City University of Hong Kong

Department of Computer Science

CS3343 Software Engineering Practice

2022/23 Semester A

Hong Kong-Mainland Flight Ticket Booking System

Analysis and Design Report

Project Group 5

ZHAO Yiwei	(56641836, PM)
LI Xiangyu	(56641559)
WANG Chenchen	(56344983)
WANG Yian	(56641105)
XIE Yifei	(56641547)
YANG Jingxian	(56644285)

1 Introduction

With the continuous progress of social development, the growth of civil aviation and the improvement of people's consumption level, more and more consumers take civil aviation, and the ticket booking system has begun to affect people's daily life and travel, and has become increasingly important. The original system has become unable to meet the demand with the rapid growth of passenger capacity of airlines and the improvement of people's requirements for convenience.

Especially currently, the influence of the pandemic has made the administration of Hong Kong and mainland China introduce traffic control measures and isolation approach, which remarkably affects communication between residence of mainland and Hong Kong. The latest data from CTS shows that on November 5, the number of international air tickets booked reached a new high this year. The data shows that since November, international air ticket bookings have shown a significant upward trend. In the first half of November, the number of international air tickets booked by domestic airlines increased by nearly 140% year on year. In this case, the original system efficiency is seriously insufficient.

On November 11, the joint prevention and control mechanism of the State Council issued the latest notice, announcing several measures to optimize and adjust the prevention and control measures. Among them, the circuit breaker mechanism for inbound flights was canceled, and the centralized isolation time for inbound personnel was shortened by two days. After the news was released, the instantaneous search volume of international air tickets on Qunar platform reached the peak in nearly a year. The search volume of air tickets on Ctrip platform doubled compared with the previous day, and the search volume of international air tickets on the same journey also increased by 448% within one hour compared with the same period of the previous day.

After the release of the "20" measures, many insiders believed that it would make the way for passengers to return home more convenient, thus contributing to the further increase of international flights. Since the second half of the year, the number of international flights that have been strictly controlled in the past two years has substantially increased. The number of international flights planned to be carried out by SIA since October 30 has more than doubled compared with the same period last year in winter and spring.

Thus, our group expected to develop a new program to help residence to go back and force in Hong Kong and mainland, and to bring an efficient operation interface. Users can choose their departure and destination and the departure time on the program and the program will show a clear list of ticket in order of cost. Users can also change the logic of sorting. By the clear operation interface, users will get a better experience and will make a significant progress of the efficiency of air ticket booking system, which will facilitate the development of tourism, and therefore promote the economic affected by the pandemic recovery.

2 Design Constraints

2.1 Commercial Constraints

The time limit of the project is 13 weeks from the launch of the program on September 2nd. The time is relatively tight and some extensive functions cannot be done, so we can only simplify the design of the program.

Also, our manpower is limited. There are only 6 members in our team, so it will require our team members to put more effort into our own responsible field. Members need to learn and be familiar with the target and make efforts to achieve it.

2.2 Compliance Constraints

Our program will integrate all the tickets information of planes travel to and from between Hong Kong and the mainland China. However, the different ticket website will show different price for same flight. In this case, we should check all the information and choose the suitable ticket for users by some algorithm.

Also, we should ensure the formality and accuracy of the data source. We need to do a thorough early-stage investigation and find the data from the official data providers.

2.3 Functional Constraints

Our program should be equipped with following key functions according to the requirement:

1. Users can choose the departure and destination cities and the departure time and acquire the ideal flight ticket from the system.
2. Maintenance personnel can add or remove the flights according to the top management of the corporation or the flight company's announcement.
3. The system should have different languages for different groups of potential users

2.4 Non-functional Constraints

On the premise of satisfying the above functions, we expect our program to optimize other characteristics:

1. Fast to compute the route
2. Easy UI actions
3. Simple maintenance operations

2.5 Stylistic Constraints

Because of the limitation of the language (Java) and the lack of skills and experience of the members, we only expect to successfully reflect the information of tickets users reserved, and we do not require other more stylistic designs.

2.6 Systems Constraints

Due to the time limit and user requirements, the system will only be executable on the Windows/Mac system. We do not provide the mobile-end planning system.

2.7 Skill Level Constraints

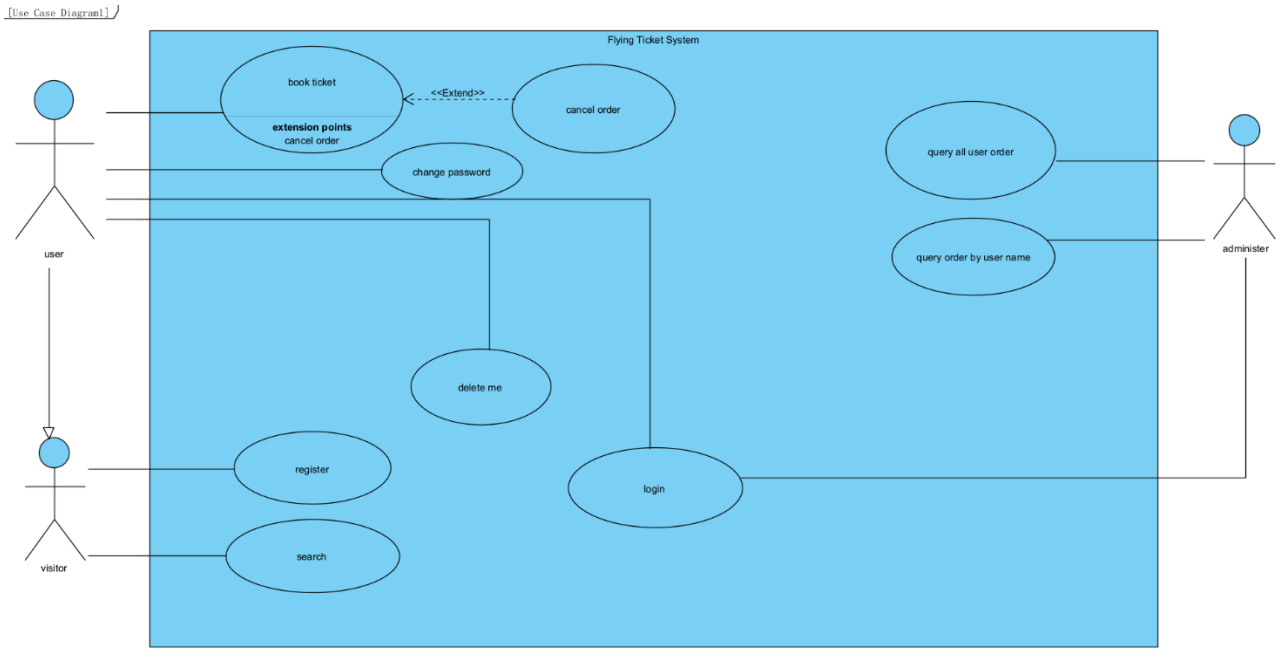
This is our first time to process such a general project, and we lack related experience.

For the use of the IDE and version control tools, we need to adapt them for some time. Also, none of us had processed UI programming before, and none of us know about SWING, a GUI toolkit designed for Java.

Also, we did not know about the knowledge of project management, and a good management will lead to smoother progress of future works. Therefore, we still need to do that to gain experience.

3 Use Case Diagram and Specifications

3.1 Use Case Diagram



3.2 Use Case Specifications

4 Use-Case Name:	Book Ticket	
Actor(s):	User	
Description:	This use case describes the process of booking ticket.	
Reference ID:	TS-01	
Precondition:	The user has been logged in.	
Trigger:	This use case is initiated when the user clicks the button to book the ticket in flights list page.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: This use case is initiated when the user clicks the button.	**Step 2:** The system checks the flights information in the database. **Step 3:** The system modifies the seat number information and ticket status in the database. **Step 4:** The system updates the order data. **Step 5:** The system displays the ticket.
Alternative Courses:	**Step 3:** If the purchased is greater than the sold, the system prompts the user that the quantity is insufficient.	
Postcondition or Results:	The user sees the booking is successful.	

Use-Case Name:	Cancel Order	
Actor(s):	User	
Description:	This use case describes the process of canceling order.	
Reference ID:	TS-02	
Precondition:	There exists order booked before.	
Trigger:	This use case is initiated when the user clicks the button to cancel the order in my order page.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: This use case is initiated when the user clicks the button.	**Step 2:** The system checks the order information in the database.

		Step 3: The system deletes the order from Order table. **Step 4:** The system calls my order page.
Alternative Courses:	Nil	
Postcondition or Results:	The user sees the cancelling is successful.	

Use-Case Name:	Change Password	
Actor(s):	User	
Description:	This use case describes the process of changing password.	
Reference ID:	TS-03	
Precondition:	The user has been logged in.	
Trigger:	This use case is initiated when the user clicks the button to change the password in personal page.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: This use case is initiated when the user clicks the button. **Step 2:** The user inputs the new password.	**Step 3:** The system replaces the old password with the new one in the database. **Step 4:** The system backs to the personal page.
Alternative Courses:	Nil	
Postcondition or Results:	The user sees the changing is successful.	

Use-Case Name:	Delete Me
Actor(s):	User
Description:	This use case describes the process of deleting the user account.
Reference ID:	TS-04
Precondition:	The user has been logged in.
Trigger:	This use case is initiated when the user clicks the button to delete his account in personal page.

Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: This use case is initiated when the user clicks the button. **Step 2:** The user inputs the password.	**Step 3:** The system verifies the username and password in the User table. **Step 4:** The system deletes the corresponding data. **Step 5:** The system jumps to the home page.
Alternative Courses:	**Step 4:** If the password is incorrect, the system noticed the user wrong inputted password.	
Postcondition or Results:	The user sees the deleting is successful.	

Use-Case Name:	Login	
Actor(s):	User, Admin	
Description:	This use case describes the process of login.	
Reference ID:	TS-05	
Precondition:	Nil	
Trigger:	This use case is initiated when user clicks the button to login.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: This use case is initiated when the user clicks the button. **Step 2:** The user inputs the username and the password.	**Step 3:** The system verifies the username and password in the User table. **Step 4:** The system posts the login success message.
Alternative Courses:	**Step 4:** If the username does not exist or the password is not matching the username, the system returns the error.	

Postcondition or Results:	The user sees the login is successful.

Use-Case Name:	Register	
Actor(s):	Visitor	
Description:	This use case describes the process of register.	
Reference ID:	TS-06	
Precondition:	Nil	
Trigger:	This use case is initiated when user clicks the button to register.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: This use case is initiated when the user clicks the button. **Step 2:** The user inputs the username, the password and repeated password.	**Step 3:** The system compares the password and the repeated password. **Step 4:** The system inserts the data into User table. **Step 5:** The system logins this user account.
Alternative Courses:	**Step 4:** The system tells the user that the two passwords are inconsistent.	
Postcondition or Results:	The user sees the signup and login are successful.	

Use-Case Name:	Search	
Actor(s):	Visitor, User	
Description:	This use case describes the process of search.	
Reference ID:	TS-07	
Precondition:	Nil	
Trigger:	This use case is initiated when user inputs the departure, destination, and date, then clicks the button to search in the home page.	
Typical Course of Events:	**Actor Action**	**System Response**

	Step 1: This use case is initiated when user inputs the information and clicks the button. **Step 4:** The user asks the flight list to query by price.	**Step 2:** The system searches the flight with departure and destination in the day. **Step 3:** The system gives a list of flights with increased time. **Step 5:** The system gives a list of flights with increased price.
Alternative Courses:	**Step 3:** If the departure or destination is not supported in the city array, the system tells the user that the aircraft service in the city is not currently supported.	
Postcondition or Results:	The user sees the flights list.	

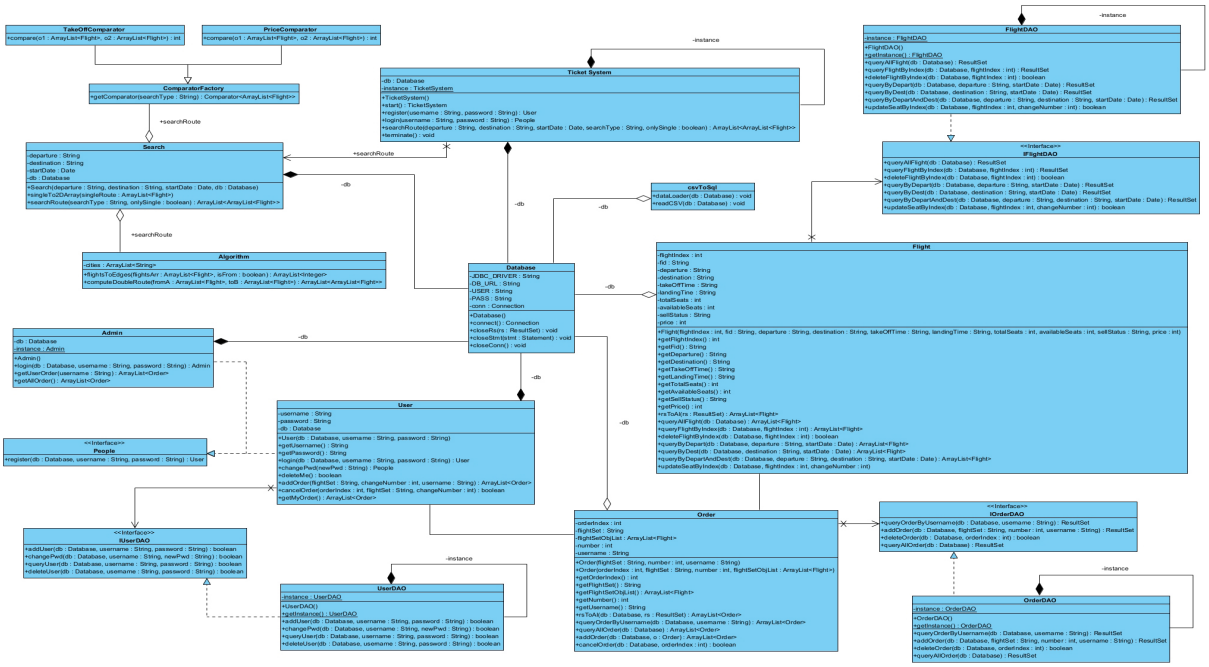
Use-Case Name:	Query All User Order	
Actor(s):	Admin	
Description:	This use case describes the process of querying all user orders.	
Reference ID:	TS-08	
Precondition:	The admin has been logged in.	
Trigger:	This use case is initiated when the admin clicks the button to query all user order.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: This use case is initiated when the admin clicks the button.	**Step 2:** The system selects all order data in the Order table. **Step 3:** The system shows the order list.
Alternative Courses:	Nil	

Postcondition or Results:	The admin sees the order list.

Use-Case Name:	Query Order by User Name	
Actor(s):	Admin	
Description:	This use case describes the process of querying order by username.	
Reference ID:	TS-09	
Precondition:	The admin has been logged in.	
Trigger:	This use case is initiated when the admin clicks the button to query order by username.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: This use case is initiated when the admin clicks the button. **Step 2:** The admin inputs the username.	**Step 4:** The system selects all order data in the Order table with this username. **Step 5:** The system shows the order list.
Alternative Courses:	**Step 4:** If the username does not exists, the system outputs an error.	
Postcondition or Results:	The admin sees the order list.	

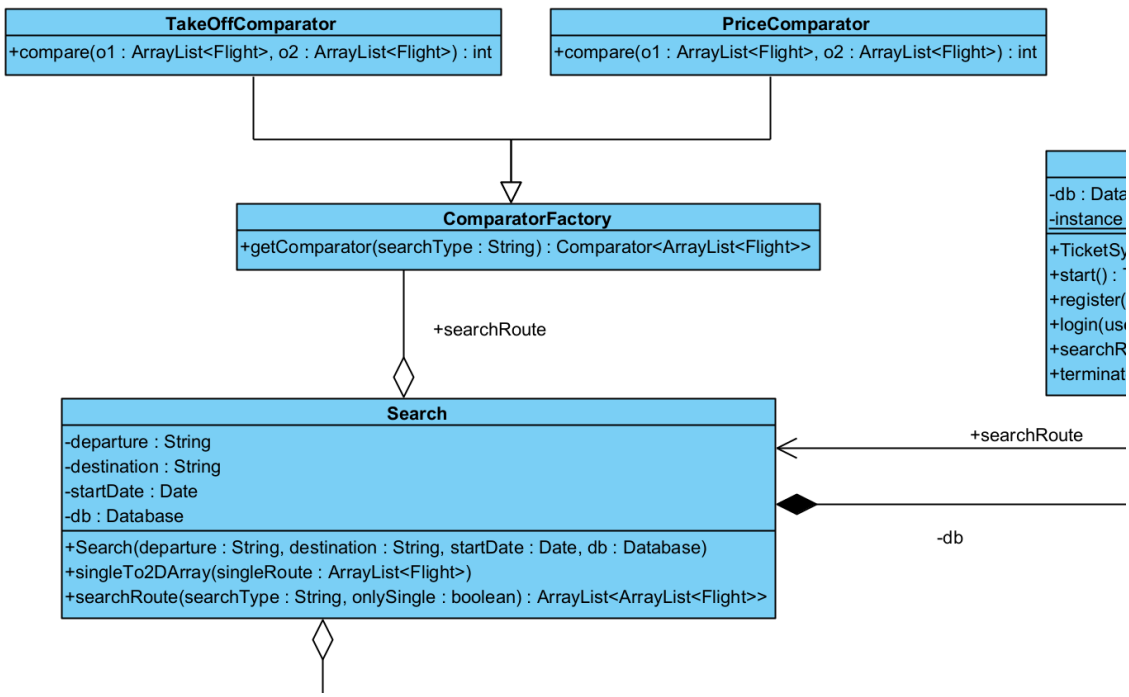
4 Class Diagram Design

4.1 Overall Class Diagram



4.2 Design Principles

4.2.1 Open-Close Principle (OCP)

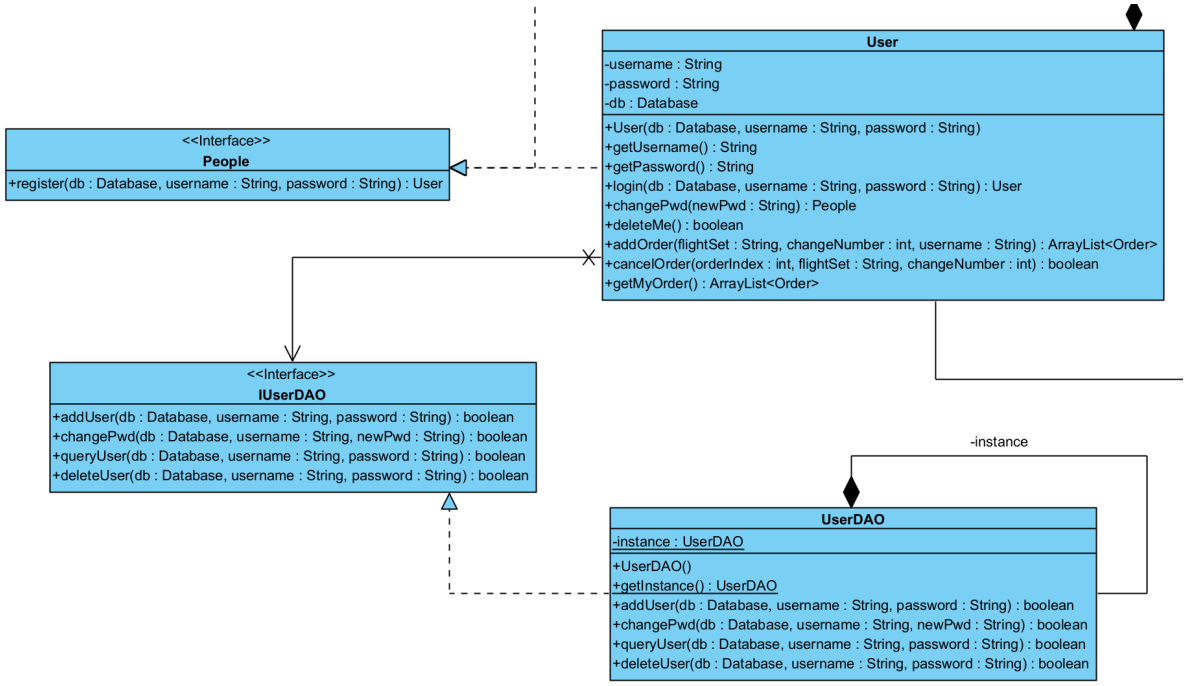


Open-Closed Principle (OCP) states that classes should be open for extension but closed for modification, which is usually achieved by inheritance so that new subclasses could be easily extended. Compliance with OCP makes it easy to extend classes and incorporate new behavior without modifying existing code. Such a design is resilient to change and can accept new features to meet changing needs.

In our system design, we fully implemented the requirements of OCP, such as the "Comparator Factory" and "Search" in the above diagram is a typical example.

Our "Search" is based on the "Comparator Factory" superclass. We only need to know that "getComparator" can compare and give the correct list without knowing its internal working flow or specific codes of the "Search". When we develop a more suitable algorithm in the future, we just need to make this new algorithm class implement the "Comparator Factory" interface, and then we can directly call it in the "Search". Similarly, if a new criterion comes along, we can choose or develop an algorithm that will do the job. All of this is done without knowing the details of the existing code, or even making any changes, simply inheriting the corresponding superclass and calling it when appropriate.

4.2.2 Dependency Inversion Principle (DIP)



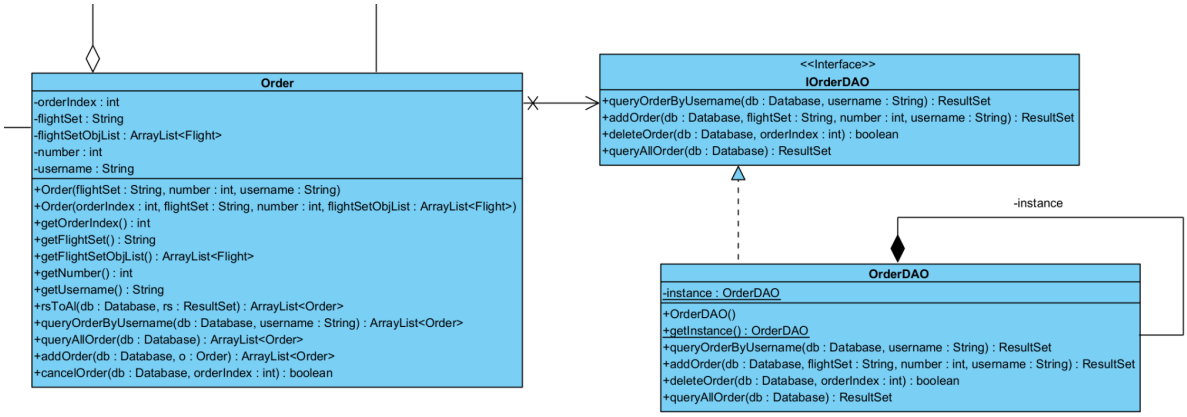
The dependency Inversion Principle (DIP) states that high-level modules should not depend on Low-level modules. Both of them should depend on abstractions. DIP using abstract/virtual class or interface is a way to achieve OCP mentioned above.

We keep this principle in mind when designing our projects, such as the "People – User – IUserDAO - UserDAO" module at the back end.

In this module, all programming activities are directed towards the "interface" rather than the specific underlying "implementation". That is, the high-level component "UserDAO" only uses an abstract "IUserDAO". Similarly, in the Factory method pattern, the "User" class requires only one variable that belongs to the high-level component, "People", while the specific instantiation of one of the many compliant data lists is determined at run time depending on the particular situation.

Following this DIP design principle allows us to greatly reduce the dependency of our programs, making our software more flexible to adapt to future changes and challenges.

4.2.3 Law of Demeter (LoD)

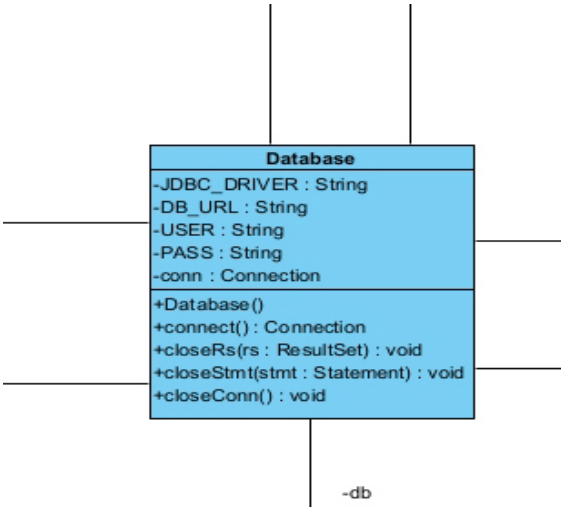


The Law of Demeter is a design guideline for developing software applications. This principle states that an object should never know the internal details of other objects.

In our project, we try to make the objects independent of each other as much as possible. The OrderDAO class only invokes itself and direct component objects. Its behavior of a specific method, such as queryOrderByUsername() is completed by itself, rather than pointing to another object Order to implement.

The LoD principle can reduce the dependency between classes and coupling, which is defined as the degree of interdependence that exists between software modules and how closely such modules are connected to each other. Therefore, our software can be easily maintained, tested, and updated.

4.2.4 Single Responsibility Principle



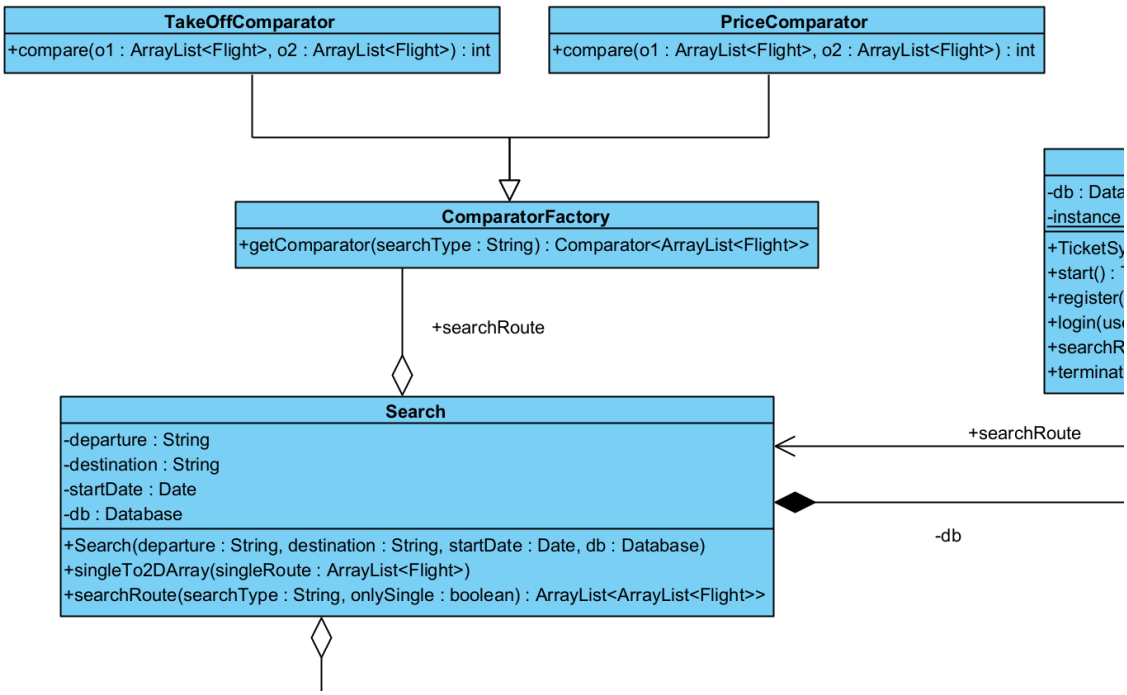
The single responsibility principle is a computer programming principle that states that every module, class, or function should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility.

For the Database class, we follow close to the line of Single Responsibility Principle. In this class, methods have the function of methods, classes have the function of classes, and modules have the function of modules. Each function in the class only provides functionality for its own responsibility of the Database.

The core of single responsibility principle is decoupling and enhancing cohesion. If a class has too many responsibilities, it is equivalent to coupling these responsibilities. A change in one duty may weaken or inhibit the ability of this class to perform other duties. This coupling will lead to fragile design, and when changes occur, the design will suffer unexpected damage. If you want to avoid this phenomenon, you should follow the principle of single responsibility as far as possible.

4.3 Design Patterns

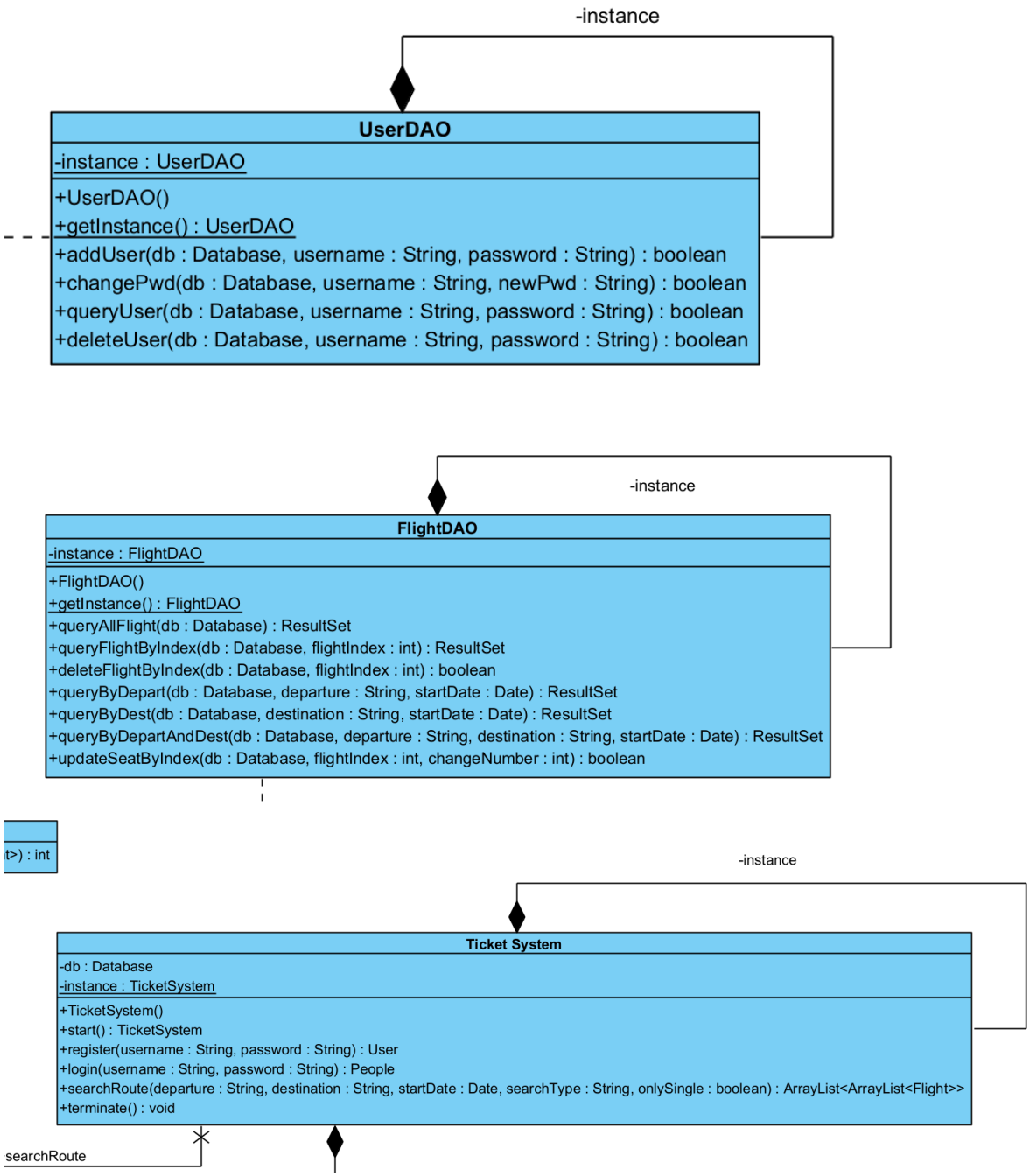
4.3.1 Factory Method Pattern



We apply factory method pattern here to encapsulate the instantiation of a specific array list type. Just like the above figure, a factory method interface called "searchRoute(searchType: String, onlySIngle: boolean)" in the abstract Search class is to create concrete objects. Other method used in this abstract class may use the concrete "ArrayLiist<ArrayList<Flight>>" created by the factory method function.

We implement factory mathed pattern have many advantages. It allows us to create classes and all the other functionality closely related to it and we do not need to figure out which product is actually created. By using it, different classes can be connected tighter and will not be looked so separate and alone, the OCP principle is implemented and the cohesiveness of the system is ensured. In practice, the choice of which subclasses in our system will naturally determine the actual array list.

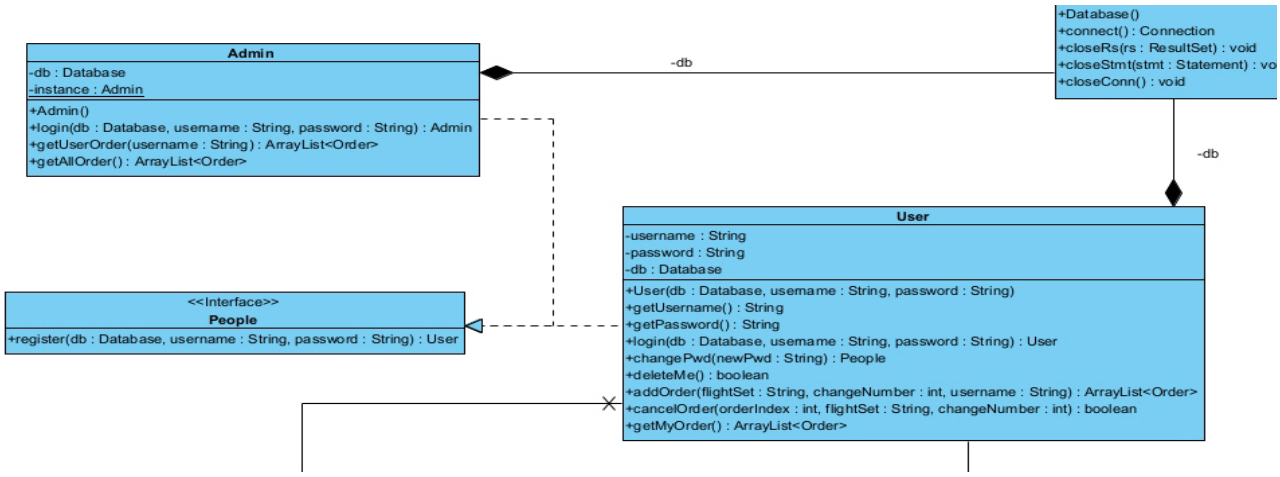
4.3.2 Singleton Pattern



We frequently implement Singleton pattern in our project design, like TicketSystem, FlightDAO and UserDAO. If a class only needs a single object, we will use the Singleton pattern to ensure that no additional instances are created that would lead to inconsistent information. It significantly decrease the system's resourse usage and

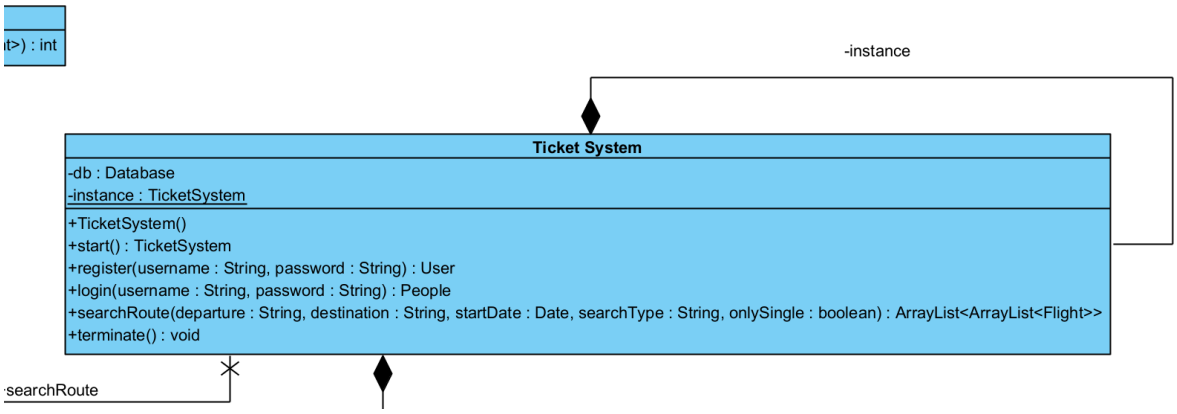
avoid wastes, therefore our program's performance is improved a lot. As shown above, TicketSystem, FlightDAO and UserDAO thses three classes show good examples of using this pattern.

4.3.3 State Pattern



The state pattern is mainly used to solve the problem that objects need to output different behaviors to the outside world when they are in multiple state transitions. We define a interface which consists of many state objects, and other "states" like "Admin" and " User" also include the interface's objects, the methods used in classes like "login()", "getUsername()" and "getUserOrder()" are all from interface. The state and behavior are one-to-one, and the states can be converted to each other. When the intrinsic state of an object changes, it is allowed to change its behavior, and the object looks like it has changed its class.

4.3.4 Facade Pattern



Facade pattern provides an enteracne for a set of interfaces in subsystem, indicating that a “high-level interface” is defined. In our system, we define class TicketSystem as Facade, it contains many basic functions that are also available in its subsystem classes. This Facade makes the subsystem easier to use (encapsulates the complexity of the subsystem and makes it easier for the client to invoke). The Facade pattern shields the details of the internal subsystem by defining a consistent interface, so that the caller only needs to make calls with this interface and does not need to care about the internal details of the subsystem.

5 PROGRAM FLOW AND ALGORITHMS

5.1 overview

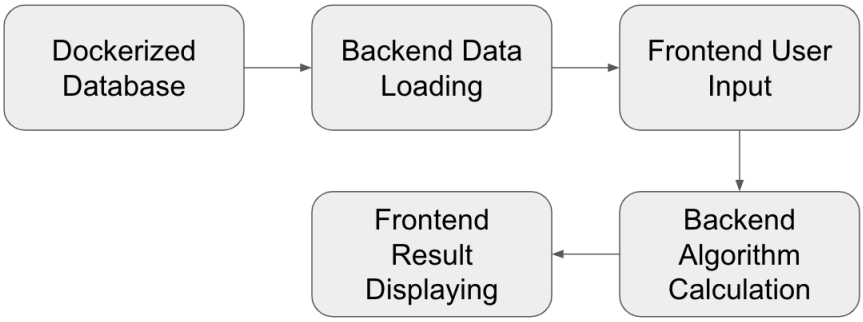


Figure 5.1 Program Flow of the System

The whole flow of the internal program can be separated into five parts as shown above.

This project uses MySql database to store the flights, orders and users information. In order to make the project easier to test and use, we implemented a dockerized database for the project. After running the shell script, there will be a new mysql database container running in the backend.

The flight route information is all stored in a well-structured csv file containing about 30 thousand entries because an external API is not allowed in this project. A data processing function is written to process the data in csv, and the whole structure is designed to be easy when switching the data source, that is, a developer can just add a new data process function if necessary. The data process function is also well- designed to simulate the real-time data behavior, that is, dates in the database will change automatically according to the first time of loading.

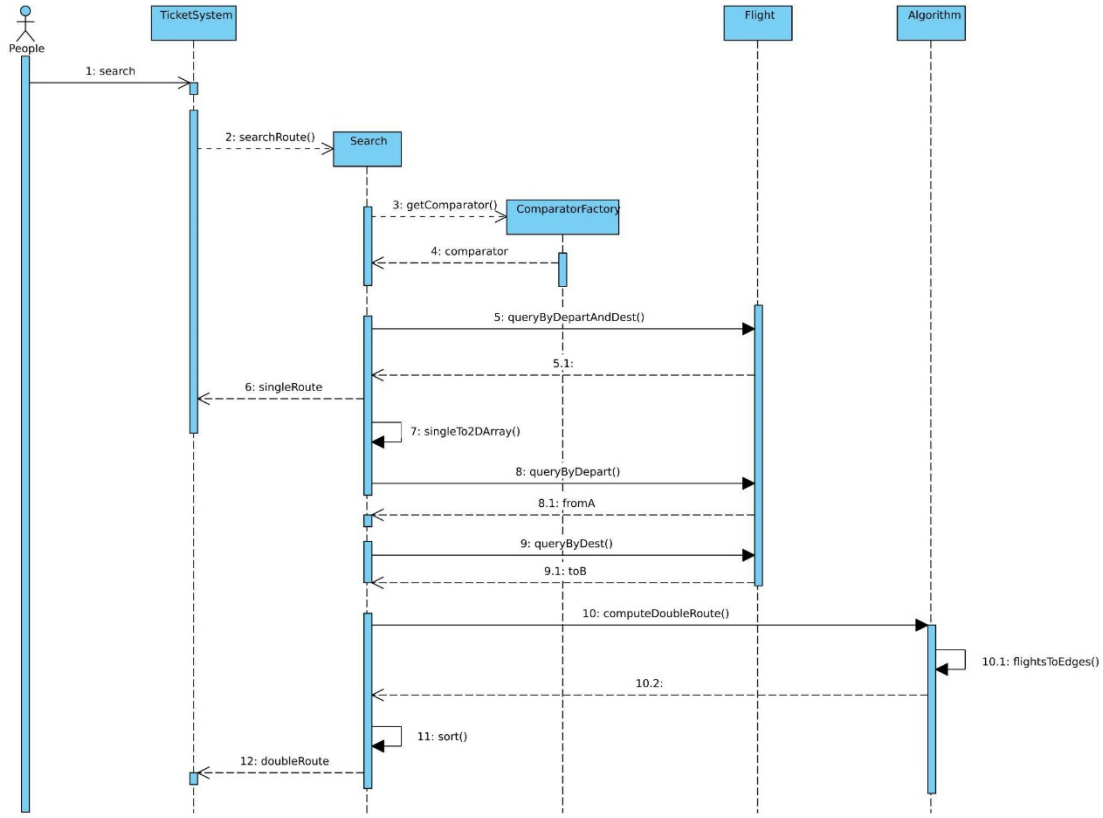
Data will be loaded when the user opens this project for the first time, and can be updated manually by running the *PreStart.java* file. After the database and the backend is ready, the user can input the departure and arrival location with the take-off date. After clicking search, the backend will calculate the routes using the core algorithm and

then pass the result to the frontend to display. Users can also select the sorting behavior depending on his or her needs.

5.2 Sequence Diagrams

5.2.1 Core Algorithm

Basically, our search algorithm queries data entries according to different requirements and then converts them to the edges of a graph, and then searches whether there are routes between the place of departure and destination. Finally, the result is passed to different comparators to fulfill the users' needs.



In the search function, a boolean called *onlySingle* can be used to decide if the user only wants direct flights. As a result, the direct flights and non-direct flights are calculated separately.

- Goodness of Fit

  As there may be different user requirements, providing a choice for users who only need direct flights will improve the algorithm efficiency because

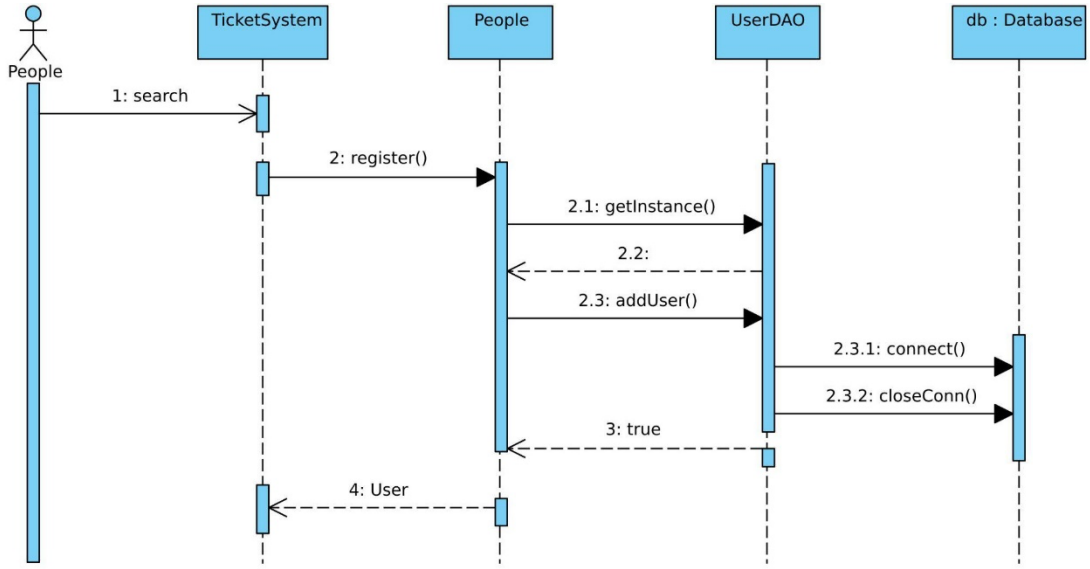
calculating direct flights is faster. Converting the cities into nodes and flights to edges makes it easy to implement graph algorithms and calculating routes. The sorting of routes is handled by the comparators implemented in the factory pattern, which is relatively independent of the core route calculation, which is easy when expanding the program.

- Code Implementation

Data	**Usage**
String departure	The place of departure.
String destination	The destination.
String startDate	The take-off date.
Database db	The database used.
String searchType	The type of comparator to use.
boolean onlySingle	If only direct flights are needed.
ArrayList<ArrayList<Flight>> singleRoute	The direct flights result.
ArrayList<Flight> FromA	The flights depart from the place of departure.
ArrayList<Flight> ToB	The flights depart to the destination.
ArrayList<ArrayList<Flight>> ret	The final sorted result.

First the function will be initialized by the first four parameters. After that, the function will get the direct flights by querying the database directly and format the return values as *singleRoute*. If the user also needs non-direct flights, the function will query the database by destination and the place of departure to find if there exists available flights to get there. An available flight should be open for selling, that is, the available number of seats should not be zero.

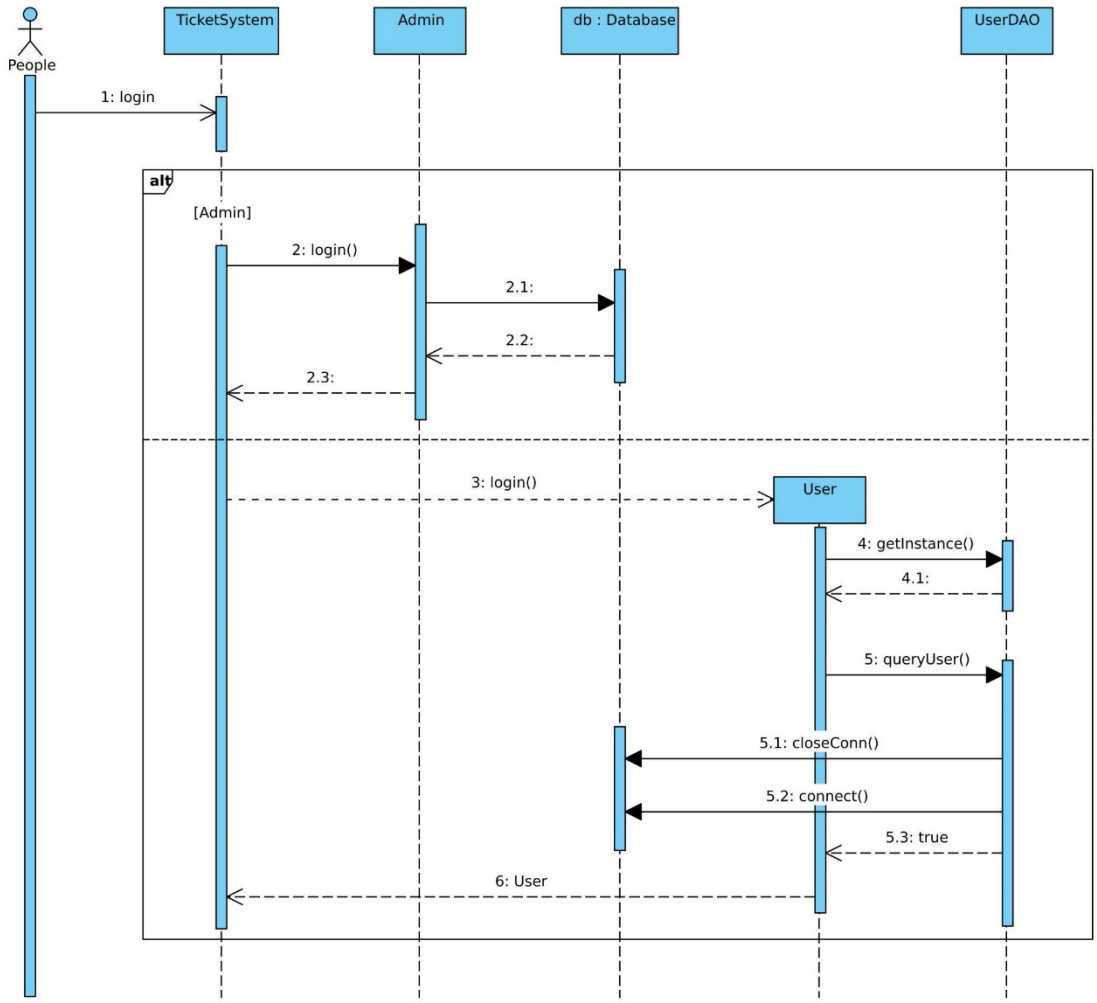
5.2.2 Register Flow



In our system, a new user can search for the flight routes without registering an account, but if he or she wants to make an order, an account is needed. We store all the user information in the database, including the orders every user made.

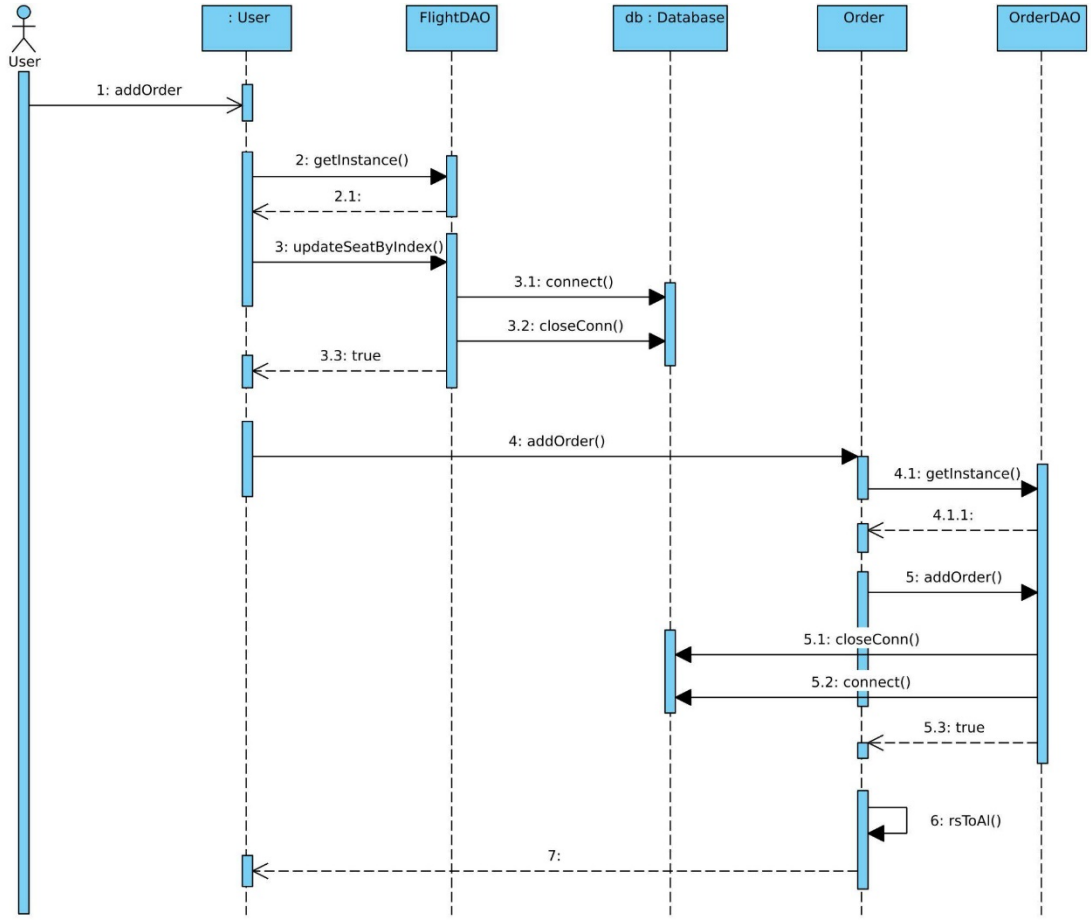
When the user wants to make an order after searching, he or she first clicks the sign up button, puts in the username and password and then the database access object will handle the request and create a new query in the database.

5.2.3 Login Flow



There are two kinds of accounts in our system, that is, the admin and normal users. When logging in, the system will automatically detect which role is the account. And then the frontend will display different interfaces to the admin or user with different functions. The admin can query all orders and query orders by username, and the backend also provide services for the admin that he or she can query flights and delete flights, though we did not add the function to the frontend.

5.2.4 Booking Flow



Only a user can add an order. When the user clicks the *book now* button in the frontend, the system will first update the seat numbers of the relative flights, and then add the order information into the order table. When the information is successfully recorded, the backend will send back the order information to the frontend for displaying.