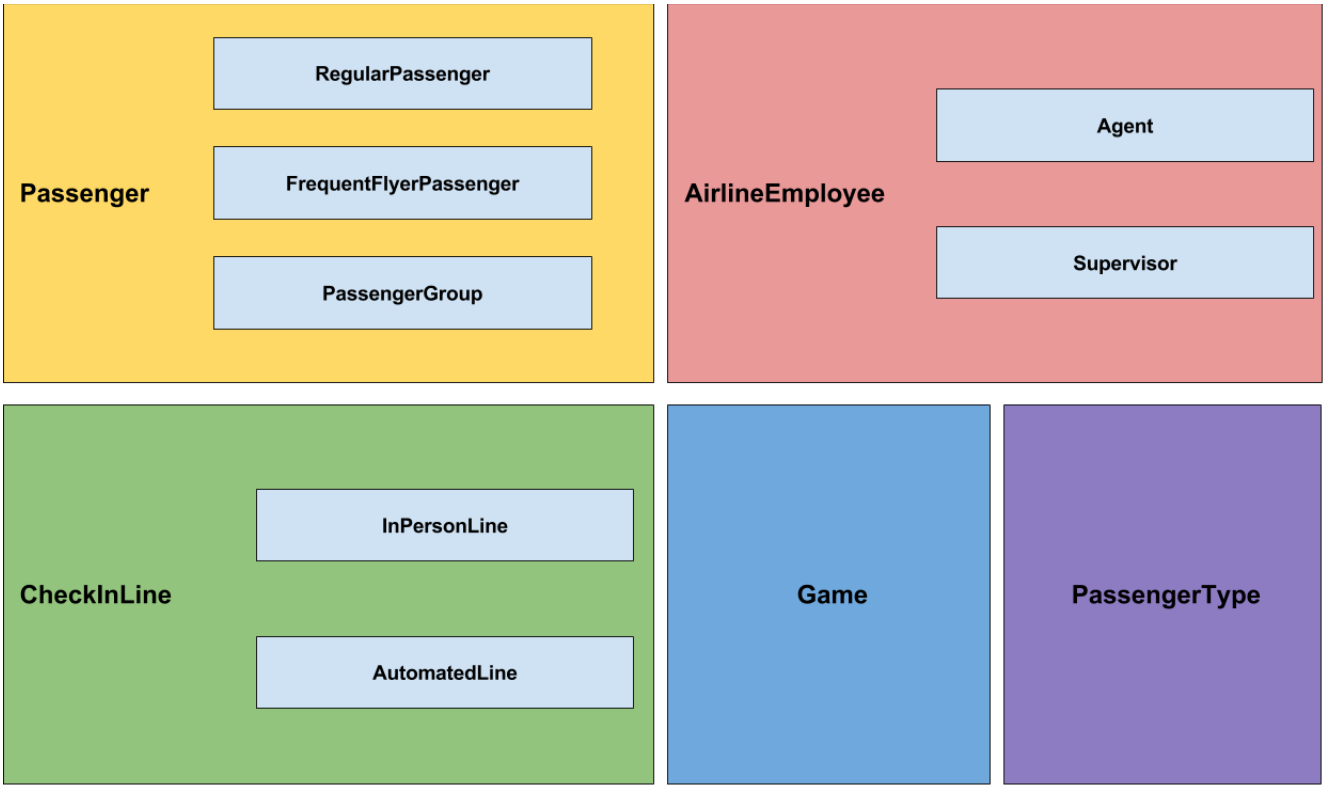
AirVille Design Document

Israel Hill



Passenger → Abstract Class

This class represents passengers in the AirVille game. Each passenger has a process time associated with it which provides the time it will take before this passenger will be de-queued from the line.

Fields:

- private int processTime

Methods:

- int getProcessTime(): returns the process time
- void setProcessTime(): set the process time
- boolean isSlow(): return whether or not this passenger is one of the slow types
- int getTimeBusy(): return the process time of this passenger

Constructor:

- takes EnumSet of PassengerType
- Passenger(EnumSet<PassengerType>)

RegularPassenger → extends Passenger

This class extends the Passenger class and represents normal (non-frequent flyers) passengers.

Fields: none

Constructor:

- super(EnumSet<PassengerType>)
- RegularPassenger(EnumSet<PassengerSet>)

FrequentFlyerPassenger → extends Passenger

This class extends the Passenger class and represents frequent flyer passengers. Frequent flyers are processed before Regular passengers in InPersonLines.

Fields:

- none

Constructor:

- super(EnumSet<PassengerType>)
- FrequentFlyerPassenger(EnumSet<PassengerType>)

PassengerGroup → extends Passenger

PassengerGroups extends Passenger and represents groups of passengers. Passenger groups can be of size 2-20. Passenger groups are processed slower than single passengers but faster than processing each member of the group individually.

Fields:

Defines how many members are in the group

- private numMembers

Methods:

- int getNumMembers(): return the number of member in the group

Constructor:

- call super(EnumSet<PassengerType>)
- PassengerGroup(EnumSet, numMembers)

AirlineEmployee → Abstract Class

This is an abstract class that represents airline employees. Airline employees work at CheckInLines to process passengers.

Fields:

- private boolean isBusy

Methods:

- boolean isBusy(): return whether or not this employee is busy
- abstract void setline(): assign this worker to a line
- abstract CheckInLine getLine(): return the line this employee is currently assigned to
- int getTimeBusy(Passenger): passenger's process time determines the worker's time busy.

Constructor:

- default constructor

Agent → extends AirlineEmployee

This class extends AirLineEmployee and represents an airline agent. Agents can work at InPersonLines or float between a predefined number of automated lines. Agents do not process passengers as fast as supervisors.

Fields:

- private final lineLimit (number of lines agent can float between)

- private array of CheckInLines

Methods:

- void addAutomatedLine(AutomatedLine): assign another line to this employee
- void removeAutomatedLine(AutomatedLine): remove a line from this employee

Constructor:

- Agent(int lineLimit)
- call super()

Supervisor → extends AirlineEmployee

This class extends AirLineEmployee and represents airline employees. Supervisors can process passengers faster than agents; thus they have a constant called "process time decrease" which is a constant that can be subtracted from the total processing time of a passenger. Supervisors can float between any number of automated lines.

Fields:

- private final int PROCESS_TIME_DECREASE
- private array of automated lines

Methods:

- override abstract methods in parent class
- void move(CheckInLine1, CheckInLine2): move a supervisor from line 1 to line 2

Constructor:

- calls super()
- Supervisor()

PassengerType → Enum

Passengers can be of type: excess baggage, re-routed, overbooked, or normal.
The first three types will be processed slower than normal passengers.

Fields:

- private String passengerType

Methods:

- getPassengerType(): returns the string representation of the enumerated type.

Constructor:

- takes a String passengerType
- PassengerType(passengerType)

CheckInLine → Abstract Class

This is an abstract class that represents check in lines at the airport. Passengers queue at these lines, thus the supporting data structure for extending types of this class is a queue.

Fields:

These fields let the game know what workers are currently attending what lines.

- private Agent
- private Supervisor

Methods:

If the line has an airline employee, you can process the next employee

- abstract method processNextPassenger(): process and de-queue next passenger
- abstract method addPassenger(): add passenger to line
- boolean hasAgent(): check if there is an agent at this line
- boolean hasSupervisor(): check if there is a supervisor at this line
- getSupervisor(): return the supervisor at this line
- getAgent(): return the agent at this line
- setSupervisor(): set a supervisor to this line
- setAgent(): set an agent to this line

Constructor:
- set agent and supervisor to null
- CheckInLine

InPersonLine → extends CheckInLine

This is a class that extends CheckInLine and represents lines that require a worker to process passengers.
Fields:
- private Queue of type FrequentFlyerPassenger
- private Queue of type RegularPassenger

Methods:
- override processNextPassenger (pop passengers from the correct queue, frequent flyers first)

Add passenger to correct queue based on their type
- override addFrequentFlyerPassenger()

Constructor:
- super()
- InPersonLine()

AutomatedLine → extends CheckInLine

This class extends CheckInLine and represents an automated line that requires a worker to be "floating over it" to process passengers.
Fields:
- private Queue of type Passenger

Methods:
- override processNextPassenger (pop passenger from front of queue)
- override addPassenger

Constructor:
- super()
- AutomatedLine()

Game → Class

The game class represents in game processes. Here is where all the instances of entities are stored so they can be referenced by the other development teams. In addition, this class keeps track of the players diamonds and offers methods that will allow other developers to tie in code for purchasing more agents, supervisors, lines, and diamonds from Zonga.
Fields:

- private int Diamonds
- private array of Automated lines
- private array of in person lines
- private array of Agents
- private array of Supervisors

Methods:
Add a diamond
- void addDiamond()

Adds an additional agent
- void redeemAgent()

Adds an additional supervisor
- void redeemSupervisor()

Add another automated line
- void redeemAutomatedLine()

Add another in person line
- void redeemInPersonLine()

Check if the player has diamonds before he/she can redeem a new entity
- boolean hasDiamonds()

Adds line to corresponding array
- void addInPersonLine()

Adds line to corresponding array
- void addAutomatedLine()

Adds agent to array of agents
- void addAgent()

Adds supervisor to array of supervisors
- void addSupervisor()

return array of in person lines
- array getInPersonLines()

return array of automated lines
- array getAutomatedLines()

return array of agents
- array getAgents()

return array of supervisors
- array getSupervisors()

Constructor:
- Game()
- Initialize the game to have one agent and one check in line.

Error Handling:

Because this is a game, I want it to be as robust as possible. To do this, I will try to handle errors locally when possible. For example, if someone tries to create a group that is less than size 2, I will default it to 2. If they try to create a group that is greater than size 20, I will default it to 20. I plan on having one error class that handles invalid operations. Some invalid operations are: exceeding an agent's automated line limit, moving an agent/supervisor to a line that already has one, etc. This error will extend Exception (as to not halt the program), create an error message, and deny completing the action if it is not possible. One example of this is the move method in Supervisor. If you try to move a supervisor to a line that already has a supervisor, I will create a new exception with the proper error message and deny moving the supervisor.

Testing:

To test my program, I will test my methods in isolation to ensure they behave as I wish. I will write standard JUnit tests to test each of my classes. I will use the testing techniques we learned in the previous assignments to make sure I properly cover my bases. I will also include a stress test that will simulate the creation of many entities over time to ensure that the program still works properly with a large set of data.