**Dream Flights**



Session: 2023 – 2027

**Submitted by:**

Muhammad Ahmed Butt 2022-CS-18

**Supervised by:**

Dr. Prof Awais Hassan

Department of Computer Science

**University of Engineering and Technology**

**Lahore Pakistan**

**Table of Contents**

[**1.** **INTRODUCTION:** 4](#_Toc141545075)

[**1.1** **Overview:** 4](#_Toc141545076)

[**1.2** **Objectives:** 4](#_Toc141545077)

[**1.3** **Intended Functionality:** 5](#_Toc141545078)

[**2.** **OOP-CONCEPTS:** 5](#_Toc141545079)

[**2.2** **Association:** 5](#_Toc141545080)

[**2.3** **Inheritance:** 6](#_Toc141545082)

[**2.4** **Polymorphism:** 6](#_Toc141545083)

[**3.** **COMPARE WITH PROCEDURAL PROGRAMING:** 6](#_Toc141545084)

[**3.1** **Reusability and Code Sharing:** 6](#_Toc141545085)

[**3.2** **Maintainability and Scalability:** 7](#_Toc141545086)

[**3.3** **Code Readability and Understandability:** 7](#_Toc141545087)

[**3.4** **Data Security and Abstraction:** 7](#_Toc141545088)

[**4.** **DESIGN PATTERN IMPLEMENTATION:** 7](#_Toc141545089)

[**4.1** **Business Logic (BL) Design Pattern:** 8](#_Toc141545090)

[**4.2** **Data Access Layer (DL) Design Pattern:** 8](#_Toc141545091)

[**4.3** **User Interface (UI) Design Pattern:** 8](#_Toc141545092)

[**5.** **CLASSES DETAILS:** 9](#_Toc141545093)

[**5.1** **Business Logic (BL) Classes** 9](#_Toc141545094)

[**5.2** **Data Access Layer (DL) Classes:** 12](#_Toc141545095)

[**6.** **Wireframes:** 20](#_Toc141545097)

[**7.** **COMPLETE CODE:** 34](#_Toc141545102)

[**8.** **CONCLUSION:** 75](#_Toc141545106)

# **INTRODUCTION:**

# **Overview:**

**Problem Statement:** The airline industry is a complex ecosystem that requires efficient management of various operations to ensure smooth functioning and exceptional customer service. It requires an airline management system that simplifies these requirements

**Solution:** The dream flights aims to address the problem by utilizing technology to booked flights more easily. It provides an organized system that manages flights and provides an easy interface for its users to book or manage flights

**Significance:** The development and implementation of a Dream Flights system can have a positive impact in the field of airline managing system. An app for managing airline stuff, like flights and schedules, can be super helpful. It tells passengers and airline workers what's happening with flights in real-time, like if there are delays or gate changes. Passengers can use it to check in on their phones, pick seats, and even change their bookings quickly. For airline staff, it makes things smoother too,

# **Objectives:**

**For ADMIN**

**Easy Addition of Flight:** Adding a flight on the app is as easy as a few clicks just enter the flight details like date, time, and destination, and you're all set to manage it effortlessly..

**Can Cancel Flight:** Removing a flight from the app is simple – just select the flight you want to remove and confirm your choice..

**Can Update Flight:** Simply edit the flight details with the new information and save the changes. Stay flexible with your travel plans by easily updating flight details anytime, anywhere, right from your fingertips.

**View Flights:** where you'll find a comprehensive view of all available flights, including their departure and arrival times, destinations,

**Add Employee** Adding an employee on the app is straightforward – fill in their details like name, id, designation and salary and they're ready to be part of your team

**Remove Employee:** Removing a employee from the app is simple – just select the Employee you want to remove and confirm your choice..

**For User**

**Search Flight:** Just search for the flight using its number or departure/arrival details, and all the information will be right at your fingertips.

**Book A Flight:** Just enter your travel details like destination, dates etc, and preferred times, then choose from the available flight options.

**Give Feed Back:** Providing feedback on the app is quick and easy – simply navigate to the "Feedback" section, where you can share your thoughts, suggestions, or any issues you encountered while using the app.

**Cancel Flight:** Just locate your booked flight in the "My Bookings" section, select the option to cancel, and confirm your decision.

# **Intended Functionality:**

**User-Friendly Interface:** The blood donation management system will have a user-friendly interface that allows individuals to easily search for specific blood groups based on their requirements. The interface will be intuitive, responsive, and accessible to a wide range of users.

**Real-Time Availability:** The system will show you right away which flights are available. This helps you quickly find the flight you need, especially in emergencies or when plans change suddenly.

**Flight Management:**

* Tracking of flight schedules and availability.
* Management of passenger preferences.
* Booking and reservation of flights.

# **OOP-CONCEPTS:**

In the context of flight management systems, object- oriented programming (OOP) principles can be utilized to improve system design and functionality. Here are some examples:

# **Association:**

Association is of two types of aggregation and composition; Aggregation is applied in the management system. Aggregation is present between different classes like all classes of UI to classes of BL and DL. Also, there is aggregation between Flight class with Client class Client contains list of Flight

# **Inheritance:**

Inheritance is another key concept in OOP that allows the creation of hierarchical relationships between classes. In the Dream Flights , inheritance can be utilized to establish relationships between several types of users to a single person class. For example, a Client class could inherit from a more general Person class, inheriting its properties and methods while adding specific characteristics or behaviors. Also, there is inheritance between Admin and Person

# **Polymorphism:**

Polymorphism enables objects of different classes to be treated interchangeably, allowing for flexibility and extensibility in the system. The polymorphic behavior occurs when you treat instances of subclasses as instances of the abstract class AdminDL. For example, if you have a method that accepts AdminDL objects, you can pass instances of subclasses to that method, and it will work because of polymorphism. The method can then call the overridden methods specific to the subclass, allowing different behavior depending on the actual runtime type of the object being used.

# **COMPARE WITH PROCEDURAL PROGRAMING:**

Following are the various aspects in which OOP is better than procedural programing:

# **Reusability and Code Sharing:**

# **COMPARE WITH PROCEDURAL PROGRAMING:**

Following are the various aspects in which OOP is better than procedural programing:

# **Reusability and Code Sharing:**

* OOP promotes code reusability through the concept of inheritance. Inheritance allows classes to inherit properties and behaviors from other classes, reducing the need to rewrite code. This not only saves development time but also makes the codebase more efficient and easier to maintain.
* In procedural programming, reusing code requires copying and pasting or creating separate functions, which can lead to code redundancy and maintenance issues.

# **Maintainability and Scalability:**

* OOP: Changes or updates to specific functionalities, such as adding a new type of aircraft or introducing a new passenger class, can be localized within the relevant classes. For instance, extending the Flight class to accommodate new aircraft types is easier through inheritance and polymorphism.
* Procedural: Making changes or adding features typically involves modifying multiple functions scattered throughout the codebase, which can be error-prone and challenging to maintain as the system grows.

# **Code Readability and Understandability:**

* OOP: Representing real-world entities as objects (e.g., flights, passengers, airports) and their interactions through methods and properties makes the code more readable and understandable. For instance, methods like bookSeat() and checkAvailability() in the Flight class represent actions related to flights.
* Procedural: Without the conceptual organization provided by classes and objects, procedural code can become more convoluted and harder to comprehend, especially as the system complexity increases.

# **Data Security and Abstraction:**

* OOP: Encapsulation allows for better data security by hiding the internal implementation details of classes and providing controlled access to data through methods. For instance, only specific methods within the Flight class may be allowed to modify booking information, ensuring data integrity.
* Procedural: Procedural programming typically lacks built-in mechanisms for data security and abstraction, making it more prone to data integrity issues and unauthorized access to sensitive data.

# **Maintainability and Scalability:**

* OOP promotes code maintainability and scalability. With encapsulation and modularity, making changes to a specific functionality or fixing issues becomes easier because the affected code is localized within the relevant class. Additionally, OOP's ability to extend existing classes through inheritance allows for the addition of new features without modifying the existing codebase extensively.
* In procedural programming, making changes or adding features often involves modifying multiple functions, increasing the likelihood of introducing errors and making maintenance more challenging.

# **Code Readability and Understandability:**

* OOP encourages a more natural representation of real-world entities and their relationships. This makes the code more readable and understandable, as classes and objects closely resemble the entities and interactions they model.
* Procedural programming, on the other hand, can lack this intuitive representation, making it harder to grasp the overall system design and the relationships between various parts of the code.

# **DESIGN PATTERN IMPLEMENTATION:**

In the blood donation management system, the utilization of design patterns helps ensure modularity, separation of concerns, and maintainability. Here is how the project incorporates the Business Logic (BL), Data Access Layer (DL), and User Interface (UI) design patterns:

# **Business Logic (BL) Design Pattern:**

In the flight management system, the Business Logic (BL) design pattern can be implemented using the Model-View-Controller (MVC) architectural pattern to encapsulate the core functionality and operations of the system. Here's how it can be applied:Model: The model represents the business logic and operations of the flight management system. It encapsulates functionalities such as managing flights, passengers, airports, bookings, and other relevant entities. For example, the model would include classes and methods for adding new flights, updating flight schedules, managing passenger reservations, and handling payment transactions.

# **Data Access Layer (DL) Design Pattern:**

In the flight management system, the Data Layer (DL) design pattern, implemented through the Repository pattern for databases and dedicated components for file handling, ensures smooth interaction between the application's logic and data storage.

# **User Interface (UI) Design Pattern:**

* In a WinForms application, the user sees buttons, forms, and menus on the screen, clicking or typing to interact. Developers arrange these elements with a design tool or code. When users do something, like click a button, the app reacts, like showing a message.
* In a console app, everything happens with text. Users type commands, and the app shows text responses. It's simpler than Win Forms but works well for tasks like managing files or running scripts.Both ways let users interact, but Win Forms is graphical, and console apps are text-based.

# **CLASSES DETAILS:**

The project consist of following classes structure in its DLL:

**Bussiness Layer**

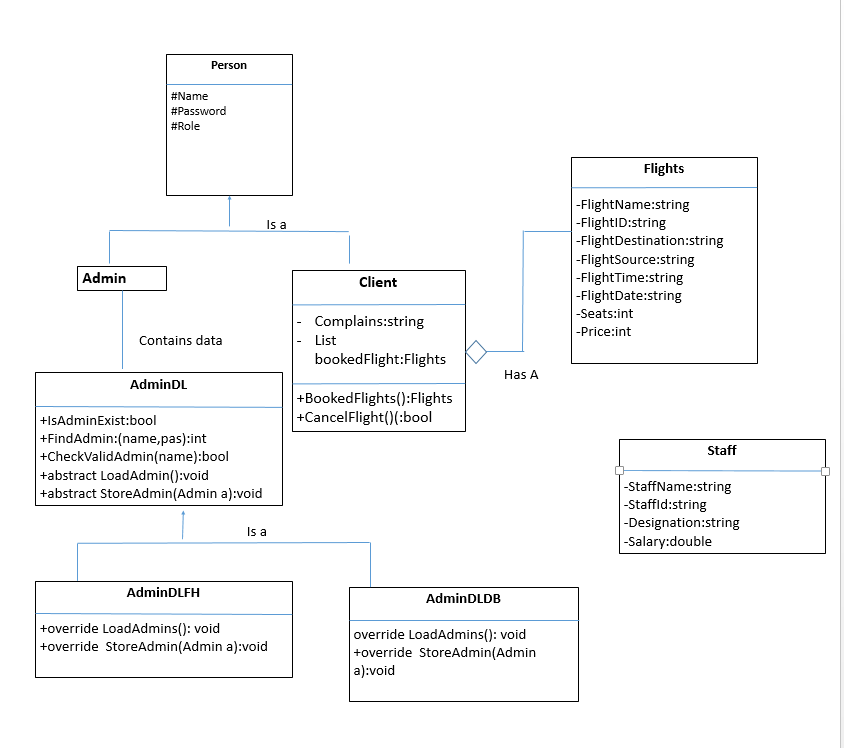
* Admin
* Client
* Staff
* Person
* Flight

**Data Layer**

* AdminDLDB AdminDLFH
* ClientDLDB ClientDLFH
* StaffDLDB StaffDLFH
* FlightDLDB FlightDLFH

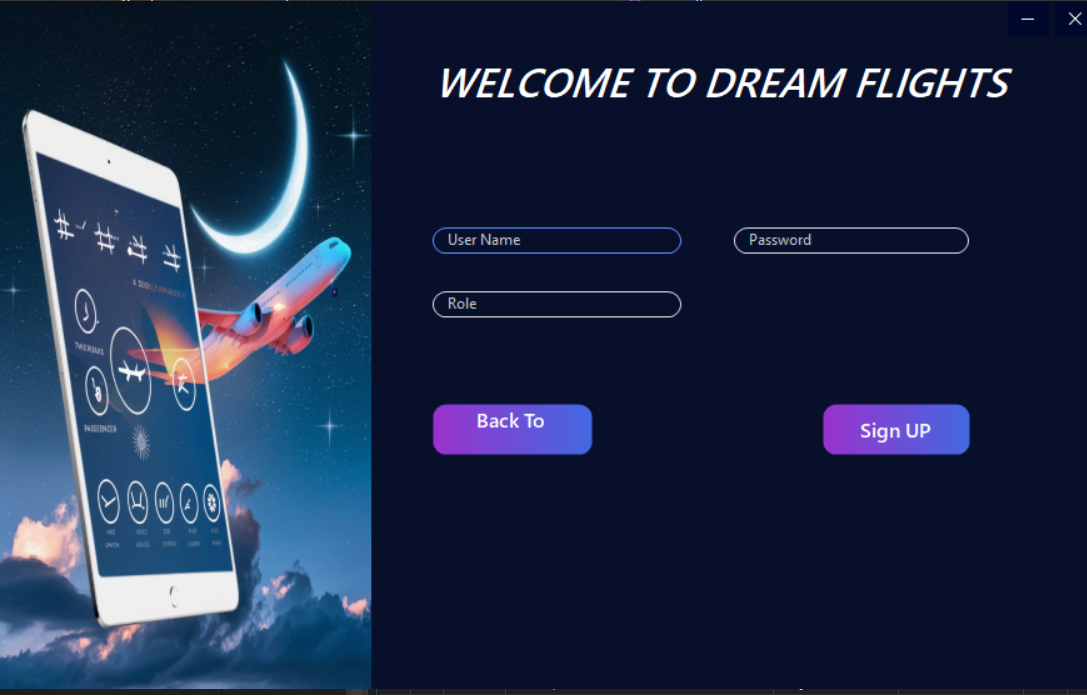
**Abstract DL**

* Admin
* Staff
* Flight
* Client

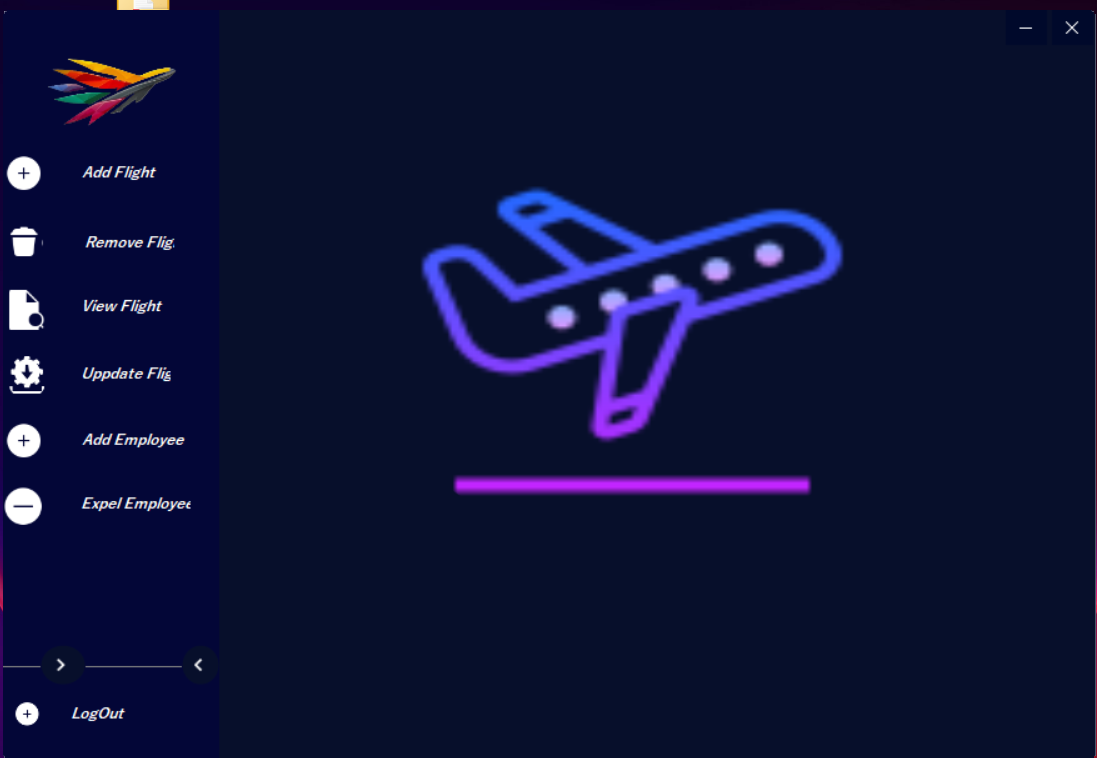
**CRC MODEL**

# **Wireframes:**

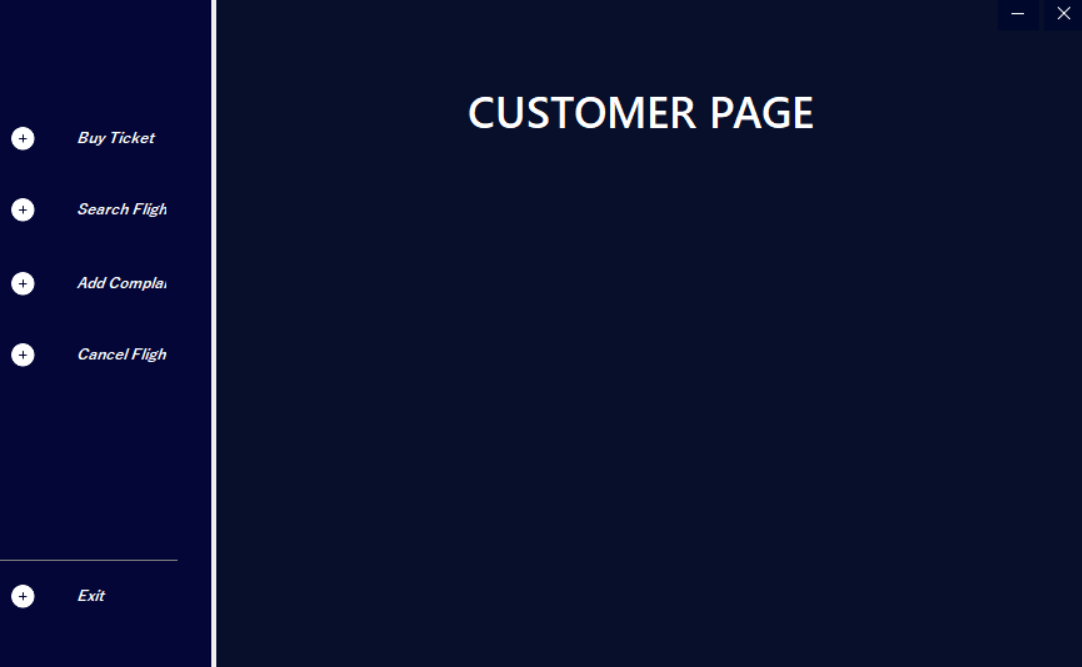
Login Sign Up Page



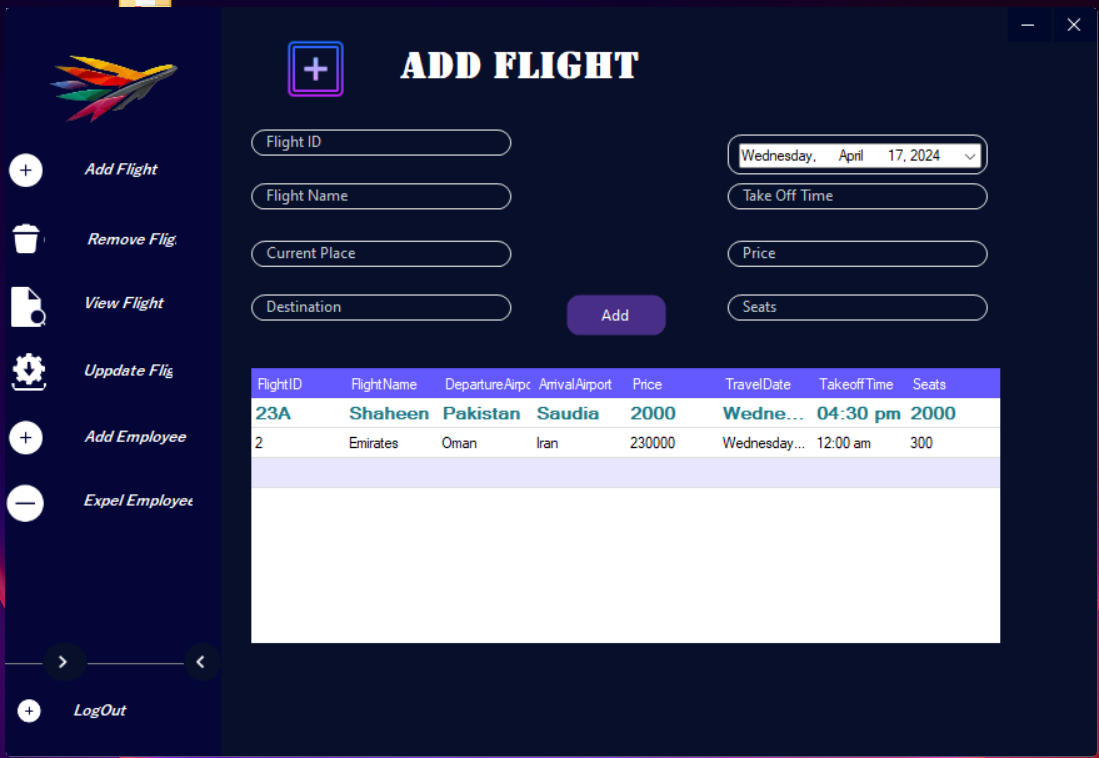
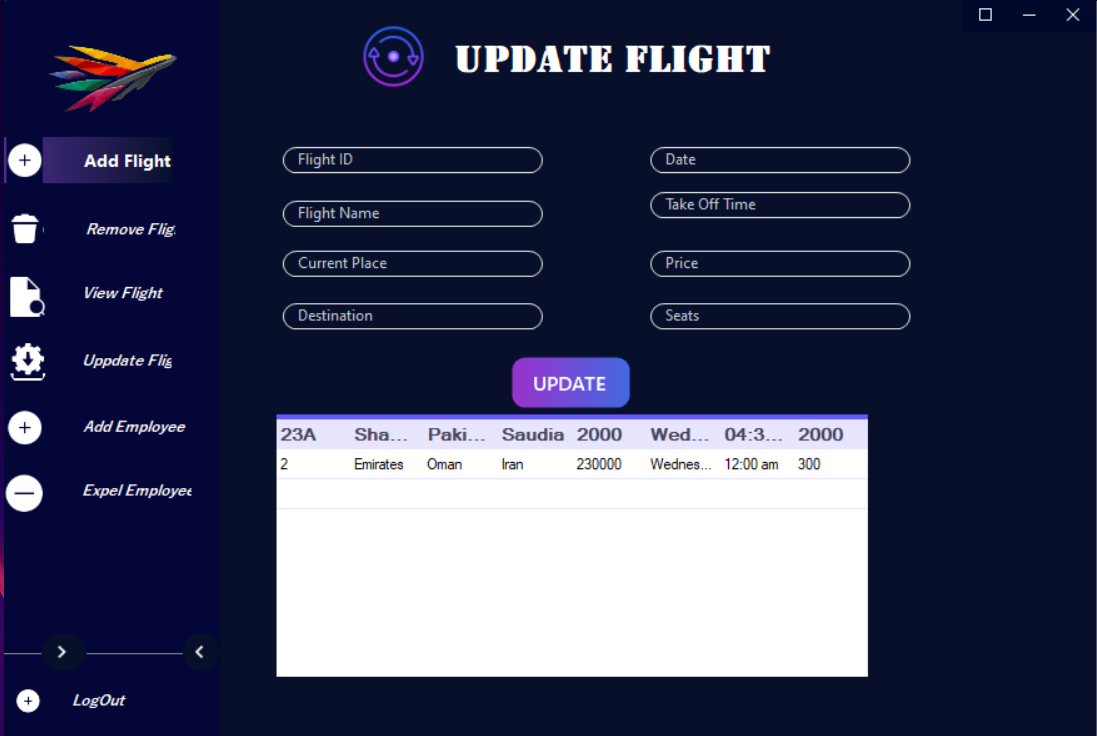
Admin Page

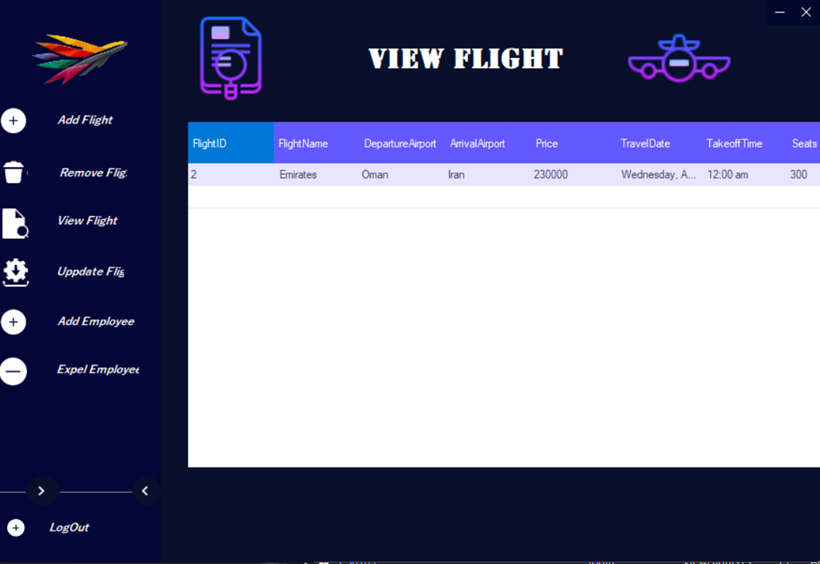


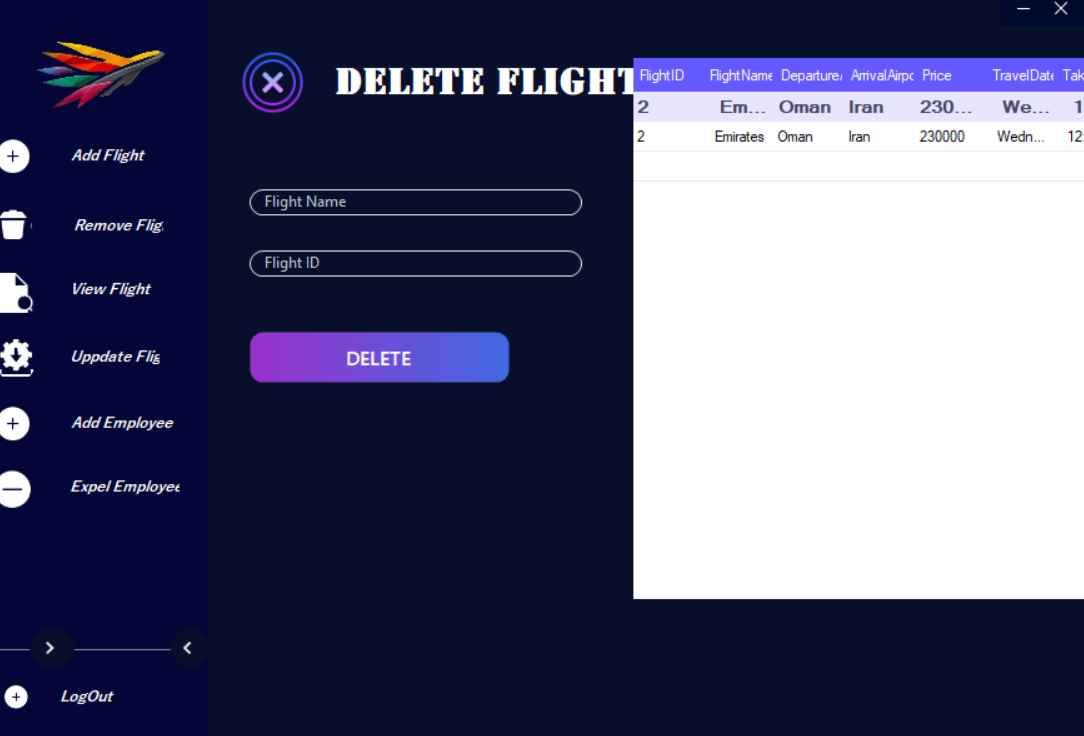
Client Page

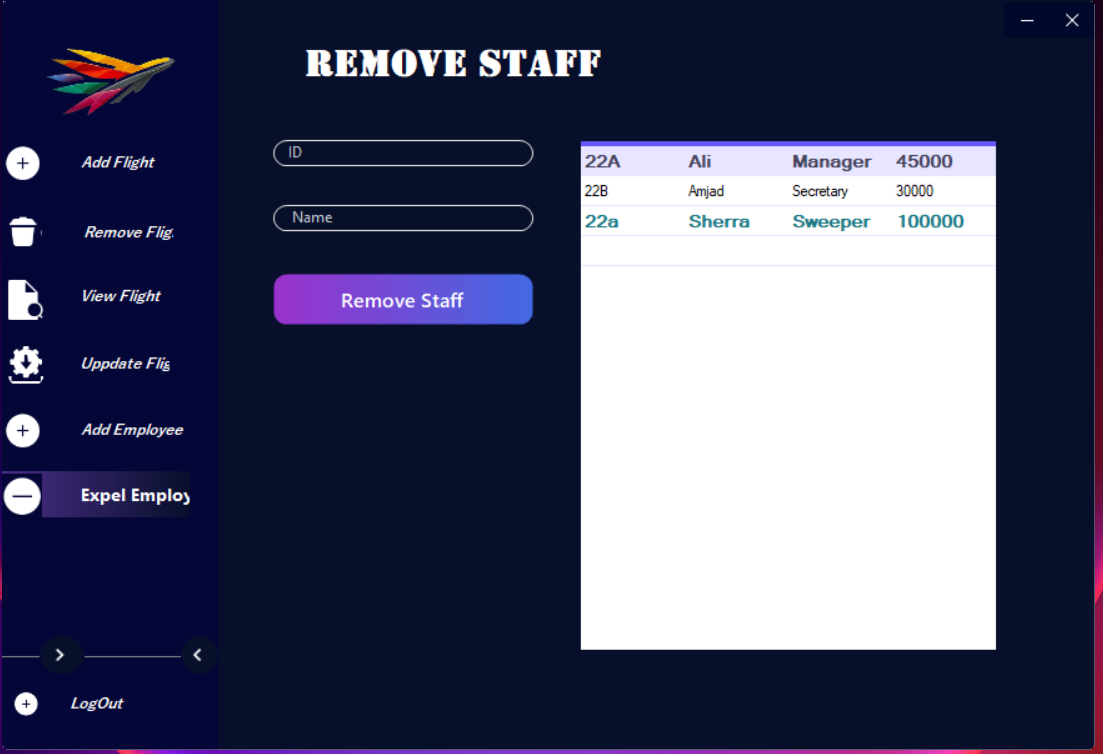


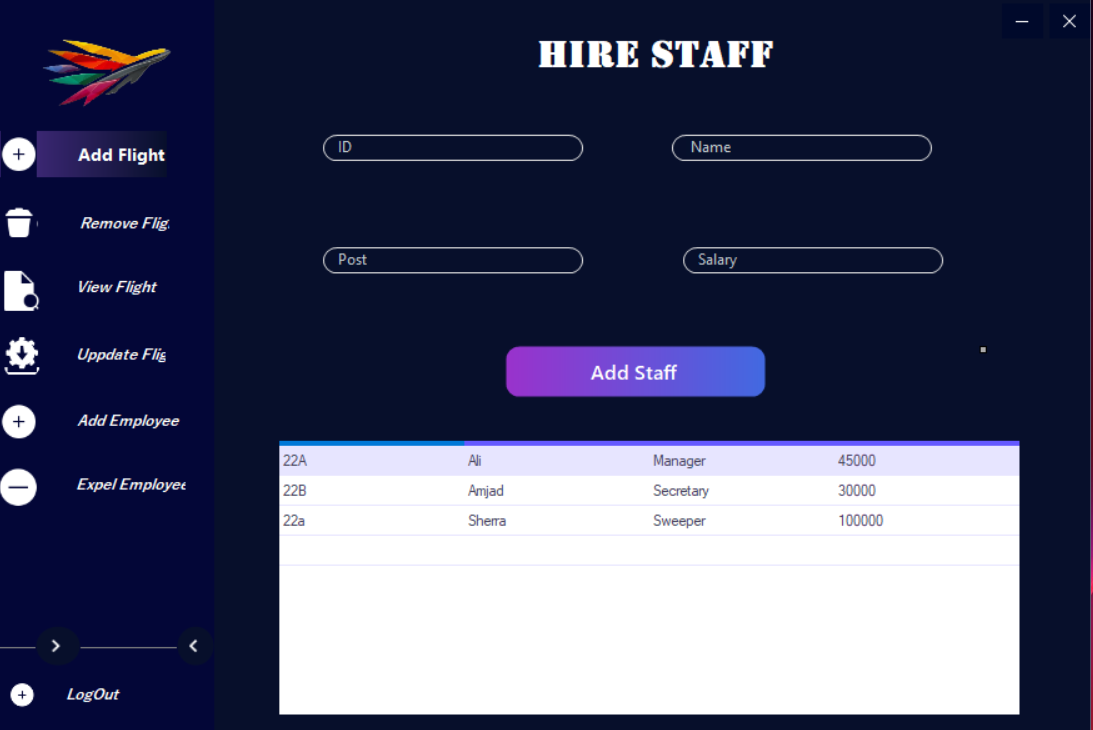
Add Flight

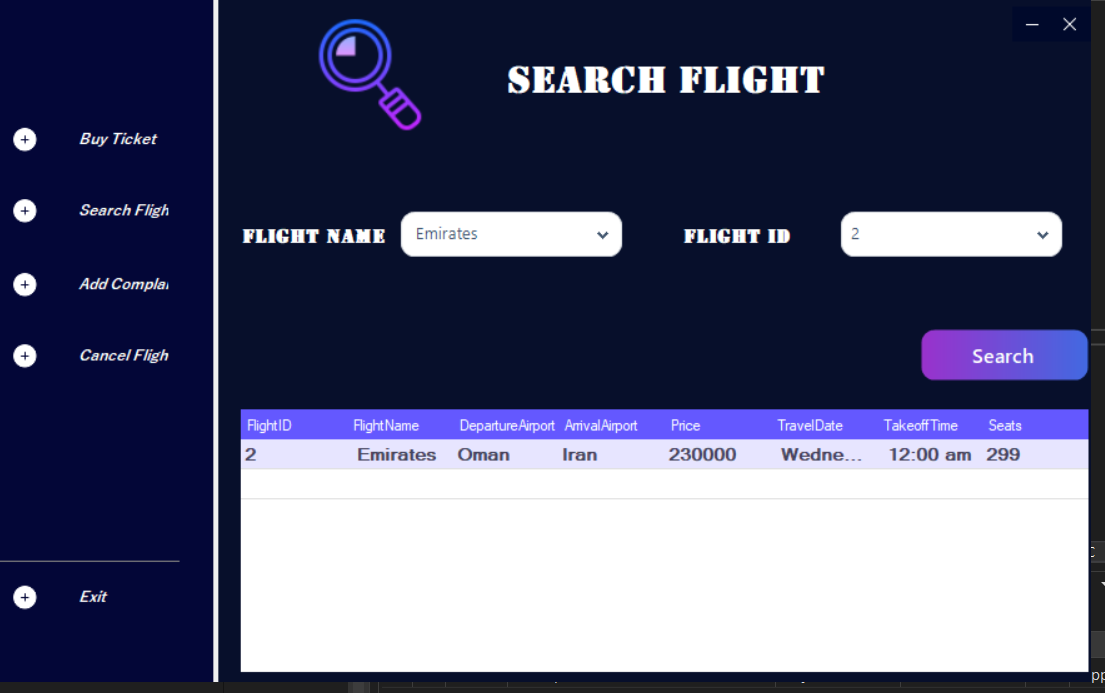


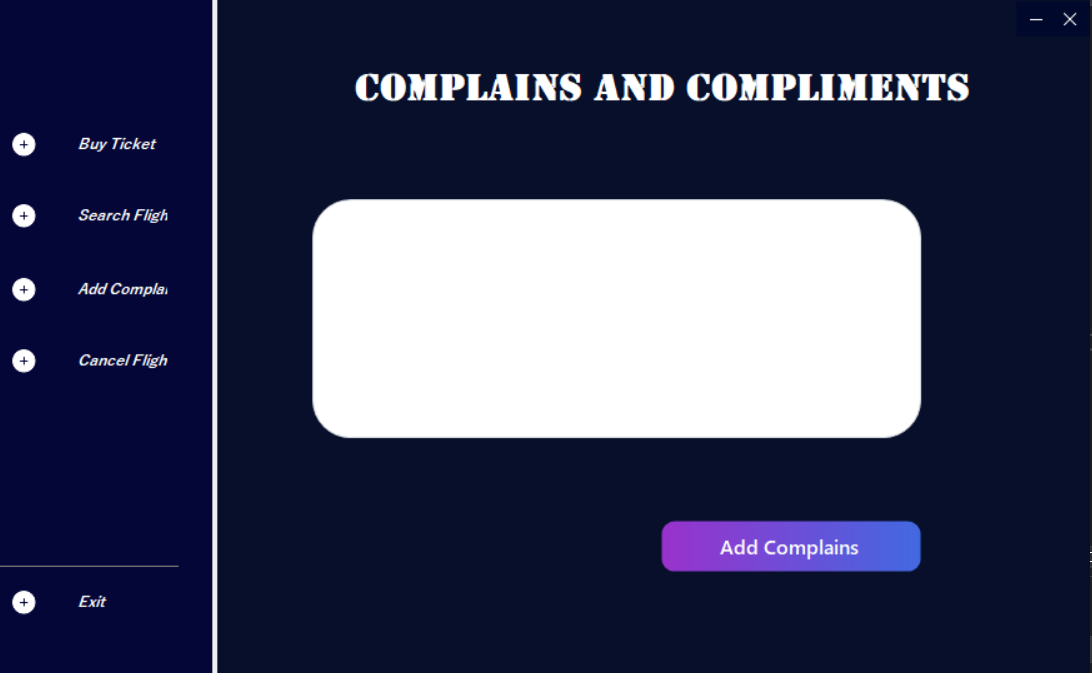


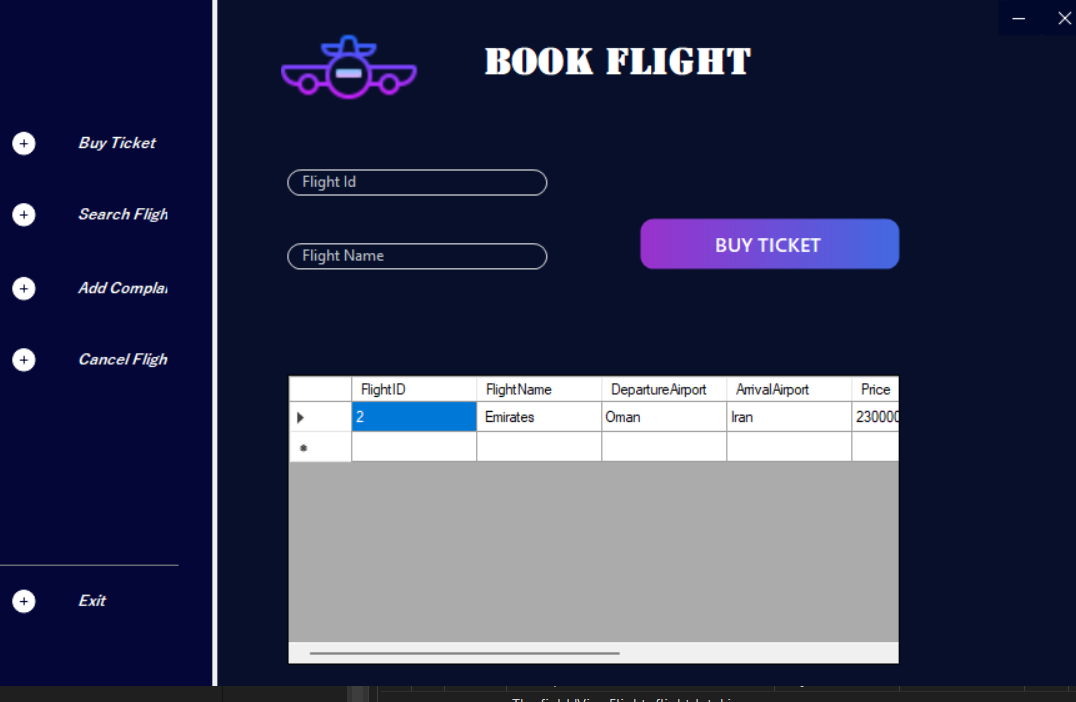


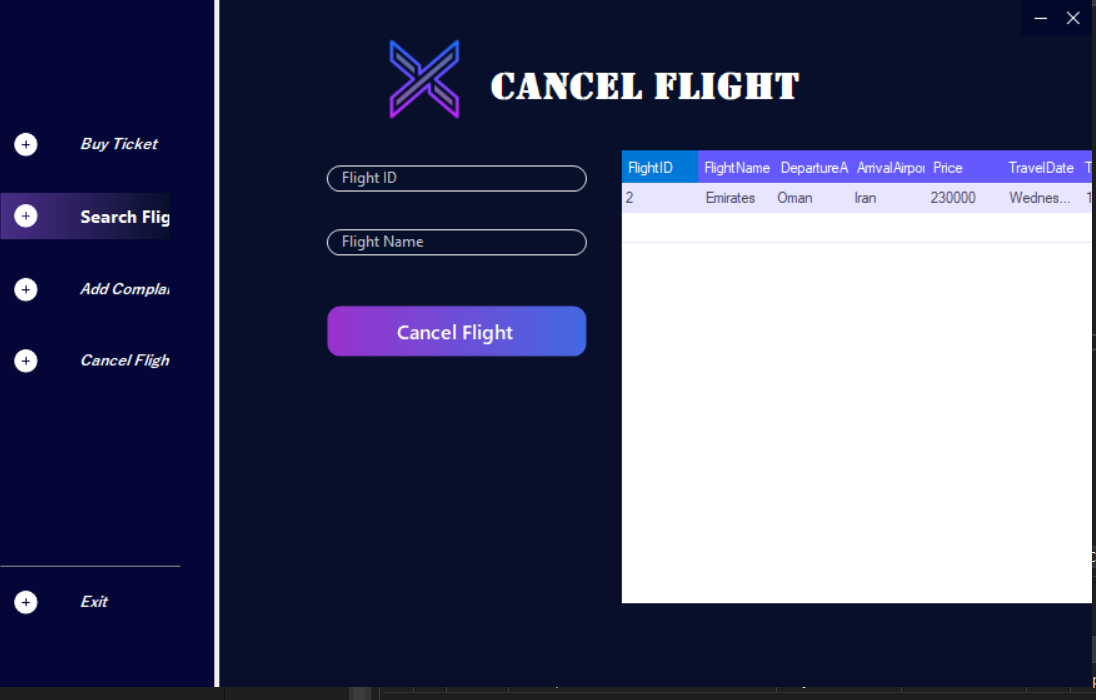












# **CONCLUSION:**

# **7.1 Lesson Learned:**

Throughout the project, several valuable lessons were learned. It became evident that effective planning and requirement gathering are crucial for the successful development of such a complex system. Modularity and separation of concerns played a vital role in achieving maintainability and flexibility. The implementation of design patterns, such as the BL, DL, and UI patterns, helped in achieving a structured and organized codebase. Furthermore, continuous testing and quality assurance were imperative to identify and resolve any issues early on.

**BL CODE:**

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Xml.Linq;

namespace NewLibrary;

{

// The Client class inherits from the Person class, demonstrating the OOP concept of inheritance.

public class Client : Person

{

private string FeedBack;

private List<Flight> BookedFlights;

public Client(string name, string password, string role) : base(name, password, role)

{

BookedFlights = new List<Flight>();

}

public void SubmitFeedBack(string feedback)

{

this.FeedBack = feedback;

}

public string ViewFeedBack()

{

return ($" FeedBack By {Name}: {FeedBack}");

}

public bool BookFlight(Flight f)

{

for (int i = 0; i < BookedFlights.Count; i++)

{

if (BookedFlights[i].GetFlightID() == f.GetFlightID())

{

return false;

}

}

double Seats = f.GetSeats() - 1;

f.SetSeats(Seats);

BookedFlights.Add(f);

return true;

}

public Flight CancelFlight(string ID)

{

for (int i = 0; i < BookedFlights.Count; i++)

{

if (BookedFlights[i].GetFlightID() == ID)

{

double Seats = BookedFlights[i].GetSeats() + 1;

BookedFlights[i].SetSeats(Seats);

Flight f = BookedFlights[i];

BookedFlights.RemoveAt(i);

return f;

}

}

return null;

}

public string GetFeedBack()

{

return FeedBack;

}

public void SetFeedBack(string FeedBack)

{

this.FeedBack = FeedBack;

}

public List<Flight> GetBookedFlights()

{

return BookedFlights;

}

public void SetBookedFlights(List<Flight> BookedFlights)

{

this.BookedFlights = BookedFlights;

}

}

}

**DL Abstract Code :**

using NewLibrary;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace NewLibrary.AbstractClasses

{

public abstract class ClientDL

{

protected static List<Client> Clients = new List<Client>();

public bool IsClientExist(string name, string password)

{

foreach (Client client in Clients)

{

if (client.GetName() == name && client.GetPassword() == password)

{

return true;

}

}

return false;

}

public int FindClient(string name, string password)

{

int ClientNo = 10000;

for (int i = 0; i < Clients.Count; i++)

{

if (Clients[i].GetName() == name && Clients[i].GetPassword() == password)

{

ClientNo = i;

return ClientNo;

}

}

return ClientNo;

}

public List<Client> GetAllClients()

{

return Clients;

}

public bool CheckValidClientName(string name)

{

for (int i = 0; i < Clients.Count; i++)

{

if (Clients[i].GetName() == name)

{

return false;

}

}

return true;

}

public abstract void LoadClients();

public abstract void StoreClients(Client client);

public abstract void UpdateFeedBack(string Name, string feedback);

public abstract List<Flight> ReturnReservedFlights(string ClientName);

public abstract void StoreBookedFlights(string FlightID, string Name);

public abstract void DeleteBookedFlights(string flightID, string Name);

}

}

**DL DataBase**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Data;

using System.Data.SqlClient;

using NewLibrary.AbstractClasses;

namespace NewLibrary

{

public class ClientDL\_DB : ClientDL,IClientDL

{

private static DbConfig db = DbConfig.GetInstance();

private static ClientDL\_DB ClientDBInstance;

private ClientDL\_DB(string connectionstring)

{

LoadClients();

}

public static ClientDL\_DB ClientDBGetInstance(string connectionstring)

{

if (ClientDBInstance == null)

{

ClientDBInstance = new ClientDL\_DB(connectionstring);

}

return ClientDBInstance;

}

public void AddClient(Client client)

{

Clients.Add(client);

StoreClients(client);

}

public override void LoadClients()

{

string name, password, role;

string searchquery = "Select \* From Clients";

SqlCommand command = new SqlCommand(searchquery, db.GetConnection());

SqlDataAdapter da = new SqlDataAdapter();

da.SelectCommand = command;

DataTable dt = new DataTable();

da.Fill(dt);

for (int i = 0; i < dt.Rows.Count; i++)

{

name = dt.Rows[i]["ClientName"].ToString();

password = dt.Rows[i]["ClientPassword"].ToString();

role = dt.Rows[i]["Role"].ToString();

Client client = new Client(name, password, role);

client.SetBookedFlights(ReturnReservedFlights(name));

string feedback = dt.Rows[i]["FeedBack"].ToString();

client.SetFeedBack(feedback);

Clients.Add(client);

}

}

public override void StoreClients(Client cl)

{

string query = string.Format("INSERT INTO Clients(ClientName,ClientPassword,Role,FeedBack)" + "Values ('{0}','{1}','{2}','{3}')", cl.GetName(), cl.GetPassword(), cl.GetRole(), cl.GetFeedBack());

SqlCommand cmd = new SqlCommand(query, db.GetConnection());

cmd.ExecuteNonQuery();

}

public override void UpdateFeedBack(string Name, string feedback)

{

string query = string.Format("UPDATE Clients SET FeedBack='{0}'WHERE ClientName='{1}'", feedback, Name);

SqlCommand cmd = new SqlCommand(query, db.GetConnection());

cmd.ExecuteNonQuery();

}

public override List<Flight> ReturnReservedFlights(string ClientName)

{

string FlightID;

string searchquery = String.Format("Select \* From ReservedFlights Where ClientName='{0}'", ClientName);

SqlCommand command = new SqlCommand(searchquery, db.GetConnection());

List<Flight> reservedflights = new List<Flight>();

SqlDataReader reader = command.ExecuteReader();

while (reader.Read())

{

FlightID = reader.GetString(0);

for (int i = 0; i < FlightDL\_DB.Flights.Count; i++)

{

if (FlightDL\_DB.Flights[i].GetFlightID() == FlightID)

{

reservedflights.Add(FlightDL\_DB.Flights[i]);

}

}

}

reader.Close();

return reservedflights;

}

public override void StoreBookedFlights(string FlightID, string Name)

{

string query = string.Format("INSERT INTO ReservedFlights(FlightID,ClientName)" + "Values ('{0}','{1}')", FlightID, Name);

SqlCommand cmd = new SqlCommand(query, db.GetConnection());

cmd.ExecuteNonQuery();

}

public override void DeleteBookedFlights(string flightID, string Name)

{

string query = string.Format("DELETE FROM ReservedFlights WHERE FlightID='{0}' AND ClientName='{1}'", flightID, Name);

SqlCommand cmd = new SqlCommand(query, db.GetConnection());

cmd.ExecuteNonQuery();

}

}

}

**DL FH**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Data;

using System.Data.SqlClient;

using System.IO;

using NewLibrary.AbstractClasses;

namespace NewLibrary

{

public class ClientDL\_FH :ClientDL ,IClientDL

{

private static DbConfig db = DbConfig.GetInstance();

public static string filepath;

private static ClientDL\_FH ClientFHInstance;

private ClientDL\_FH(string Filepath)

{

filepath = Filepath;

LoadClients();

}

public static ClientDL\_FH ClientFHGetInstance(string filepath)

{

if (ClientFHInstance == null)

{

ClientFHInstance = new ClientDL\_FH(filepath);

}

return ClientFHInstance;

}

public void AddClient(Client client)

{

Clients.Add(client);

StoreClients(client);

}

public override void LoadClients()

{

string record;

string name, password, role, feedback;

if (File.Exists(filepath))

{

StreamReader Clientfile = new StreamReader(filepath);

while ((record = Clientfile.ReadLine()) != null)

{

string[] splittedrecord = record.Split(',');

name = splittedrecord[0];

password = splittedrecord[1];

role = splittedrecord[2];

feedback = splittedrecord[3];

Client cl = new Client(name, password, role);

Clients.Add(cl);

cl.SetFeedBack(feedback);

string splittedflights = splittedrecord[4];

cl.SetBookedFlights(ReturnReservedFlights(splittedflights));

}

Clientfile.Close();

}

else

{

Console.Write("File Not Found");

}

}

public override void StoreClients(Client cl)

{

StreamWriter Clientfile = new StreamWriter(filepath, true);

Clientfile.Write($"{cl.GetName()},{cl.GetPassword()},{cl.GetRole()},{cl.GetFeedBack()},");

List<Flight> flights = cl.GetBookedFlights();

//if count is 0 then write new line

if (flights.Count == 0)

{

Clientfile.Write("\n");

Clientfile.Flush();

Clientfile.Close();

return;

}

foreach (Flight flight in flights)

{

Clientfile.Write(flight.GetFlightID());

if (flight != flights.Last())

{

Clientfile.Write(";");

}

else

{

Clientfile.Write("\n");

}

}

Clientfile.Flush();

Clientfile.Close();

}

public override void UpdateFeedBack(string Name, string feedback)

{

File.WriteAllText(filepath, string.Empty);

foreach (Client cl in Clients)

{

StoreClients(cl);

}

}

public override List<Flight> ReturnReservedFlights(string ClientName)

{

List<Flight> reservedflights = new List<Flight>();

string FlightID;

string[] splittedflights = ClientName.Split(';');

for (int i = 0; i < splittedflights.Length; i++)

{

FlightID = splittedflights[i];

for (int x = 0; x < FlightDL\_FH.Flights.Count; x++)

{

if (FlightID == FlightDL\_FH.Flights[x].GetFlightID())

{

Flight fl = FlightDL\_FH.Flights[x];

reservedflights.Add(fl);

}

}

}

return reservedflights;

}

public override void StoreBookedFlights(string FlightID, string Name)

{

string clfilepath = ClientDL\_FH.filepath;

File.WriteAllText(clfilepath, string.Empty);

foreach (Client cl in ClientDL\_FH.Clients)

{

StoreClients(cl);

}

}

public override void DeleteBookedFlights(string flightID, string Name)

{

string clfilepath = ClientDL\_FH.filepath;

File.WriteAllText(clfilepath, string.Empty);

foreach (Client cl in ClientDL\_FH.Clients)

{

StoreClients(cl);

}

}

}

}

**Interfaces**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace NewLibrary

{

public interface IClientDL

{

void AddClient(Client client);

bool IsClientExist(string name, string password);

int FindClient(string name, string password);

bool CheckValidClientName(string name);

void LoadClients();

void StoreClients(Client cl);

void UpdateFeedBack(string Name, string feedback);

List<Client> GetAllClients();

List<Flight> ReturnReservedFlights(string ClientName);

void StoreBookedFlights(string FlightID, string Name);

void DeleteBookedFlights(string flightID, string Name);

}

}