Craig Lombardo

Project 1

3/814

1. Introduction:

One of the biggest problems that business owners have today is managing the customer demand they receive. This project focuses specifically on the restaurant industry and the difficult task they are set with of seating every, or most of the customers, they are presented with. More specifically managing the wait list, particularly at popular restaurants. This task is simple at first, i.e. when the restaurant initially opens and all of the tables are open; however, once those tables are filled, things become a little trickier. Managing hundreds and hundreds of customers is never easy; managing hundreds and hundreds of hungry customers is even harder. For the purposes of this lab we are experimentally testing the task of seating arrangements for popular restaurants such as Ippudo in New York City. Ippudo is a perfect model, as the restaurant does not take reservations, thus making the task of efficient seating a necessity. Customers show up and put their name on a wait list and they are called when a table is available. The goal of this project is to come up with a seating strategy that will maximize revenue and minimize wait time. It is also important to ensure a correct wait time approximation. It is not fair for a group to be told they will be seated for 5 minutes and then wait for 5 hours, that is just poor business and there will be no returning customers. The goal of the project is also to deal with object-oriented programming and construct a program that will treat customers as objects that can interact with the environment they are put in. For this lab I made a few assumptions. The first being that the table structure that we were given is beneficial for the project we are trying to complete. I also had to assume, given I don’t have anyone else’s data to go based off of, that the seating strategies I made are the best (this includes seating and wait list handling).

2. Approach:

For the purposes of this project I decided to break things up into multiple parts, sort of implementing a divide and conquer strategy so as to ensure that no one class was doing too much. There were two reasons behind this; the first being that I wanted to limit the potential for problems (although it does not change the project it makes the logic and debugging easier), the second being I wanted my program to run more like a machine and less like a piece of code. For the purposes of the approach I will be discussing the parent classes and the purpose of each of those classes, as all of their children extend their parent and thus inherit all logic that will be explained and any class that is neither a parent nor child. The first class is the Restaurant class; this class serves as the actual “restaurant” we are implementing and will be filled with varieties of seating options and customers. The restaurant will keep track of things like the total customer count, the seating and waiting list preference (which when varied supply me with different strategies), lists that indicate who is on what list. Most importantly, the restaurant brings all of the other classes together as a sort of bridge between the classes. The next class is Seating, this class is the parent class for all seating types (i.e. bar, oval, booth, table) and handles customer seating, or attempted customer seating. The children of this class each control their own seating, each keeping track of the number of seats available and which seating positions are available for a given group type. The next class is the Customer class; this class is another parent class. When instantiated, as either a SmallGroup or LargeGroup, each group is assigned a random size (1-4 for small groups, 5-8 for large groups) and the type of group is tracked for later use in seating implementation. Each group also keeps track of pertinent information such as how many people are in the group, what table they are sitting at, what time they arrived/the wait time they were told (if applicable). Past this, the Customer class does not have much more functionality, it is intended to be like an actual group, keeping track of what it’s told; however, it doesn’t control implementation, it merely changes how the implementation of the restaurant is handled. There is a class called TimedEvents; which acts like the clock/hostess of the restaurant. The clock notifies the restaurant when a new group has arrived, small or large, and checks to see when a group has finished eating their meal/ready to leave. From here customers are either added or removed from the waiting list accordingly as well as seats the customers when the time is appropriate. There are also two random number generator classes, one is a RandomGaussian and the other is the Wheel. The RandomGaussian number generator generates random arrival and stay times for the two groups, based on a normal Gaussian distribution. The Wheel class generates random numbers, which will be used to give the random group sizes used. For all lists used, I used ArrayLists, even at the oval table; although there is not normally continuity I ensured that there will be a wrap around. There will be ArrayLists that will contain Customers i.e. waiting lists and seated lists, and ArrayLists that contain Integers such as available seating indexes.

3. Methods:

For the purposes of this project I decided to run things 10-15 times for a concrete result/ average. The data collected included how many customers were served, how many meals were given to those waiting past 2 am (4 free meals a person), how many people were still eating after 2 am, how many customers of each type came to the store front, and how accurate my time estimate was. The breakdown was as follows, five tables that held four people apiece, five booths holding eight people each, and oval that can hold 30 people (continuously around) and a bar that seats 20 people. That gives us 110 seats, with small groups arriving on average 20 seconds apart from each other (standard deviation of 10 seconds), staying on average 45 minutes (standard deviation of 10 minutes), large groups arrive on average 30 seconds apart from each other (standard deviation of 15 seconds), staying on average 60 minutest (standard deviation of 15 minutes). With this given data I created different scenarios, these scenarios ranged from table preferences for the two groups and seating strategies. Each strategy was tested and revamped several times before deciding whether or not it was a valid option.

4. Data and Analysis:

Given my experimental data, it can be seen that \_\_\_\_ was the most efficient seating strategy and \_\_\_\_ was the most efficient wait time handling strategy. These results were derived from my experimental runs and the averaging of the tests run. As shown by the plots (to be added in the final draft) it can be seen that \_\_\_\_\_ is the better strategy. With this strategy I gave out \_\_\_\_ less meals and had a max wait time of \_\_\_\_ compared to the other times of \_\_\_\_\_. This strategy, over time can make me/ my restaurant \_\_\_\_ more/ seat more people.

5. Conclusion

With everything being said and the data at hand, it can be seen that \_\_\_ is a superior seating strategy in comparison to the other experimental scenarios. It may not be serving thousands of groups in a night; however, given the parameters of the restaurant we were given, I am fairly confident that the strategies I have put forth are reasonably sound. Although there is no one perfect seating strategy for every restaurant I feel as though the program I have constructed allows for flexibility to be reused for a different restaurant with different table types and through adaptation of the seating strategies it may be used to test functionality and efficiency.

6. References:

"Java Platform SE 7." *Java Platform SE 7*. N.p., n.d. Web. 06 Mar. 2015.

Weiss, Mark Allen. *Data Structures & Problem Solving Using Java*. Boston: Pearson/Addison Wesley, 2010. Print.

https://moodle.lafayette.edu/pluginfile.php/151579/mod\_resource/content/1/writeup-guidelines.pdf

https://moodle.lafayette.edu/pluginfile.php/154315/mod\_resource/content/3/p1.pdf

http://www. javapractices.com/topic/TopicAction.do?Id=62.