Market Simulation Design Document

Maksym Turkot
03/28/2021

Project Goals

The goal of the project is to create a simulation of a market and calculate certain metrics based on multiple runs of different configurations of the simulation (different numbers of different types of stalls, and their transaction speed).

Determining time spent waiting in line:
Each Customer has an ArrayList that stores times they spent waiting in each line (including time making a purchase). Once the market closes after 210 mins (3.5 hrs), the Experiment runs for 15 more minutes to allow customers currently purchasing to finish their trade. After that the Experiment class gathers all values of ArrayLists “queueTimes” for each customer and finds the average value for each customer.

Determine the average line size at each stand:
Every minute (clock iteration) experiment gathers line length data from each stand. At the end, averages those values for each stall.

Determine the average length of time customers spend shopping:
Each Customer knows initially when they will enter the market, and record the minute (clock tick) when they leave. Then, they calculate the total time they spent in the market shopping (difference between the time entered and time departed). The Experiment class then gathers times spent in the market for each customer.

Using Gaussian Random Distribution:
- In the Customer class, their arrival at the market will be calculated as a distribution with a median of 20 minutes after the market has opened, but not less than 0.
- In the Customer, the grocery list is generated as a list of random values between 1 and 6. The list is never bigger than 6 and never smaller than 1.
- In the Stall class, the time each purchase takes place is be set by a configuration, and varied with the Gaussian distribution.

Create six very different configurations:
Configurations differ by the number of stalls for each product and the time it takes each stall to sell a product. Since the goal is to help the manager determine the number and type of stalls, these are the main and only configuration variables. There is a separate configuration file for each market configuration, and ConfigManager retrieves this data. After that, it provides it to Experiment.
Configurations are as follows:
1. One stall for each product, most are slow.
2. Two stalls for each product, most are slow.
3. One stall for each product, all are fast.
4. One stall for each product, half are slow and another half are fast.
5. Three products have two slow stalls, three have only one quick stall.
6. Different number of stalls for various products with different speeds.

Create simulations of stalls based on queues:
The structure of a market is as follows: Market creates Stall objects according to the configuration data. Each Stall has a queue associated with it. Each queue will consist of customers. Market creates customers based on three types with different shopping logic (Separate class for each).

Create simulations of customers using state machines:
Each of three types of customers has a unique implementation of a state machine behavior. First one simply goes through their list in order and visits stalls corresponding to next item. Second one searches for stall that has the good on the list and has the shortest queue. Third one simply chooses the first stall they find that has the desired good.

Program Organization

Class Experiment

This class requests configuration info from ConfigManager, instantiates market objects with corresponding parameters, runs each configuration five times with different seeds, sets the clock running for a given market, gathers simulation info and orders OutputManager to write performance logs and output files.

Fields:

- ArrayList<Integer> shoppingTimes - times customers spent shopping
- ArrayList<Integer> queueTimes - average times customers spent in queues
- ArrayList<Integer> queueLengths - average lengths of queues

Methods:

- static void run() - runs the simulation. It instantiates markets with requested data, runs each market config multiple times, gathers data and orders data output.
- String dataToCsV(ArrayList<Integer> list) - converts ArrayList data into CSV string; *TESTED by checking if list is correctly converted to CSV string*

Class ConfigManager

This class manages data retrieval from config files

Fields:

- Scanner fileScanner, lineScanner - scanners
- ArrayList<String> data - stings of config data
- ArrayList<Integer> stallNumbers - info about numbers of stalls
- ArrayList<Integer> stallSpeeds - info about speeds of stalls
- String[] numbers - strings of array numbers

- String[] speeds - strings of speed numbers

Methods:

- ArrayList<Integer> getStallNumbers() - getter
- ArrayList<Integer> getStallSpeeds() - getter
- void readStallInfo(String configName) - reads config files; *TESTED by checking if info is correctly read from the file*

Class OutputManager

This class manages output file creation.

Fields:

- PrintWriter fileWriter - fileWriter
- File outputFile - output file

Methods:

- void writeLog(String name, String data) - writes log for a given run with relevant data
- void writeOutput(String name, String data) - writes output for a given run with relevant data

Class Clock

This class implements a clock.

Fields:

- int time - global time
- String logString - info for a log file

Methods:

- void tick(Market market) - iterates the clock for a given market

Class Gaussian

This implements Gaussian random distribution randomizer.

Fields:

- Random rand - Random object

Methods:

- int getGaussian(double mean, int deviation, int lowerLim, int upperLim) - gets a number in Gaussian random distribution based on giver parameters; *TESTED by checking if Gaussian numbers are generated*

Class Market

This class initializes customer, stall, and gaussian objects with corresponding parameters received from the Experiment. It then runs the market actions by telling each customer to do next shopping step and stalls to do next selling step as the time goes by (the clock ticks). It also gathers necessary log info.

Fields:

- ArrayList<Stall> stalls - stalls of this market
- ArrayList<Customer> customers - customers of this market
- Gaussian gaussian - gaussian generator for this market
- int seed - randomizer seed for this market

Methods:

- ArrayList<Stall> getStalls() - getter
- ArrayList<Customer> getCustomers() - getter
- Gaussian getGaussian() - getter
- int getSeed() - getter
- void createStalls(ArrayList<Integer> stallNumbers, ArrayList<Integer> stallSpeed) - generates stalls based on passed parameters; *TESTED by checking if stalls are generated*
- void createCustomers() - generates customers; *TESTED by checking if customers are generated*
- void runMarket() - gets the market going by telling customers to shop and stalls to sell as clock ticks;
- String toString() - gathers and formats data for log file

Class Stall

This class constructs stall objects. It also manages the selling process, and moves customers to its queue upon request and removes them from it.

Fields:

- int stallCnt - stall counter

- ArrayList<Integer> queueLengths - list of lengths of queues
- Market market - market of this stall
- Queue queue - this stall's queue
- int id - this stall's id
- int productId - product this stall is selling
- int speed - time it takes this stall to sell goods
- int randTimeOfPurchase - time of purchase for unique instant

Methods:

- int getId() - getter
- int getProductId() - getter
- Queue getQueue() - getter
- void addToQueue(Customer customer) - adds customer to queue upon request; *TESTED by checking if customers are added*
- void removeFromQueue() - removes customer from queue; *TESTED by checking if customers are removed*
- void sell() - selling logic. Checks if queue is empty, checks if previous customer is finished purchasing, and starts selling to new customer
- sell() — this method will count through the time it takes to sell a product. Will test by running this method with different products of the stall.
- int getAvgQueueLength() - calculates average queue length for this stall
- int getMinQueueLength() - calculates minimum queue length for this stall
- int getMaxQueueLength() - calculates maximum queue length for this stall
- String toString() - gathers and formats data for log file

Class Queue

This class constructs queues, defines logic for adding, removing customers.

Fields:

- Customer firstInQueue - first customer in line
- Customer lastInQueue - last customer in line

Methods:

- void add(Customer newCustomer) - adds customer to the queue; *TESTED by checking if customers are added*
- void clear() - clears the queue
- boolean isEmpty() - checks if queue is empty
- Customer peek() - gets first customer in the queue; *TESTED by checking method returns customers correctly*
- Customer remove() - removes first customer in the queue; *TESTED by checking if customers are removed*
- int size() - gets the queue's size; *TESTED by checking if size is correctly calculated*
- String toString() - prints IDs of customers in queue

Class Customer

This class constructs customers, generates their groceries, and calculates average times standing in line and shopping. Shopping logic is implemented in children classes.

Fields:

- int customerCnt - customer counter
- ArrayList<Integer> groceries = new ArrayList<>() - customer's groceries
- ArrayList<Integer> queueTimes = new ArrayList<>() - times spent in queue
- Customer soonerInQueue; - customer in queue before current
- Customer laterInQueue - customer in queue after current
- int id - customer's id
- int timeShopping - total time shopping
- int startBuyingTime - when customer started purchasing
- int timeEntered - when customer entered market
- int timeDeparted - when customer left market
- Market market - customer's market
- queueStartTime - what customer entered queue
- boolean isBuying - if customer is buying
- boolean isInQueue - if customer is in queue
- boolean departed - if customer has departed

Methods:

- ArrayList<Integer> getGroceries() - getter
- ArrayList<Integer> getQueueTimes() - getter
- int getId() - getter
- int getTimeEntered() - getter
- int getStartBuyingTime() - getter
- void setStartBuyingTime(int startBuyingTime) - setter
- Customer getSoonerInQueue() - getter
- void setSoonerInQueue(Customer soonerInQueue) - setter
- Customer getLaterInQueue() - getter
- void setLaterInQueue(Customer laterInQueue) - setter
- void generateGroceries() - generates customer's groceries; *TESTED by checking if groceries are generated*
- void doShopping() - stub for customer's logic, overridden in child classes
- int getTimeShopping() - calculates and returns time sent shopping
- int getAvgQueueTime() - calculates and returns average queue time
- getMinQueueTime() - calculates and returns minimum queue time
- getMaxQueueTime() - calculates and returns maximum queue time
- String toString() - gathers and formats data for log file

Class CustomerOne

This class extends Customer and implements unique shopping logic. This customer simply goes through their list in order and visits stalls corresponding to next item.

Fields:

- Market market - customer's market

Methods:

- void doShopping() - implements shopping logic mentioned above; *TESTED by checking if customer moves to and from the stall and removes item from a shopping list*

Class CustomerTwo

This class extends Customer and implements unique shopping logic. This customer searches for stall that has the good on the list and has the shortest queue.

Fields:

- Market market - customer's market

Methods:

- void doShopping() - implements shopping logic mentioned above;

Class CustomerThree

This class extends Customer and implements unique shopping logic. This customer simply chooses the first stall they find that has the desired good.

Fields:

- Market market - customer's market

Methods:

- void doShopping() - implements shopping logic mentioned above;

Challenges

- Putting everything together was not easy, but it worked eventually.
- Market did not function correctly due to some mistakes in time calculations in he stalls, but those issues were resolved.
- Log file completion took some time, but once running it was a critical debugging tool.
- Making choices about the design definitely spurred progress. As the program unveils, questions answer themselves.
- Summarizing data in the program would be very handy.