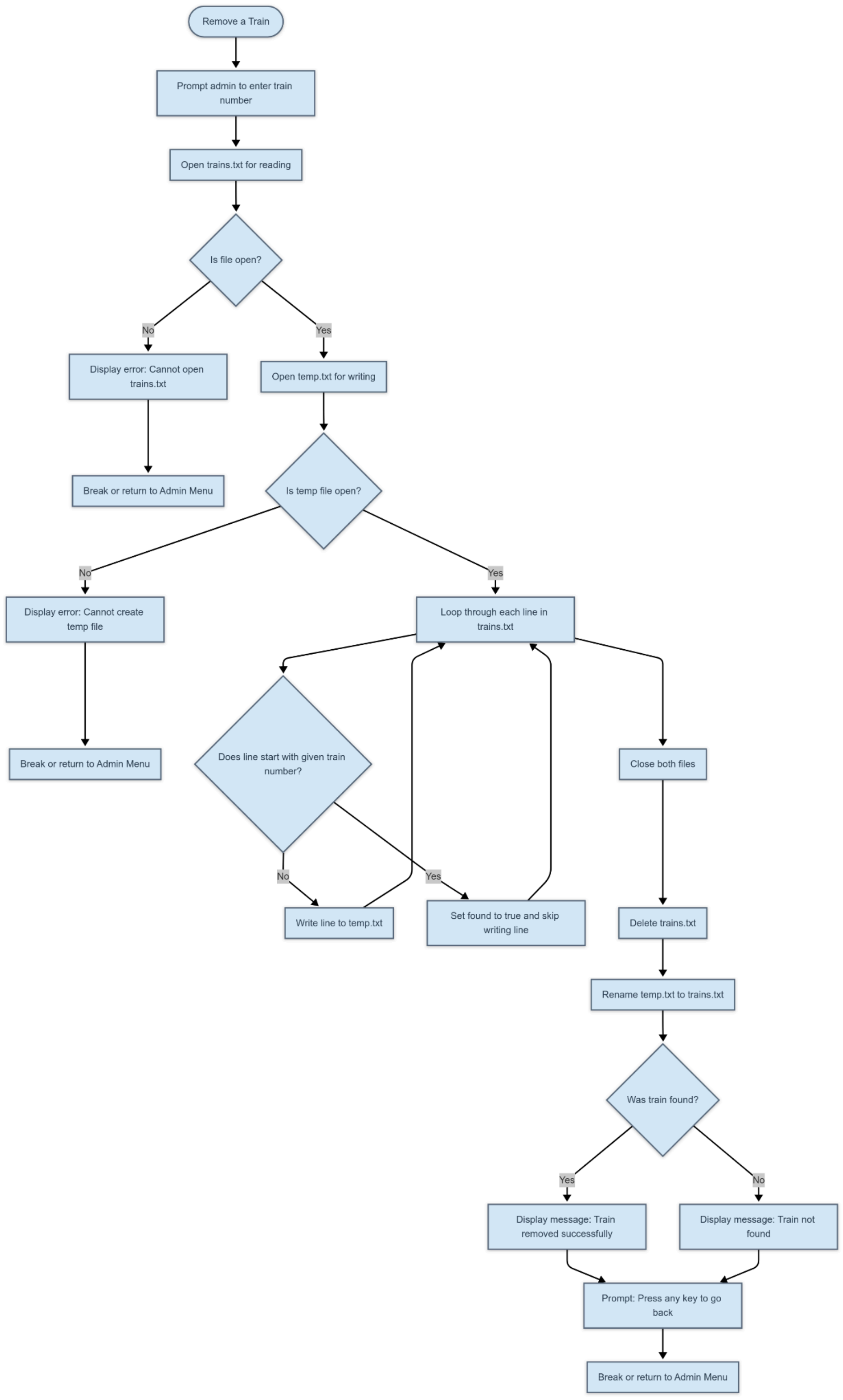


Display Train Database
Open trains.txt for reading
Is file open?
No
Yes
Display error: Cannot open trains.txt
Print train database headings
Break / Return to Admin Menu
Loop through each line in trains.txt
Split line by comma
No
Close file after reading all lines
Are 13 fields present?
Prompt: Press any key to go back
Yes
Format and print train details
Break / Return to Admin Menu

8.12 Add Train(Admin Menu Component)


Add a Train
Open trains.txt in append mode
Is file open?
No
Yes
Display error: Cannot open trains.txt for writing
Break / Return to Admin Menu
Prompt admin to enter train details
Read all train fields: trainNo, name, etc.
Write new train data to file
Close file
Display success message: Train added
Prompt: Press any key to go back
Break / Return to Admin Menu

8.13 Cancel Train(Admin Menu Component)


Remove a Train
Prompt admin to enter train number
Open trains.txt for reading
Is file open?
No
Yes
Display error: Cannot open trains.txt
Open temp.txt for writing
Break or return to Admin Menu
Is temp file open?
No
Yes
Display error: Cannot create temp file
Loop through each line in trains.txt
Break or return to Admin Menu
Does line start with given train number?
Close both files
No
Yes
Write line to temp.txt
Set found to true and skip writing line
Delete trains.txt
Rename temp.txt to trains.txt
Was train found?
Yes
No
Display message: Train removed successfully
Display message: Train not found
Prompt: Press any key to go back
Break or return to Admin Menu

9. An analysis of the role of OOP concepts usage

Here's an overview of the application of various concepts in different files across the project.

9.1 main.cpp

- *OOP Usage:* The main function creates an object of the **Management** class (Management system) and uses try-catch blocks.
- *Applied Concepts:* **Object instantiation** (creating the Management object). **Exception handling** wraps calls to loadDatabases and the Management constructor to catch runtime errors.
- *Why & Benefit:* Initialising a Management object triggers the application's control flow (the constructor sets up menus). Exception handling (try/catch) ensures file I/O or logic errors are caught gracefully, allowing the program to report issues without crashing.

9.2 railway_system.h

- *OOP Usage:* Declares multiple classes: Station, Train, Details, ticket, orderFood, registration, and Management. Also defines global variables.
- *Applied Concepts:*
    1. **Classes & Encapsulation**: Each class bundles related data and methods (e.g. Station has code and name; Train has schedule/pricing). Access specifiers (public) group data with operations.
    2. **Constructors**: Station(string c, string n); and Train(...) constructors are declared here. Constructors ensure that objects initialise their members properly.
    3. **Static Members**: Classes like Details and ticket declare static fields (e.g. static string name[6], static float charges). A static member is shared across all objects of that class.

4. **Inheritance**: The class declaration **class ticket: public Details** shows single inheritance. **ticket** derives from **Details**, inheriting its static fields (passenger details) and methods.

- *Why & Benefit:* Using classes abstracts railway entities (stations, trains, tickets) into objects. This encapsulation keeps data (e.g. train schedules, passenger details) paired with related functions. Inheritance lets ticket reuse Details' fields (PNR, customer arrays) without redefinition. Static members provide shared data across all objects (e.g. a single PNR counter), consistent with the idea that "no matter how many objects... there is only one copy" of a static member.

9.3 station.cpp

- *OOP Usage:* Implements the Station constructor using an initialiser list.
- *Applied Concepts:* **Constructor** and **Encapsulation**. The constructor initialises code and **name** members efficiently.
- *Why & Benefit:* Using the constructor ensures every **Station** object is created with valid code and name values. Initialiser lists are efficient for initialisation. Encapsulation is implied because station data is managed inside the class. This constructor centralises initialisation logic, making it easy to maintain and preventing uninitialized objects.

9.4 train.cpp

- *OOP Usage:* Implements the Train constructor, which includes input validation inside a try-catch.
- *Applied Concepts:* **Constructor**, **Exception Handling**, and **Encapsulation**. The constructor enforces valid data (e.g. no empty fields, non-negative values) and then sets class members.
- *Why & Benefit:* The constructor automates validation and initialisation of a Train object, ensuring it always represents a valid train schedule. Exception handling (throw/catch) provides a robust mechanism to report invalid input instead of silently creating a bad object. Encapsulation is shown by keeping data members private (conceptually) and only initialising them through the constructor. Overall,

this approach prevents misuse (like negative prices or missing data) at object creation.

9.5 ticket.cpp:

- *OOP Usage:* Implements methods of the **ticket** class (which inherits from Details): **Bill()**, **dispBill(), dispDatabase(), deletedata().** It also defines static members: **float ticket::charges; , int ticket::selectedTrainIndex;** and **std::string ticket::selectedClass;.**
- *Applied Concepts:*
    1. **Static Members**: **charges, selectedTrainIndex, selectedClass** are static, meaning all tickets share these across instances. They store the current booking info globally for the session.
    2. **Inheritance**: **ticket** inherits **Details** fields (PNR, passenger info). The methods use inherited static arrays (**Details::name, Details::age**, etc.) when printing or deleting tickets.
    3. **Exception Handling**: Each method uses try/catch to handle file I/O errors (e.g. opening "tickets.txt") and logic errors. For instance, in **Bill()** a runtime error is thrown if the file fails to open, caught and reported.
    4. **Encapsulation & Data Abstraction**: **Ticket** data is read from/written to files. The class methods manage file parsing and output, hiding those details from other parts of the program.
- *Why & Benefit:* Static members allow shared state for the ticket process (e.g., a single charges holds the cost of the chosen class for the current booking). Inheritance avoids duplicating passenger fields; all booking classes access Details data seamlessly. Exception handling makes the system robust: problems like failing to open files or invalid format are caught and reported, preventing crashes. This ensures the program can handle unexpected errors in booking or cancellations gracefully.

9.6 registration.cpp:

- *OOP Usage:* Implements the **registration** class and related classes: a **TrainService** interface (abstract base class) and its implementation **StandardTrainService**. Also defines helper functions and static members: **static int registration::choice;. Key methods: selectBoardingStation(),** and **trains()**.
- *Applied Concepts*:
    1. **Static Members: registration::choice** is static (shared across all registration objects).
    2. **Inheritance & Polymorphism: TrainService** is an abstract class with pure virtual methods **showAvailableTrains** and **selectTrain**. **StandardTrainService** inherits from **TrainService** and implements these methods. In **registration::trains()**, a pointer of type **TrainService*** is assigned to an instance of **StandardTrainService**. Methods are called via this base-class pointer. This is **runtime polymorphism**: the code calls service->showAvailableTrains(...) and service->selectTrain(...) without knowing the exact derived type at compile time.
    3. **Exception Handling:** The **trains()** method uses try/catch to handle any unexpected errors during booking (e.g. no trains found).
    4. **Encapsulation:** Class registration encapsulates the logic for selecting stations and booking trains. It uses the static **seatAvailability** map (from railway_system.h) and modifies **ticket::charges** and **ticket::selectedClass**.
- *Why & Benefit:* The **abstract base class** TrainService defines an interface for showing and selecting trains. This lets the code rely on a general interface, making it easy to extend (e.g. adding new service types) without changing registration. The **polymorphism** here means a single line **TrainService*** service = &trainService; can later point to any class that implements that interface, making the design flexible. Using static members (like **registration::choice**) keeps shared state when needed. Exception handling again ensures that unexpected conditions (e.g. file errors) do not crash the booking process. Overall, this setup cleanly separates concerns: registration orchestrates the process, while **StandardTrainService** handles the details of listing and selecting trains.

9.7 management.cpp:

- *OOP Usage:* Implements the **Management** class, which contains two main menu loops (**mainMenu()** for admin, **mainMenu2()** for users) and a **firstPage()** function. It uses objects of other classes (composition): Details d; registration r; ticket t; orderFood f;.
- *Applied Concepts*:
    1. **Composition**: Management contains (uses) instances of other classes (Details, registration, ticket, orderFood). Though not member fields, these objects are created on the stack in methods to utilise their functionality. This is aggregation – Management orchestrates these sub-objects to perform tasks.
    2. **Static Members**: Uses **static bool trainBooked** and **static bool customerDetailsAdded** within **mainMenu2()** to keep track of application state across loop iterations.
    3. **Exception Handling**: Input is validated using **getIntInput**, which itself loops until valid input is provided. The menus rely on exception-safe I/O to avoid crashes on invalid user input.
    4. Forward declarations (**class Details; class registration; class ticket; class orderFood;**) show that these classes are used.
- *Why & Benefit:* The Management class ties all components together, serving as the program's controller. Composition (using objects of other classes) keeps related functionality modular. For instance, **t.Bill()** and **f.orderFoodItems()** are called to handle booking and food orders. Using static flags ensures user operations occur in the correct order (e.g. can't add customer details before booking a train). This class hides the menu logic from other classes, focusing on flow control. Aggregation/composition here improves maintainability by delegating tasks to specialised classes.

9.8 database.cpp:

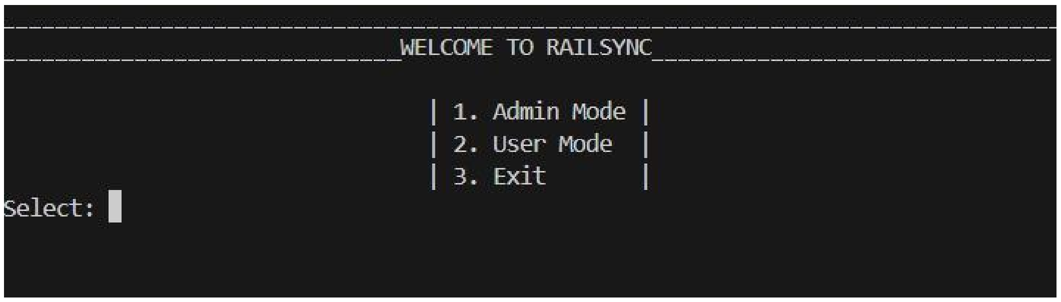
- *OOP Usage:* Implements free functions and global data: **vector<Station> stations; vector<Train> allTrains; map<string,map<string,int>> seatAvailability**; etc. Defines **initializeSeatAvailability, loadDatabases, and displayStationsMenu**.
- *Applied Concepts*:
    1. **Abstraction & Encapsulation:** Data is loaded from text files into **Station** and **Train** objects. For example, loadDatabases reads "stations.txt", creating Station objects via its constructor and pushing them into **stationsList**. Similarly for trains, it constructs Train objects with parsed data. This abstracts the file format into usable objects.
    2. **Encapsulation:** Each class's constructor handles the details of initializing objects from raw strings. The code in **loadDatabases** uses these constructors to encapsulate data validation and assignment.
- *Why & Benefit:* By encapsulating station/train data in classes, the code cleanly separates file I/O from program logic. Functions like displayStationsMenu take a const vector<Station>& and show them to the user, hiding I/O details from the rest of the system. The use of objects here helps prevent errors: for instance, pushing a properly constructed Station ensures it always has a code and name. Initialising seat availability from the Train objects centralises the tracking of seats, making it easy to update availability when booking or cancelling.

9.9 order_food.cpp:

- *OOP Usage:* Implements orderFood class. Defines a private menu map of food items and public methods: displayFoodDatabase(), orderFoodItems(int pnr), viewFoodOrder(int pnr). Includes a helper pnrExists(int pnr).
- *Applied Concepts*:
    1. **Encapsulation**: The food menu is a private member (initialized in the constructor), hidden from other parts of the code. All interactions with the menu happen through public methods. This bundling of data (menu) with operations (ordering/viewing) is classic encapsulation.
    2. **Constructor**: orderFood::orderFood() initializes the menu. This sets up the object's initial state (food options).
    3. **Static & Global Data**: It uses the global foodOrders map (keyed by (userID, PNR)), but manages adding to it.
    4. **Exception Handling**: The pnrExists method checks for valid PNR, printing an error message if not found (no try/catch needed here, but input is checked).
- *Why & Benefit:* Encapsulating the food menu in a class means all related logic (display, ordering, viewing) stays together. The constructor ensures the menu is set up before use. Checking the PNR before placing orders prevents invalid orders. This class makes it easy to modify the menu or ordering process in one place, demonstrating modularity.

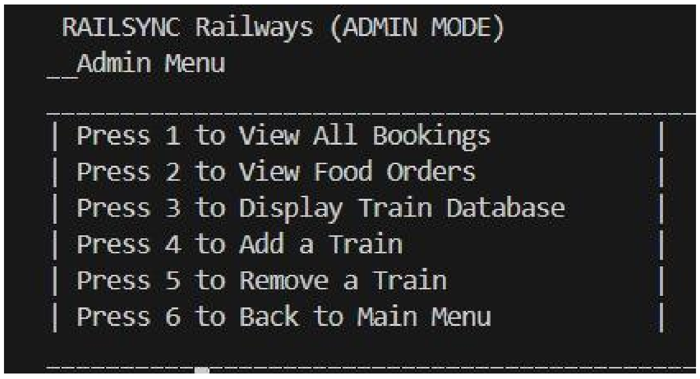
10. Snapshot samples of the output

Welcome Page:



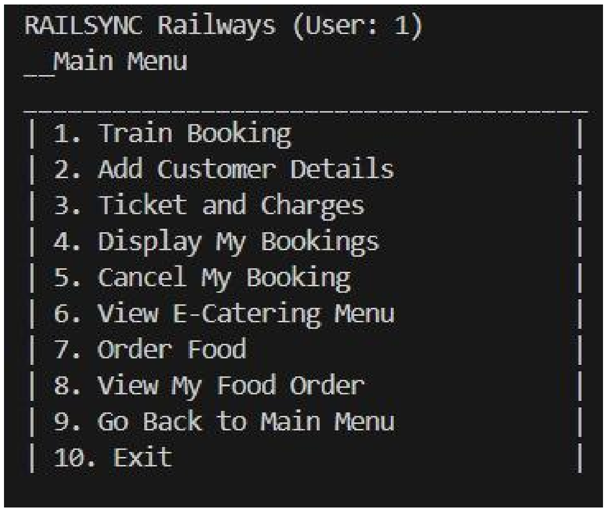
```
__________________________________________________________________________________________
___________________________WELCOME TO RAILSYNC___________________________

                                | 1. Admin Mode |
                                | 2. User Mode  |
                                | 3. Exit       |
Select:
```

Admin Menu:



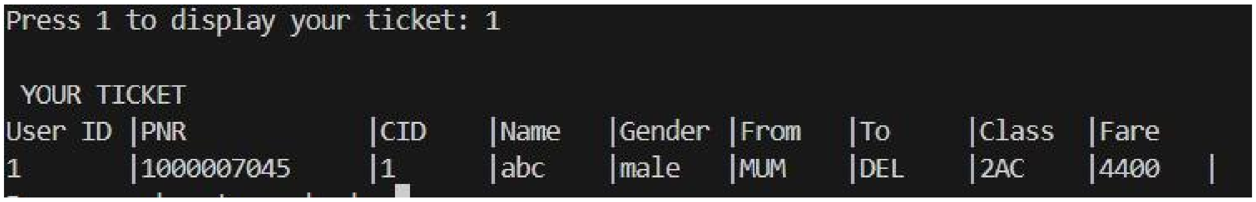
```
    RAILSYNC Railways (ADMIN MODE)
   __Admin Menu
   ____________________________________________
   | Press 1 to View All Bookings            |
   | Press 2 to View Food Orders             |
   | Press 3 to Display Train Database       |
   | Press 4 to Add a Train                  |
   | Press 5 to Remove a Train               |
   | Press 6 to Back to Main Menu            |
   ____________________________________________
```

User Menu:

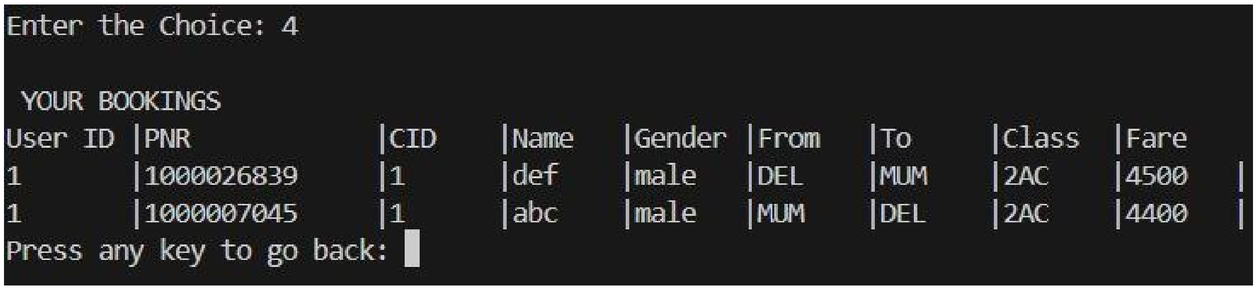


```
 RAILSYNC Railways (User: 1)
 __Main Menu
 ______________________________________
 | 1. Train Booking                   |
 | 2. Add Customer Details            |
 | 3. Ticket and Charges              |
 | 4. Display My Bookings             |
 | 5. Cancel My Booking               |
 | 6. View E-Catering Menu            |
 | 7. Order Food                      |
 | 8. View My Food Order              |
 | 9. Go Back to Main Menu            |
 | 10. Exit                           |
```

Viewing Current Ticket:


```
Press 1 to display your ticket: 1

 YOUR TICKET
User ID |PNR            |CID     |Name    |Gender |From    |To      |Class   |Fare
1       |1000007045     |1       |abc     |male   |MUM     |DEL     |2AC     |4400    |
```

Viewing All Bookings:


```
Enter the Choice: 4

 YOUR BOOKINGS
User ID |PNR            |CID     |Name    |Gender |From    |To      |Class   |Fare
1       |1000026839     |1       |def     |male   |DEL     |MUM     |2AC     |4500    |
1       |1000007045     |1       |abc     |male   |MUM     |DEL     |2AC     |4400    |
Press any key to go back:
```

11. Conclusion

The Railway Management System project successfully demonstrates the application of core Object-Oriented Programming (OOP) concepts such as classes, inheritance, polymorphism, encapsulation, and file handling in C++. Through the development of this system, we designed a functional model capable of handling real-world tasks like train bookings, ticket generation, food ordering, and administrative operations in a modular and efficient manner.

The project emphasises a user-friendly interface for passengers while ensuring that the admin has robust controls to manage trains, bookings, and food services. By dividing functionalities into different modules and using file-based data storage, we have achieved a scalable and maintainable codebase. In addition, the integration of an e-catering module adds value to the user experience, reflecting real-world enhancements seen in modern railway systems.

Overall, the Railway Management System project not only strengthened the understanding of theoretical OOP concepts but also provided practical experience in building a complete application from scratch. Future improvements could include introducing real-time database management, security features like user authentication, and a graphical user interface (GUI) to make the system even more intuitive and powerful.

12. References

- C++ Documentation: https://en.cppreference.com/w/
- GeeksforGeeks Articles:
    1. https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/
    2. https://www.geeksforgeeks.org/c-classes-and-objects/
    3. https://www.geeksforgeeks.org/access-modifiers-in-c/
    4. https://www.geeksforgeeks.org/constructors-c/
    5. https://www.geeksforgeeks.org/destructors-c/
    6. https://www.geeksforgeeks.org/encapsulation-in-cpp/
    7. https://www.geeksforgeeks.org/abstraction-in-cpp/
    8. https://www.geeksforgeeks.org/cpp-polymorphism/
    9. https://www.geeksforgeeks.org/inheritance-in-c/
    10. https://www.geeksforgeeks.org/exception-handling-c/
- OpenAI tools