1. Ad Click Event Aggregation System

Real-life examples

Requirements clarification

- **Context**
  - The input data is a log file located in different servers and the latest click events are appended to the end of the log file.
  - The attributes of input data are:
    - ad_id
    - click_timestamp
    - user_id
    - ip
    - country
- **Functional requirements**
  - Support 2 queries:
    - Aggregate the number of clicks of ad_id in the last M minutes.
    - Return the top 100 most clicked ad_id every minute.
  - For above queries, support aggregation filtering by different attributes (e.g. ip, user_id, etc.).
- **Non-functional requirements**
  - Correctness of the aggregation result is important.
  - Properly handle delayed or duplicate events.
  - Robustness. The system should be resilient to partial failures.
  - End-to-end latency should be a few minutes, at most.

Estimation

System interface definition

Aggregate the number of clicks of ad_id in the last M minutes

URL: GET /v1/ads/{:ad_id}/aggregated_count

Request

Field	Description	Type
from	Start minute (default is now minus 1 minute)	long
to	End minute (default is now)	long
filter	An identifier for different filtering strategies. (For example, filter = 001 filters out non-US clicks)	long

Response

Field	Description	Type
ad_id	The identifier of the ad	string
count	The aggregated count between the start and end minutes	long

Return top N most clicked ad_ids in the last M minutes

URL: GET /v1/ads/popular_ads

Request

Field	Description	Type
count	Top N most clicked ads	integer
window	The aggregation window size (M) in minutes	integer
filter	An identifier for different filtering strategies	long

Response

Field	Description	Type
ad_ids	A list of the most clicked ads	array

Data model definition

Raw data

ad_id	click_timestamp	user	ip	country
ad001	2021-01-01 00:00:01	user1	207.148.22.22	USA
ad001	2021-01-01 00:00:02	user1	207.148.22.22	USA
ad002	2021-01-01 00:00:02	user2	209.153.56.11	USA

Aggregated data

aggregated_result table

ad_id	click_minute	filter_id	count
ad001	202101010000	0012	2
ad001	202101010000	0023	3
ad001	202101010001	0012	1
ad001	202101010001	0023	6

filter table

filter_id	region	IP	user_id
0012	US	*	*
0013	*	123.1.2.3	*

most_clicked_ads table

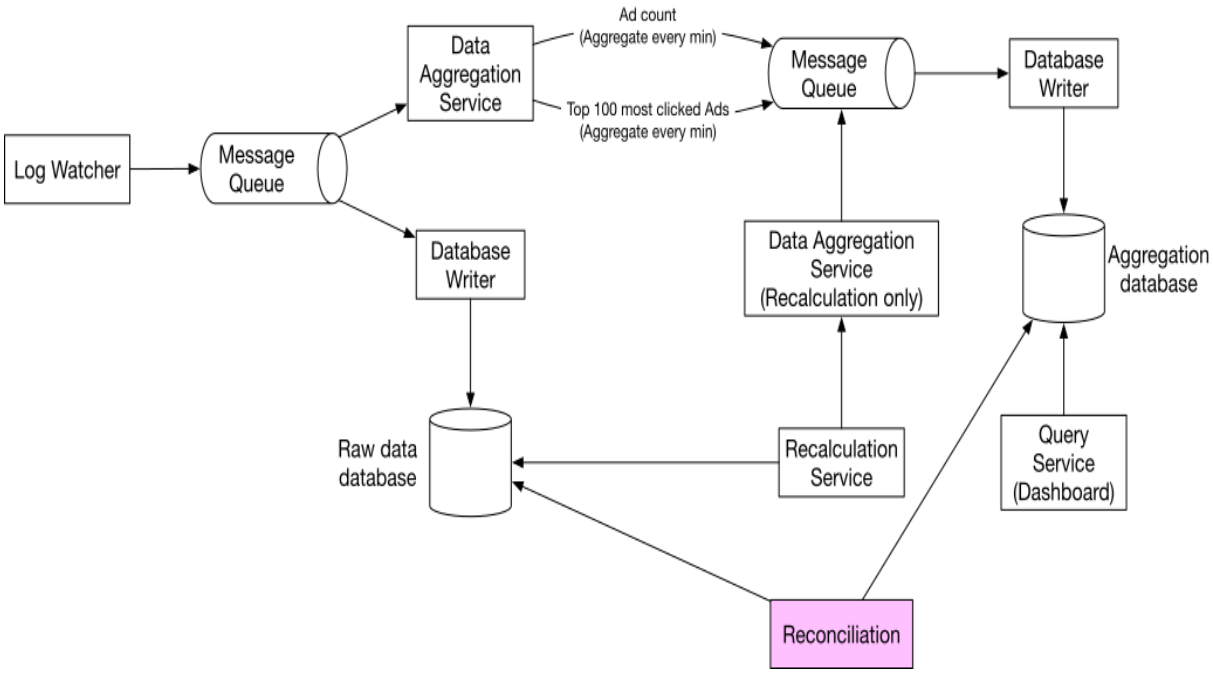
Field	Type	Description
window_size	integer	The aggregation window size (M) in minutes.
update_time_minute	timestamp	Last updated timestamp (in 1-minute granularity)
most_clicked_ads	array	List of ad IDs in JSON format.

Solution

Store both raw data and aggregated data

- The use cases of raw data
    - Debugging.
    - Recalculate the aggregated data from the raw data, after a bug fix.
    - Serve as backup data.
- The use cases of aggregated data
    - Run queries on aggregated data for better performance.

High-level design



- **Log Watcher**
    - Reads log and publish ad click event data into the left message queue.
- **Message Queue (Left)**
    - Stores ad click event data.
    -

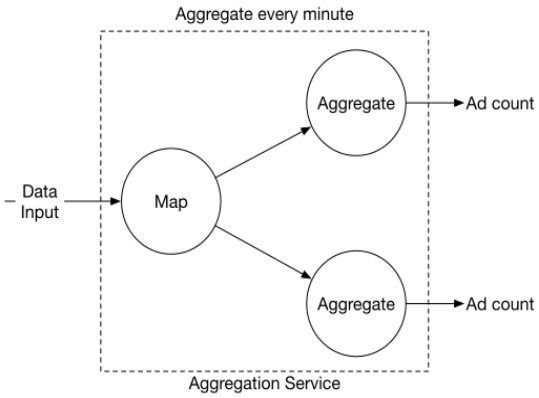
- **Message Queue (Right)**
  - Stores
    - Ad click counts aggregated at per-minute granularity.
    - Top N most clicked ads aggregated at per-minute granularity.
  - Achieves end-to-end exactly-once semantics (atomic commit).
- **Data Aggregation Service**
  - Processes raw data and generate aggregated data.
- **Recalculation Service**
  - Recalculate the raw data and generate new aggregated data when we discover a major bug.
  - Use snapshot to capture system status so that no need to replay data from the beginning of raw data.
- **Reconciliation**
  - Sort the ad click events by event time in every partition at the end of the day, by using a batch job and reconciling with the real-time aggregation result.

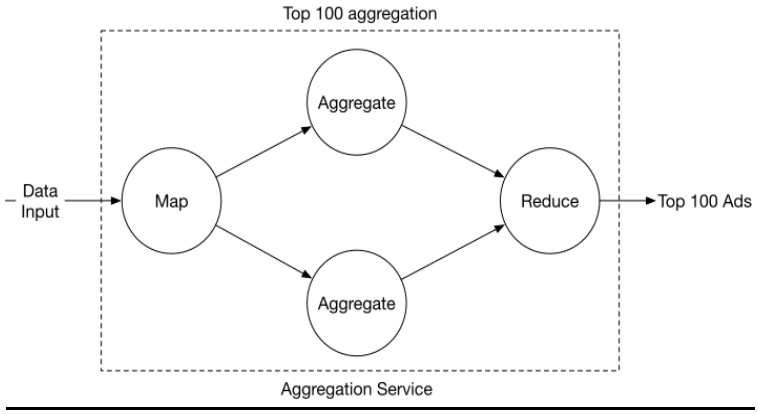
Detailed design

Data Aggregation Service

Idea

- Use MapReduce framework
- Break down the system into Map/Aggregate/Reduce nodes.





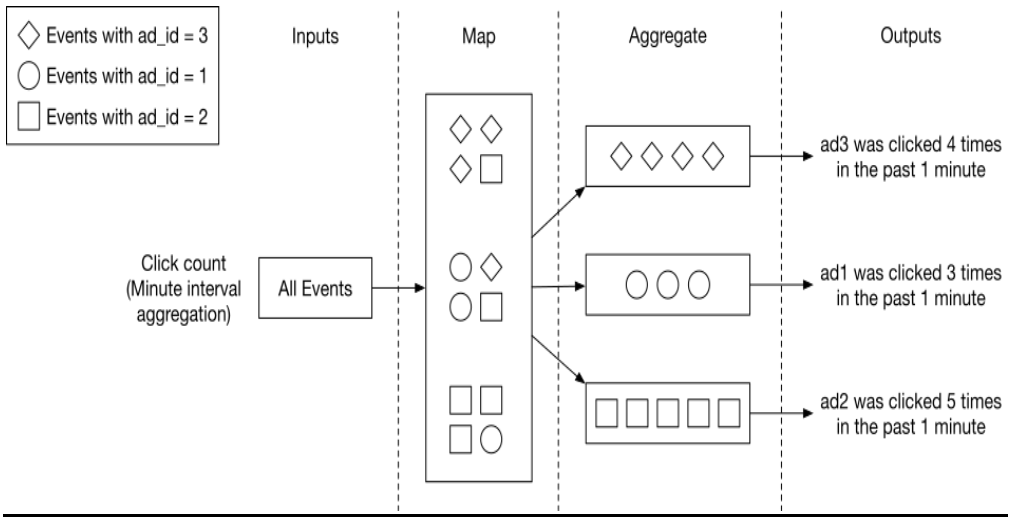
Components

- **Map node**
  - Reads data from a data source, and then filters and transforms the data.
- **Aggregate node**
  - Counts ad click events by ad_id in memory every minute.
- **Reduce node**
  - Reduces aggregated results from all “Aggregate” nodes to the final result.

Use cases

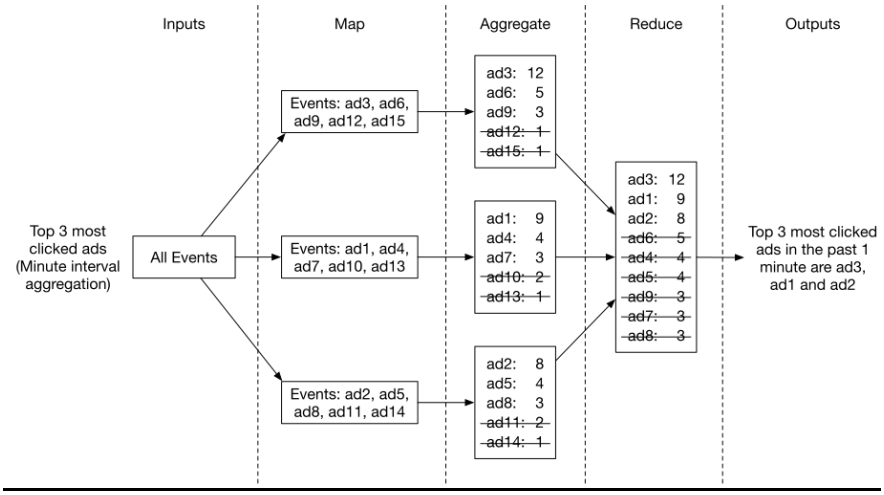
Case 1: Aggregate the number of clicks

- Input events are partitioned by ad_id (ad_id % 3) in Map nodes and are then aggregated by Aggregation nodes.



Case 2: Return top N most clicked ads

- Input events are partitioned by ad_id (ad_id % 3) in Map nodes.
- Each Aggregate node maintains a heap data structure to get the top 3 ads within the node efficiently.
- The Reduce node reduces 9 ads (top 3 from each aggregate node) to the top 3 most clicked ads every minute.



Case 3: data filtering

- Pre-defined filtering criteria and aggregate based on them

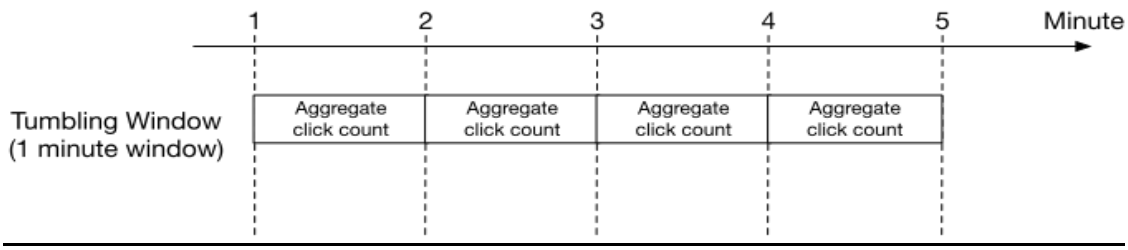
Aggregation window

- **Type of windows**
  - ✓ Tumbling window: Has a fixed length, and every event belongs to exactly one window.
  - ✓ Hopping window: Has a fixed length, but allows windows to overlap in order to provide some smoothing.
  - ✓ Sliding window: Contains all the events that occur within some interval of each other.
  - ✓ Session window: Has no fixed duration. Group together all events for the same user that occur closely together in time, and the window ends when the user has been inactive for some time.

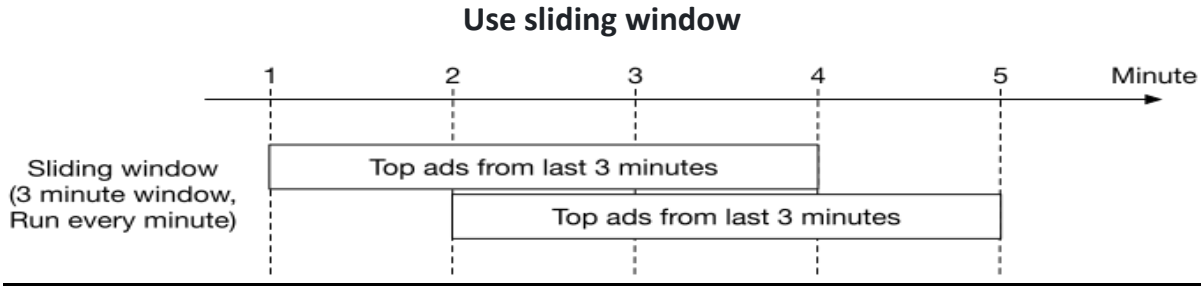
Which window should be used for our use cases

Case 1: Aggregate the number of clicks

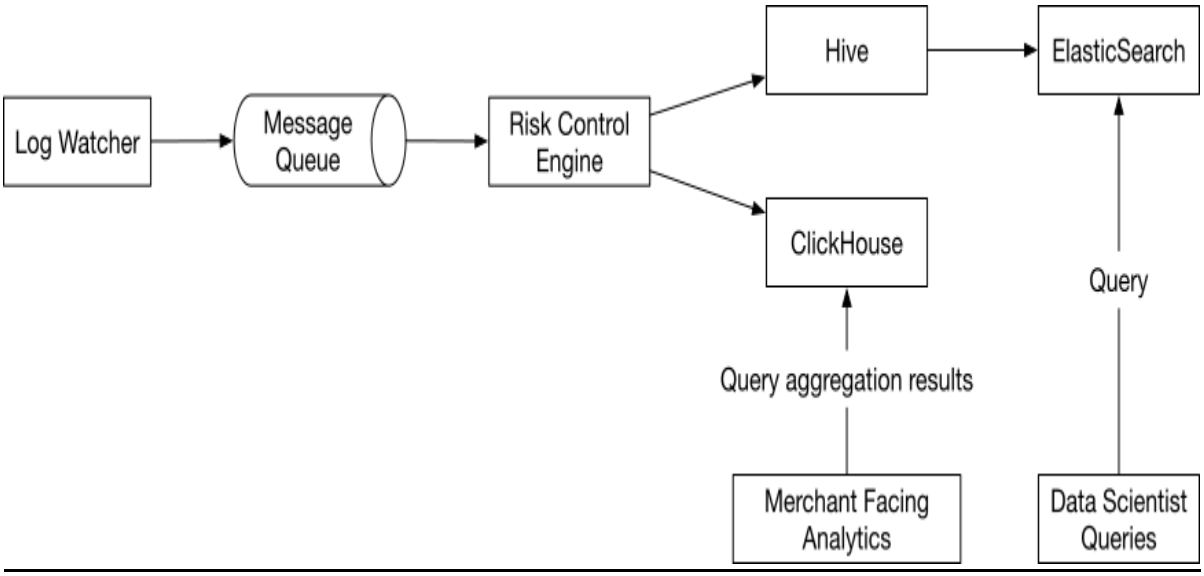
Use Tumbling window



Case 2: Return top N most clicked ads



Alternative design



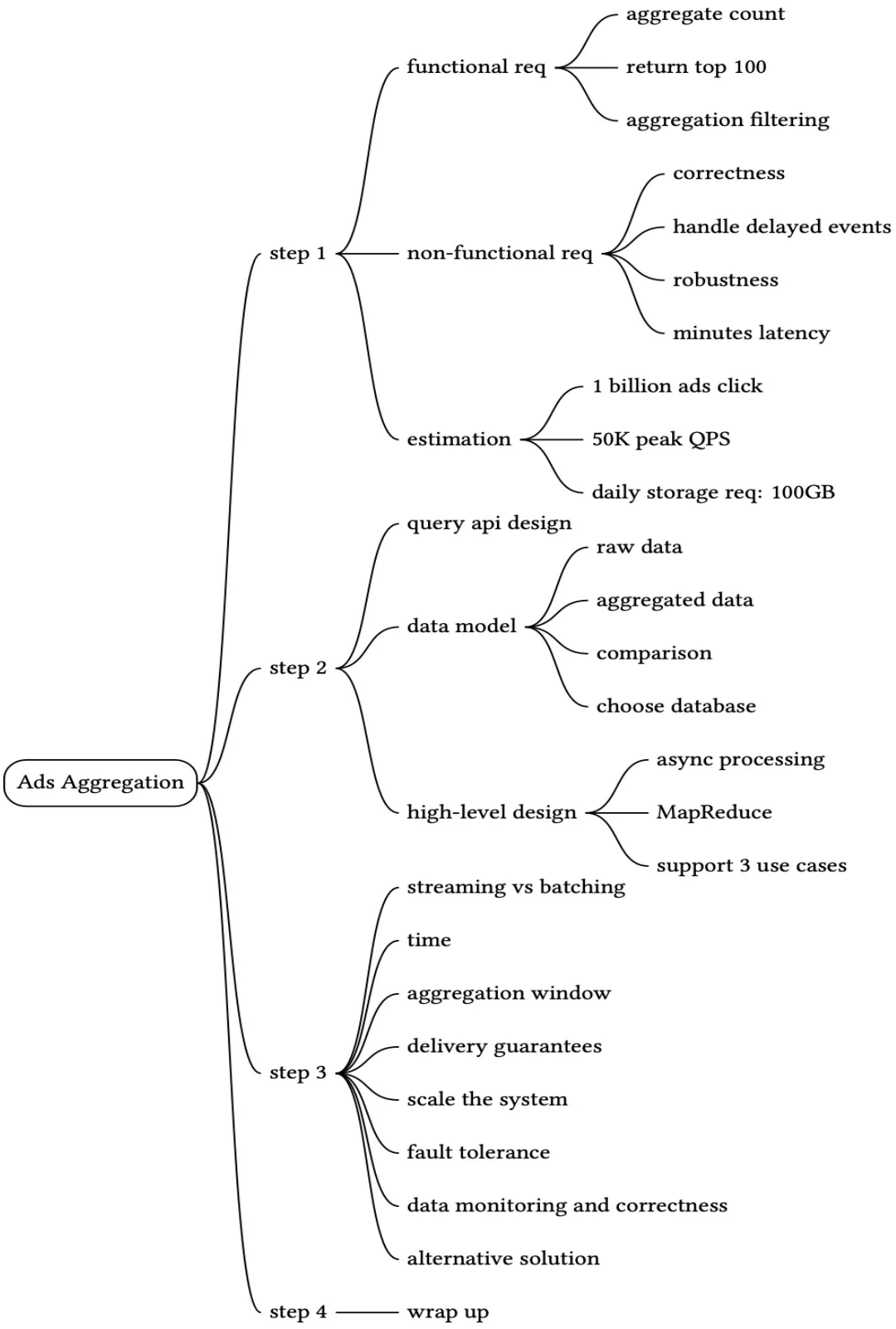
- **Hive**
    - Store ad click data (raw data).
- **ElasticSearch**
    - Enable faster queries.
- **ClickHouse**
    - OLAP database to store aggregated data.

Key points

- Store both raw data and aggregated data, raw data for debugging and backup, aggregated data for fast queries.

- Data aggregation service uses MapReduce framework and break down the problem into Map/Aggregate/Reduce nodes.
- Need to recalculate the raw data and generate new aggregated data when we discover a major bug.
- Should use event time rather than processing time.

Summary



References

- [System Design Interview – An insider's guide | Ad Click Event Aggregation]

2. Airline Management System

An Airline Management System is a managerial software which targets to control all operations of an airline. Airlines provide transport services for their passengers. They carry or hire aircraft for this purpose. All operations of an airline company are controlled by their airline management system.

This system involves the scheduling of flights, air ticket reservations, flight cancellations, customer support, and staff management. Daily flights updates can also be retrieved by using the system.

System Requirements

We will focus on the following set of requirements while designing the Airline Management System:

1. Customers should be able to search for flights for a given date and source/destination airport.
2. Customers should be able to reserve a ticket for any scheduled flight. Customers can also build a multi-flight itinerary.
3. Users of the system can check flight schedules, their departure time, available seats, arrival time, and other flight details.
4. Customers can make reservations for multiple passengers under one itinerary.
5. Only the admin of the system can add new aircrafts, flights, and flight schedules. Admin can cancel any pre-scheduled flight (all stakeholders will be notified).
6. Customers can cancel their reservation and itinerary.
7. The system should be able to handle the assignment of pilots and crew members to flights.
8. The system should be able to handle payments for reservations.
9. The system should be able to send notifications to customers whenever a reservation is made/modified or there is an update for their flights.

Use Case Diagram

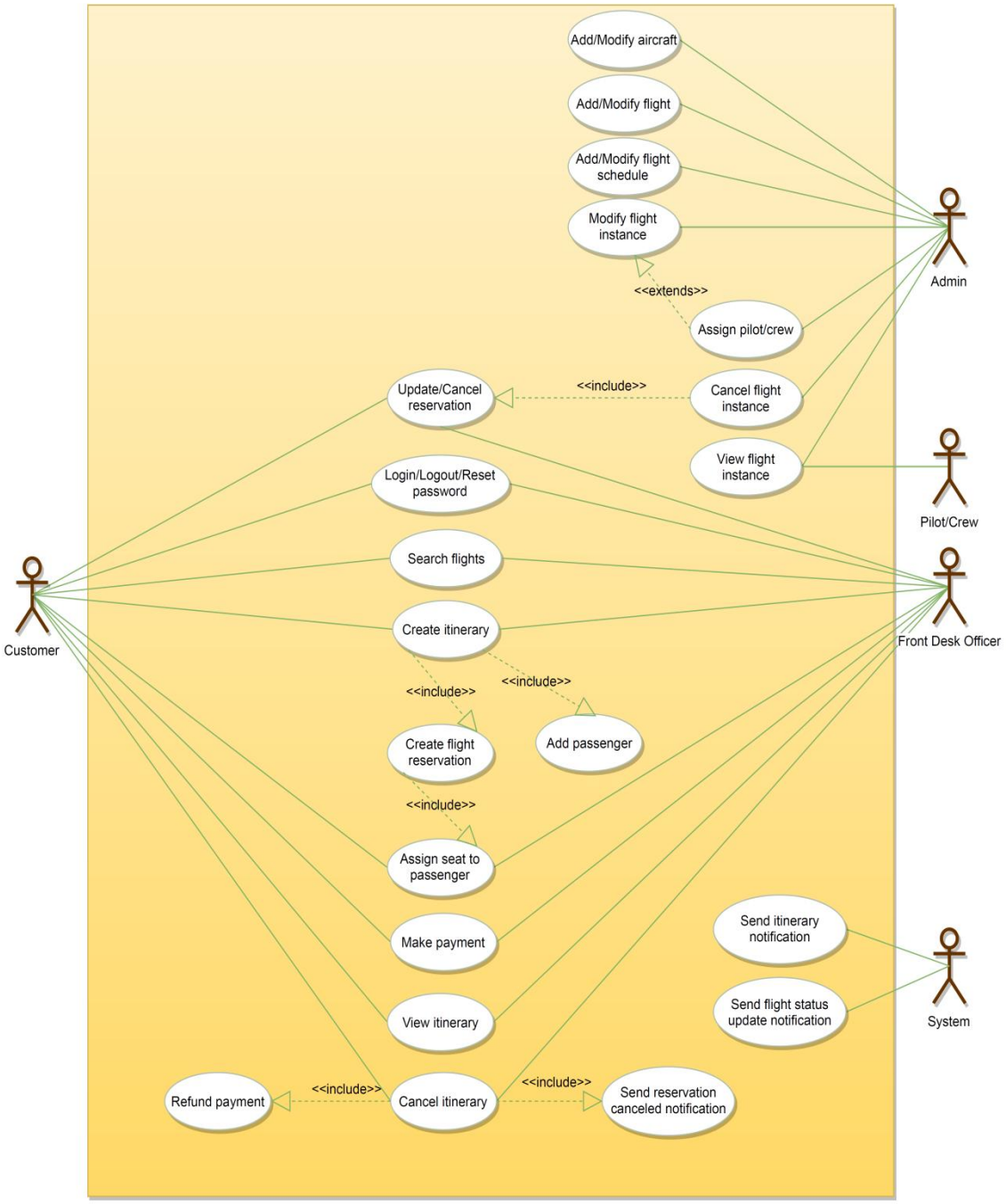
We have five main Actors in our system:

- **Admin**: Responsible for adding new flights and their schedules, canceling any flight, maintaining staff-related work, etc.
- **Front desk officer**: Will be able to reserve/cancel tickets.
- **Customer**: Can view flight schedule, reserve and cancel tickets.
- **Pilot/Crew**: Can view their assigned flights and their schedules.
- **System**: Mainly responsible for sending notifications regarding itinerary changes, flight status updates, etc.

Here are the top use cases of the Airline Management System:

- **Search Flights**: To search the flight schedule to find flights for a suitable date and time.
- **Create/Modify/View reservation**: To reserve a ticket, cancel it, or view details about the flight or ticket.
- **Assign seats to passengers**: To assign seats to passengers for a flight instance with their reservation.
- **Make payment for a reservation**: To pay for the reservation.
- **Update flight schedule**: To make changes in the flight schedule, and to add or remove any flight.
- **Assign pilots and crew**: To assign pilots and crews to flights.

Here is the use case diagram of our Airline Management System:

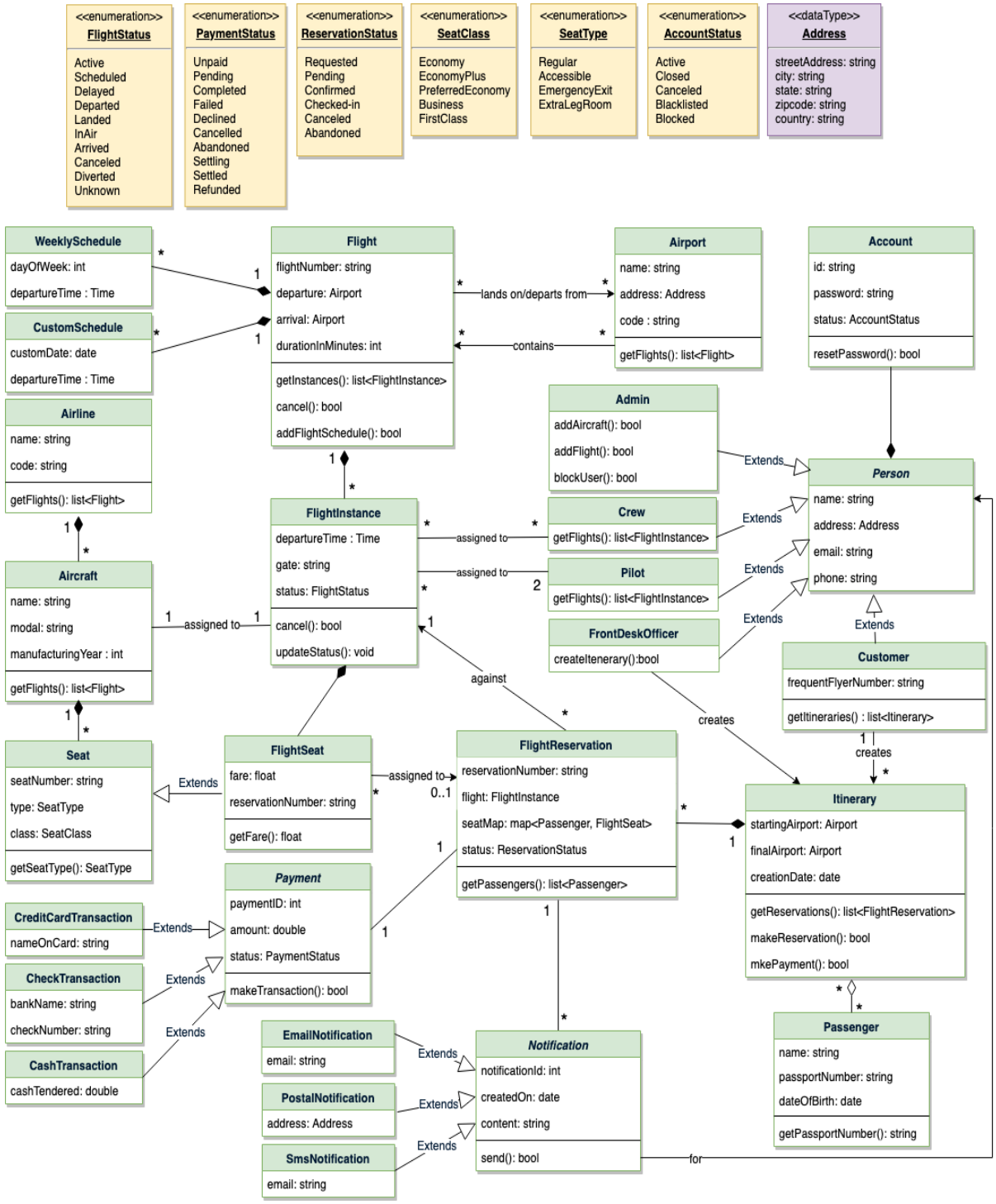


Use Case Diagram for Airline Management System

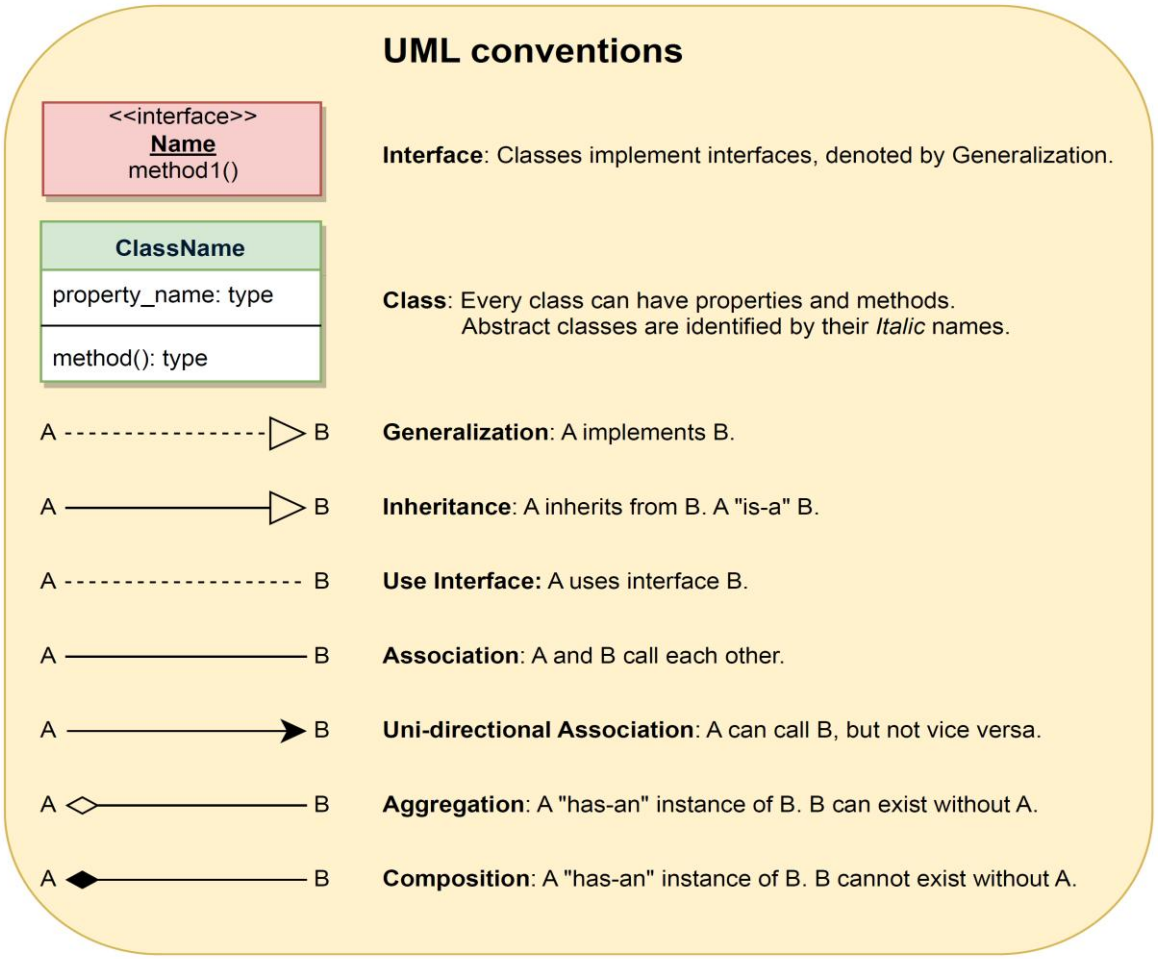
Class Diagram

Here are the main classes of our Airline Management System:

- **Airline**: The main part of the organization for which this software has been designed. It has attributes like 'name' and an airline code to distinguish the airline from other airlines.
- **Airport**: Each airline operates out of different airports. Each airport has a name, address, and a unique code.
- **Aircraft**: Airlines own or hire aircraft to carry out their flights. Each aircraft has attributes like name, model, manufacturing year, etc.
- **Flight**: The main entity of the system. Each flight will have a flight number, departure and arrival airport, assigned aircraft, etc.
- **FlightInstance**: Each flight can have multiple occurrences; each occurrence will be considered a flight instance in our system. For example, if a British Airways flight from London to Tokyo (flight number: BA212) occurs twice a week, each of these occurrences will be considered a separate flight instance in our system.
- **WeeklySchedule and CustomSchedule**: Flights can have multiple schedules and each schedule will create a flight instance.
- **FlightReservation**: A reservation is made against a flight instance and has attributes like a unique reservation number, list of passengers and their assigned seats, reservation status, etc.
- **Itinerary**: An itinerary can have multiple flights.
- **FlightSeat**: This class will represent all seats of an aircraft assigned to a specific flight instance. All reservations of this flight instance will assign seats to passengers through this class.
- **Payment**: Will be responsible for collecting payments from customers.
- **Notification**: This class will be responsible for sending notifications for flight reservations, flight status update, etc.



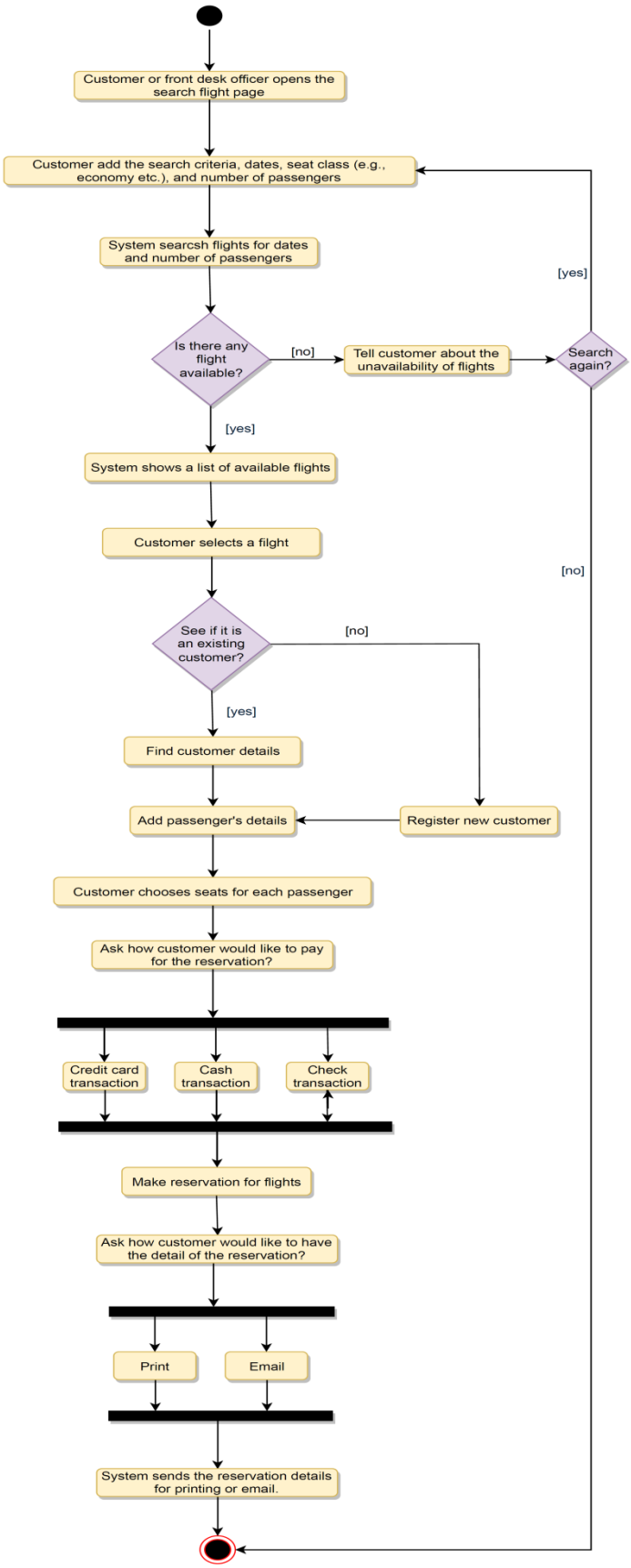
Airline Management System Class Diagram



Class Diagram for Airline Management System

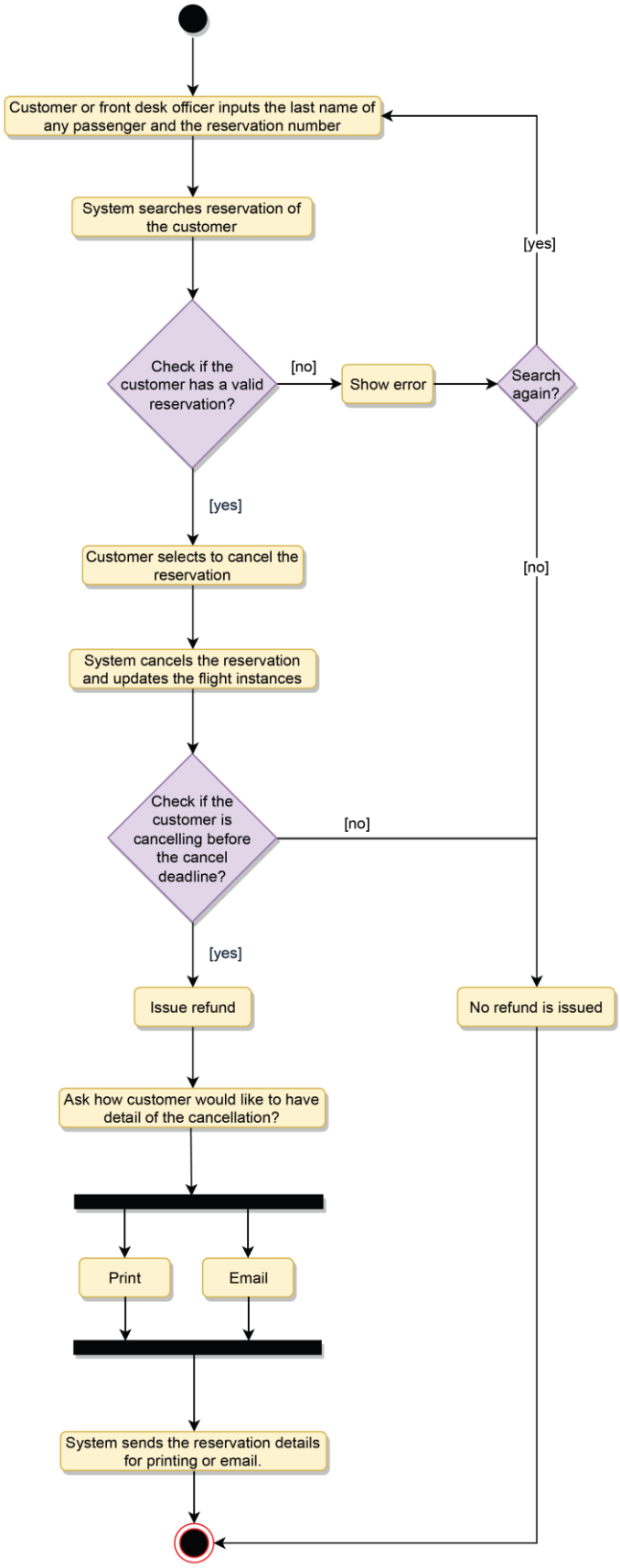
Activity Diagrams

Reserve a ticket: Any customer can perform this activity. Here are the steps to reserve a ticket:



Airline Management System Reserve Ticket Activity Diagram

Cancel a reservation: Any customer can perform this activity. Here are the set of steps to cancel a reservation:



Airline Management System Cancel Reservation Activity Diagram

Code

Here is the code for major classes.

Enums and Constants: Here are the required enums, data types, and constants:

```
import java.util.List;

enum FlightStatus {
   ACTIVE(1), SCHEDULED(2), DELAYED(3), DEPARTED(4), LANDED(5), IN_AIR(6), ARRIVED(7), CANCELLED(8), DIVERTED(9), UNKNOWN(10);

   private int value;

   FlightStatus(int value) {
      this.value = value;
   }

   public int getValue() {
      return value;
   }
}

enum PaymentStatus {
   UNPAID(1), PENDING(2), COMPLETED(3), FILLED(4), DECLINED(5), CANCELLED(6), ABANDONED(7), SETTLING(8), SETTLED(9), REFUNDED(10);
```

```
    private int value;

    PaymentStatus(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

enum ReservationStatus {
    REQUESTED(1), PENDING(2), CONFIRMED(3), CHECKED_IN(4), CANCELLED(5), ABANDONED(6);

    private int value;

    ReservationStatus(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
```

```
    }
}

enum SeatClass {
    ECONOMY(1), ECONOMY_PLUS(2), PREFERRED_ECONOMY(3), BUSINESS(4),
FIRST_CLASS(5);

    private int value;

    SeatClass(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

enum SeatType {
    REGULAR(1), ACCESSIBLE(2), EMERGENCY_EXIT(3), EXTRA_LEG_ROOM(4);

    private int value;

    SeatType(int value) {
```

```
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

enum AccountStatus {
    ACTIVE(1), CLOSED(2), CANCELED(3), BLACKLISTED(4), BLOCKED(5);

    private int value;

    AccountStatus(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

class Address {
```

```
    private String street;
    private String city;
    private String state;
    private String zipCode;
    private String country;

    public Address(String street, String city, String state, String zipCode, String country)
{
        this.street = street;
        this.city = city;
        this.state = state;
        this.zipCode = zipCode;
        this.country = country;
    }

    // Getter methods
}

class Account {
    private String id;
    private String password;
    private AccountStatus status;

    public Account(String id, String password, AccountStatus status) {
```

```
        this.id = id;
        this.password = password;
        this.status = status;
    }

    public void resetPassword() {
        // Implement password reset functionality
    }
}

abstract class Person {
    private String name;
    private Address address;
    private String email;
    private String phone;
    private Account account;

    public Person(String name, Address address, String email, String phone, Account account) {
        this.name = name;
        this.address = address;
        this.email = email;
        this.phone = phone;
        this.account = account;
```

```
    }
}

class Customer extends Person {
    private String frequentFlyerNumber;

    public Customer(String frequentFlyerNumber) {
        super("", null, "", "", null);
        this.frequentFlyerNumber = frequentFlyerNumber;
    }

    public void getItineraries() {
        // Implement method to fetch itineraries
    }
}

class Passenger {
    private String name;
    private String passportNumber;
    private String dateOfBirth;

    public Passenger(String name, String passportNumber, String dateOfBirth) {
        this.name = name;
```

```
        this.passportNumber = passportNumber;
        this.dateOfBirth = dateOfBirth;
    }

    public String getPassportNumber() {
        return passportNumber;
    }
}

class Airport {
    private String name;
    private Address address;
    private String code;

    public Airport(String name, Address address, String code) {
        this.name = name;
        this.address = address;
        this.code = code;
    }

    public void getFlights() {
        // Implement method to get flights
    }
```

```
}

class Aircraft {
    private String name;
    private String model;
    private int manufacturingYear;
    private List<Seat> seats;

    public Aircraft(String name, String model, int manufacturingYear) {
        this.name = name;
        this.model = model;
        this.manufacturingYear = manufacturingYear;
    }

    public void getFlights() {
        // Implement method to get flights
    }
}

class Seat {
    private String seatNumber;
    private SeatType type;
    private SeatClass seatClass;
```

```
    public Seat(String seatNumber, SeatType type, SeatClass seatClass) {
        this.seatNumber = seatNumber;
        this.type = type;
        this.seatClass = seatClass;
    }
}

class FlightSeat extends Seat {
    private double fare;

    public FlightSeat(String seatNumber, SeatType type, SeatClass seatClass, double fare) {
        super(seatNumber, type, seatClass);
        this.fare = fare;
    }

    public double getFare() {
        return fare;
    }
}

class WeeklySchedule {
    private String dayOfWeek;
```

```
    private String departureTime;

    public WeeklySchedule(String dayOfWeek, String departureTime) {
        this.dayOfWeek = dayOfWeek;
        this.departureTime = departureTime;
    }
}

class CustomSchedule {
    private String customDate;
    private String departureTime;

    public CustomSchedule(String customDate, String departureTime) {
        this.customDate = customDate;
        this.departureTime = departureTime;
    }
}

class Flight {
    private String flightNumber;
    private String departure;
    private String arrival;
    private int durationInMinutes;
```

```
    private List<WeeklySchedule> weeklySchedules;
    private List<CustomSchedule> customSchedules;
    private List<FlightInstance> flightInstances;

    public Flight(String flightNumber, String departure, String arrival, int durationInMinutes) {
        this.flightNumber = flightNumber;
        this.departure = departure;
        this.arrival = arrival;
        this.durationInMinutes = durationInMinutes;
    }
}

class FlightInstance {
    private String departureTime;
    private String gate;
    private FlightStatus status;
    private Aircraft aircraft;

    public FlightInstance(String departureTime, String gate, FlightStatus status, Aircraft aircraft) {
        this.departureTime = departureTime;
        this.gate = gate;
        this.status = status;
```

```
        this.aircraft = aircraft;
    }

    public void cancel() {
        // Implement method to cancel flight instance
    }

    public void updateStatus(FlightStatus status) {
        // Implement method to update flight status
    }
}

class FlightReservation {
    private String reservationNumber;
    private Flight flight;
    private List<Seat> seatMap;
    private String creationDate;
    private ReservationStatus status;

    public FlightReservation(String reservationNumber, Flight flight, List<Seat>
seatMap, String creationDate, ReservationStatus status) {
        this.reservationNumber = reservationNumber;
        this.flight = flight;
        this.seatMap = seatMap;
```

```
        this.creationDate = creationDate;
        this.status = status;
    }

    public void fetchReservationDetails(String reservationNumber) {
        // Implement method to fetch reservation details
    }

    public void getPassengers() {
        // Implement method to fetch passengers
    }
}

class Itinerary {
    private String customerId;
    private Airport startingAirport;
    private Airport finalAirport;
    private String creationDate;
    private List<FlightReservation> reservations;

    public Itinerary(String customerId, Airport startingAirport, Airport finalAirport,
String creationDate) {
        this.customerId = customerId;
        this.startingAirport = startingAirport;
```

```
        this.finalAirport = finalAirport;
        this.creationDate = creationDate;
    }

    public void getReservations() {
        // Implement method to get reservations
    }

    public void makeReservation() {
        // Implement method to make a reservation
    }

    public void makePayment() {
        // Implement method to make payment
    }
}
```