Airport Management System — Test Description

Goal

Design a small airport management system using JavaScript.
The system must use **OOP principles** (classes, inheritance, encapsulation and polymorphism) and include **unit tests**.

Students must decide **how to design the class structure and inheritance**, based on the requirements below. Detail your choices in your readme.

Entities and Features

Flight

Each flight must have:

- Flight name
- Airline
- Flight number
- Maximum number of passengers
- Regular ticket price
- VIP ticket price
- Tickets list
    - The list size must match the maximum number of passengers
    - At the start, tickets have **no owner name**

Passenger Types

Student Passenger

Properties:

- Name
- ID number
- Amount of money
- School / university name

Behavior:

- Can buy a ticket with **10% discount**
- If the passenger does not have enough money → return `false`
- **Cannot get a discount on FIRST CLASS (VIP) tickets**
    - VIP tickets are paid at full price

Regular Passenger

Properties:

- Name
- ID number
- Amount of money
- Workplace
- Knows an airport employee (true / false)

Behavior:

- Cannot buy tickets if there is not enough money → return `false`
- If the passenger knows an airport employee:
    - 20% discount on regular tickets
    - 15% discount on VIP tickets
- If not:
    - No discount

Ticket Types

Regular Ticket

- Random ticket number
- Price
- Owner name

VIP Ticket

- Ticket number
- Price
- Owner name
- Benefits list:
    - Free alcohol
    - Free food
    - Hot towels

Airport System

You must create an **Airport class** that:

- Creates **3 flights (set/ask or randomize needed data)**
- For each flight:
    - 90% regular tickets
    - 10% VIP tickets
- Stores all flights inside the airport

Running requirements:

1. create an instance of an airport.
2. create 2 instances of a passenger - student and regular.
3. buy a regular ticket for one of the flights for the regular passenger
4. buy a vip ticket for the student passenger for one of the flights.

Unit Testing Requirements

You must write **unit tests** for the following flow:

Ticket Purchase

Test two cases when creating a passenger and trying to buy a ticket for a flight:

1. Not enough money
    - The method returns `false`
2. Enough money
    - The ticket owner name matches the passenger name
    - The passenger's money is reduced by the correct ticket price

make sure the tests run by running: node --test

⭐ Bonus Task (Optional) — New Entity + Unit Tests

New Entity: Baggage

Baggage Properties

- Baggage ID
- Owner ID (passenger ID)
- Weight (kg)
- Type: `carry_on` or `checked`

Baggage Rules

- Baggage can be added **only after the passenger bought a ticket** for the flight
- Each flight has a **maximum total baggage weight**
- If adding baggage exceeds this limit → return `false`

Baggage Fees

Student Passenger

- `carry_on`: free up to 7kg, otherwise +20
- `checked`: +30

Regular Passenger

- Knows airport employee:
    - `carry_on`: free up to 10kg, otherwise +10
    - `checked`: +20

- Does not know employee:
    - `carry_on`: free up to 7kg, otherwise +20
    - `checked`: +40

If the passenger does not have enough money → return `false`
On success:

- Passenger money is reduced
- Baggage is added to the flight

Bonus Unit Tests (Required if Bonus is Done)

Write unit tests for:

1. **No Ticket → No Baggage**
    - Adding baggage without a ticket returns `false`
2. **Fee Calculation**
    - Verify correct fee and money update for:
        - Student carry-on
        - Regular passenger with employee connection
3. **Flight Weight Limit**
    - Adding baggage that exceeds the flight limit returns `false`

Grading Table (100 Points + Bonus)

Area	Points

OOP Structure	25
Flight & Ticket Behavior	25
Passenger Payment Logic	20
clean code	10
Unit Tests	20
Bonus: Baggage Entity	**+10**