EECS 293
Software Craftsmanship
2015 Fall Semester

Programming Assignment 11

Due at the beginning of your discussion session on

November 10-13, 2015

Reading

- Section 6.2 (skip "Present a consistent level of abstraction in the class interface" and "Provide services in pairs with their opposites") in Code Complete
- Section 6.4 in Code Complete
- Items 13, 14, 15 in Effective Java
- Section 19.6 in Code Complete

Programming

You have just been hired as the lead software architect at Zonga, the famous game company, to design and develop a new game called AirVille. Congratulations!

Your first assignment is to develop the object-oriented architecture of AirVille. As you have learned in Section 5.1, you fully expect the design process to be sloppy, non-deterministic, based on heuristics, and iterative (and possibly very frustrating). However, you also shoot for a product that is tidy, clearly addresses the relevant trade-offs, priorities, and restrictions: in other words, it will be a beautiful architecture.

Requirements

On your first day, you are debriefed on the new game, and told about its main concept and requirements. AirVille is set at the check-in counters in an airport. The player manages the counters and airline agents who are responsible for checking in passengers into flights. The player will get points and rewards as he checks in passengers. Faster progress in the game depends on the efficiency of the check-in process (and on how much help the player is willing to buy from Zonga).

Passengers queue at check-in lines. There are two main types of lines: in-person and automated. In-person lines can be regular or frequent-flyer: the frequent-flyer passengers are always checked in before the regular passengers. Each agent sits at his counter and processes passengers in turn, always calling the first frequent-flyer passenger if any, and otherwise calling the first regular passenger. In contrast with in-person lines, automated lines comes into a single category for both regular and frequent-flyer passengers. Since there is significantly less human interaction with automated check-in, a single agent can float among a pre-set number of automated check-in stations.

In addition to regular airline agents, there are also supervisors. A supervisor can float between check-in lines. When the supervisor joins an agent at his counter or at an automated station, passengers are processed faster than if the regular agent were on his own. Every player starts with a single supervisor, but you can pay Zonga to provide more supervisors. The player can shift the supervisor from station to station, or keep him on the sideline to anticipate future emergencies.

In AirVille, the player has the ability of moving the supervisor. As he accumulates more points, he can hire more agents and supervisors and create new check-in lines. However, passengers will be created automatically over the course of the game, and new passengers are constantly arriving at the check-in counters.

Passengers autonomously queue at a check-in line. Occasionally, they will have to be redirected from automated to in-person check-in.

Passengers tend to have different requests and their check-in can take a different and unpredictable length of time (but less time with a supervisor than with a regular agent). Some passengers (but not many) are also able to check in without assistance at the automated stations. Conversely, if a passenger is taking too long, the player has the option of moving the supervisor to that check-in line, to buy more supervisors from Zonga, or to create more check-in lines (if the player has enough points). There are some particularly slow classes of passengers:

- Passengers with excess baggage take more time due to the need to double check the baggage weight and then purchase the additional fare.
- Passengers who have been repeatedly re-routed due to flight delays and cancellations. Most of these passengers can only be helped by the supervisor and meanwhile they block the whole queue behind them.
- If the plane is overbooked, some passengers will be in line to get off the plane and take an airline bonus.

These types of slow passengers will have to be processed in-person, regardless of whether they initially queued up in the in-person or automated line.

Passengers come as either single passengers or a part of a group. A passenger group must be checked in together in the same line. Although a passenger group takes more time than a single passenger does, the group takes less time than if each member had to be checked-in individually. If the player processes a large group of 10+ passengers in less than five minutes, the player will get a diamond. Diamonds can also be bought from Zonga, and can be redeemed for check-in lines, agents, or supervisors.

Along with the requirements for your project, you are also given an overview of other Zonga departments. The User Interface team is responsible for the graphical user interface and animation in the AirVille game. Although your group will eventually work with the User Interface team, you do not have to worry about user input or graphics and should focus instead on the core AirVille classes. The Real-Time team is in charge of data structures and algorithms for simulation and gaming, which involves scheduling events. You do not have to worry about the specifics of the game simulation, but your code should be able to work with the Real-Time team's simulation.

Along with the other project requirements, in order for the core classes to work with the User Interface and Real-time teams, these classes should provide:

- Information on which entities are present in the game
- Whether or not entities are busy
- How long entities will be busy (if they are busy)

The core AirVille classes are not responsible for stalling the game or for scheduling events.

Design

After your initial debrief, your assumptions turned out to be valid: the requirements have a lot of latitude (to put it mildly), the whole design process looks especially wicked, but you are also determined to create the best object-oriented architecture. Time to get to work!

Create a design document that describes your architectural decisions. Your document can optionally contain sketches and diagrams of your architecture. Your objective is that your design document should be used as a blueprint for the implementation. You can define classes or interfaces as you see fit. Commonalities among classes or interfaces should be expressed through inheritance or containment. For each class or interface, you should describe at least the abstraction it captures, its position in the inheritance hierarchy, the signature of its constructors and public methods, its private data structures (if any), and the pseudo-code of any complicated method.

You can outline your architecture for error handling and your approach to testing.

Discussion Guidelines

The discussion will focus on class design.

Submission

Submit design documents that describes your architectural decisions. No implementation is required: you will implement your design in the next programming assignments. This assignment can be submitted on Blackboard.