

Project 2 – The Hairdresser’s Shop

Background

The aim of this project is a statistical simulation of a hairdresser’s shop, finding by stochastic methods how many customers it can serve per day, or how much revenue is generated.

The shop has a fixed number of seats for waiting customers. Only one hairdresser works there, who serves one customer at a time, in the order in which they have arrived. Customers who enter the shop take a seat (if available) and wait in the queue until it’s their turn. If there is no free seat, they leave again. Customers can also leave if they feel they’ve waited for too long, which will free up their space in the queue.

We will simulate the arrival of new customers by a (pseudo-)random process, and the “waiting tolerance” of customers is varied at random as well. We will consider various models of the behaviour of customers, how long it takes to serve them and how much they pay.

The shop in this assignment is, of course, entirely fictional, and any resemblance to actual hairdresser’s shops or their customers is entirely coincidental.

Tasks

As a general convention, all real numbers in the following will be represented by `double` floating point numbers in the code, and all integers by the `int` type. Times are given in minutes (integer). Where the specification refers to probabilities, these are floating point numbers between 0 and 1 (inclusive). Random choices will always be made according to a uniform distribution.

In object types, all (non-static) methods specified on this assignment sheet should be declared as `public`, fields and internally used helper methods (if any) should be declared as `private`. Functions and procedures (static methods) should be left with their default visibility.

Unless specified otherwise, you do not need to check input parameters for consistency or for exceptional input values.

1) A simple barber shop

In this part, we simulate a simple barber shop. Customers come in for a haircut which always takes 15 minutes. That is, the barber serves one customer every 15 minutes, as long as any are waiting; otherwise he takes a 15-minutes break. Customers have a certain “waiting tolerance” time, after which they become annoyed and leave, varying between 30 and 90 minutes. We aim to find the (average) number of customers served every day in the shop.

We model the queue of the shop as follows: What we need to know about each customer is the time that they’re still willing to wait. We capture this number in an integer (in minutes). Hence we store the queue of the shop as an array of integers, say `q`, each entry representing the state of one of the n seats. `q[0]` correspond to the front of the queue. `q[n-1]` to its end. A waiting customer is represented by a positive integer in the appropriate entry, namely the number of minutes that the customer is still prepared to wait; a 0 entry means a free seat. We will always assume that there are no free seats “in the middle”, hence zeros occur only at the end of the array. Functions that modify the array should make sure that again there are “no free seats in the middle”. For example, the following depicts a queue with 5 seats and 3 customers waiting, which are willing to wait for 43, 23 and 57 more minutes, respectively; the customer with 43 minutes remaining waiting time will be served next.

q[0]	q[1]	q[2]	q[3]	q[4]
43	23	57	0	0

The queue arrays should be handled as *mutable*, that is, functions working on the queue should modify the entries of the given array, rather than working on a copy.

All functions and procedures in this part should be implemented in a class `SimpleShop`.

- (a) Write a function `peopleWaiting` which takes an array of integers as its input – interpreted as a queue as above – and returns the number of people waiting in the queue.
- (b) Write a function `addToQueue` that adds an entry to the end of the queue if a free space is available. It should take the queue (as an array) and the “waiting tolerance” of the new customer (assumed to be > 0) as its input. It should return a boolean value, namely `true` if the entry was successfully added, and `false` if there was no space in the queue.
- (c) Write a function `removeFromQueue` that will remove the first entry from the queue, i.e., the next customer to be served. It should then advance the queue, i.e., move all waiting customers forward, leaving an additional free space at the end. It should return a boolean value, namely `true` if there was a customer in the queue and `false` if the queue was empty.
- (d) Write a procedure `expire` that takes a queue as its input, and an integer representing a time (in minutes) that has passed. It should subtract this time from the “waiting tolerance” of all customers in the queue. If the tolerance limit has been reached for any customer (e.g., if the time given is 15 minutes and they have no more than 15 minutes of “waiting tolerance” left), then the customer should be removed from the queue (they “leave the shop”) and any remaining customers should advance forward.
- (e) Write a function `serveOneCustomer`, taking a queue array and a probability p as its input. The function should do the following: It removes the first customer from the queue; then expires 15 minutes from the queue as in (d) (while the customer is being served, or the barber goes for a break). After that, it simulates arriving customers by doing the following 15 times:
 - With probability p , add a new customer to the end of the queue.
 - The new customer (if any) has a waiting tolerance of at least 30 minutes but less than 90 minutes, randomly chosen.

The function should return a boolean value, namely, whether a customer was served in this turn or not.

- (f) Write a function `runDay` which simulates the running of an entire day in the shop. It takes a queue, the number n of 15-minutes turns in the day, and a probability p as its input. The function then runs n turns of the type described in (e), starting the first one with the queue as given. It counts the number of customers actually served in the day, and returns it as an integer.
- (g) Write a function `averageNumberServed` which computes the number of customers served per day, on average. Its parameters are: the number of turns per day; the number of days d to average over; the probability p ; and the number of seats. It does d times the following: It sets up a new, empty queue of the specified length, and simulates one day on it as in (f), with parameters as given, recording the number of customers served. It then computes the arithmetic average over the number of customers served, and returns this as a floating point number.

2) More sophisticated customers

We now want to extend the situation a bit. There is still one hairdresser in the shop, but the shop now has two types of customers, male and female.

Male customers come for a simple haircut. This takes 15 minutes as above, and they pay £12 for it. They are willing to wait for between 30 and 90 minutes. However, when they see that a *single* customer before them is served for longer than 35 minutes, they become frustrated and leave immediately.

Female customers ask for more sophisticated services, which will take the hairdresser between 20 and 50 minutes. They pay a flat fee of £25, and wait up to 120 minutes.

There is still only one queue, in which both male and female customers wait. They arrive in certain proportions, as specified in detail below. The hairdresser serves them in the order they arrive. If there are no customers in the queue when the hairdresser finishes with the previous one, then he goes for a coffee, which will take 10 minutes.

We will model this with object-oriented techniques, using a class `Customer` and its subclasses. The queue is now stored as an array of `Customer` objects (rather than integers), and empty seats are represented by `null`.

(a) Create a class `Customer` which should have the following methods:

- `wait`: simulates that the customer waits in the queue for a number of minutes, given as an integer parameter, while another customer is being served. Returns nothing.
- `hasLeft`: Returns whether the customer has left the shop (true/false). No parameters.
- `getRequiredTime`: Returns the time in minutes required to serve the customer. No parameters.
- `getPayment`: Returns the amount (integer, in GBP) which the customer needs to pay. No parameters.

In this class, the customer should take 10 minutes to be served, pay £1, and never leave the shop. The method `wait` does nothing here (but will do below). The class should be left with its default constructor.

(b) Create two subclasses `MaleCustomer` and `FemaleCustomer` of `Customer`. In these, override the methods `wait`, `hasLeft`, `getRequiredTime` and `getPayment` to reflect what was said about male and female customers above. Create constructors that, for `MaleCustomer`, take the “waiting time tolerance” (integer) as a parameter, and for `FemaleCustomer`, the time it will take to serve the customer.

(c) Create a class `AdvancedShop`, which should have a field of type `Customer[]` that stores the state of the shop’s queue. The constructor of the class should take one integer parameter ($n > 0$), and should initialize the queue as having n seats with all of them empty. Implement a method `getQueue` which returns the current state of the queue.

Warning: It is very important that you implement and thoroughly test this part, as I will use it to verify your results in the following questions.

(d) Inside `AdvancedShop`, create methods `addToQueue`, `removeFromQueue`, and `expire` which work as in Part 1(b)-(d) of the project, but with the following differences: The queue array is no longer given as a parameter, rather the methods use the queue stored in the corresponding field of the `AdvancedShop` object. Also, details of new customers for `addToQueue` are no longer specified by an integer, but rather as a `Customer` object. `removeFromQueue` should return the customer object removed from the queue, or `null` if none was in the queue. Invoke methods from the `Customer` class as appropriate.

- (e) Inside **AdvancedShop**, write a method **arrivingCustomer** with no parameters that returns a randomly generated **Customer** object (a “newly arriving customer”) according to the following rules: 70% of customers are female, 30% are male. Male customers have a random waiting tolerance ≥ 30 and < 90 minutes. Female customers take (at random) ≥ 20 and < 50 minutes to be served.
- (f) Write a method **AdvancedShop.serveOneCustomer** which takes a probability p as its input. It removes the next customer from the queue; then it expires the time required to serve this customer (or 10 minutes if no customer was waiting). Then, once for each minute in the time frame just mentioned, it adds with probability p a new customer to the queue. New customers are chosen at random as in (e). It returns the payment received from the customer served, if any.
- (g) Write a method **AdvancedShop.runDay** that simulates an entire day; it takes the number of minutes in the day and the probability p as inputs, and then runs turns as in (f) until the given length of day has been exceeded. (If the last customer takes longer to serve than the remaining time in the day, the hairdresser will continue to serve them and then close the shop, all remaining persons in the queue will leave.) It returns the total payment collected on that day.
- (h) Write a *function* **AdvancedShop.averageRevenue** which averages the result of (g) over several days: It takes the number of minutes per day (> 0), the number of days $d > 0$, the probability p , and the length of the queue as its input. Then, for each day, it sets up a new **AdvancedShop** with the desired number of seats, simulates one day on it as in (g), and records the payment received. It averages this payment over d days and returns that average in floating point format.

In (h), if any of the input parameters is outside the allowed range, the function should throw a `java.lang.IllegalArgumentException`.

3) Documentation and analysis

- (a) *Source code documentation:*

Within the source code, add Javadoc comments to every (non-private) function, procedure, method and class. In these, describe the purpose of the function, class, etc., and document all parameters and return values. Also, document all exceptions that are thrown by your own code. Generate the Javadoc output and hand it in together with the project.

- (b) *Test plan:*

For question 1(b)–1(d), write a test plan: What relevant examples can you use to test your code? What are the input values you would use, and what are the expected outputs? Give at least 3 relevant test cases for each of the methods, covering different scenarios including “edge cases”.

- (c) *Analysis:* Run your program for different configurations, answering *two* of the following questions in examples:
- In Part 1, how much does the number of customers change when the probability p is varied (other parameters being kept fixed)?
 - In Part 2, how much does the revenue of the shop change when the probability p is varied (other parameters being kept fixed)?

- In Part 1, for fixed p , how much does the number of seats in the queue influence the number of customers served?
- In Part 2, for fixed p , how much does the number of seats influence the revenue of the shop?

Alternatively, you can choose to change other parameters of the setup (prices, maximum waiting times, ...) or evaluate the averages of other quantities (the actual waiting time per customer, the length of the queue, the coffee intake of the hairdresser, ...).^{*} In any case, present *two* pieces of analysis.

^{*}This may require a modification to your code. In that case, please make sure that the version you submit adheres exactly to the specification above, otherwise you may fail some of the tests.

How to prepare your submission

Start by downloading the template files from Moodle. However, none of the classes required are defined there, you need to create them on your own.

Read the assignment sheet carefully. Be sure to implement all classes, functions and methods with exactly the names and parameters that are specified.

While preparing your code, please follow these style and formatting guidelines. (This forms part of the assessment.)

- Place all code block delimiters – { and } – on separate lines.
- Within code blocks, indent the code by 4 spaces with respect to the surrounding code.
- Choose all names for variables, parameters, functions, methods and fields to start with a lowercase character.

Make sure that your code compiles correctly. If the source code files that you submit do not compile (for whatever reason), you will receive zero marks for the affected parts of the project.

The code template includes a unit test. This test does *not* check the output of your code – it only verifies whether you have declared your functions and methods with the correct names and with the correct parameter/return types. You are advised to use the unit test to check your function declarations for any typos. (For running the unit test, you will first need to create all classes used in this project. You can leave them empty if needed.)

Test every piece of your code with meaningful test data, including any special or exceptional values where relevant. Verify that the output of your code is plausible.

Once you are satisfied with the code, create a JAR file in BlueJ as follows:

- Select “Project → Create Jar file” in the menu.
- Leave “Main class” set to “none”.
- ***Be sure to tick “Include source” and “Include BlueJ project files”.***
- Click “Continue” and save the file in a convenient location.

This JAR file is the one that you need to submit. Note that this will automatically contain the Javadoc output if this has been generated beforehand.

Please prepare the documentation 3(b) and 3(c) in a single PDF file. You can use any word processing program of your choice to produce it.

Please do not mention your name, user id or exam number in the code or the documentation, as your submission will be graded anonymously.

Hand-in, late submissions

Please submit your work by uploading it on Moodle (see the “week 10” section of the course) before

Thursday, 19 January 2017 (week 2 Spring term), 20:00h.

Your submission should contain exactly two files: one JAR file (containing the code) and one PDF file (containing the documentation/analysis).

Late submissions will incur a penalty of 10% of the *available* marks for each day that the assignment is submitted late; submissions that are more than 5 days late will be awarded a mark of 0. Please be sure to press the “Submit assignment” button on Moodle before the deadline.

If your submission is affected by Exceptional Circumstances (such as, being ill), you may be granted a deadline extension. For that, you would need to hand in an Exceptional Circumstances claim, accompanied by evidence. If you think this may apply to you, please contact the Assessment Administrator at maths-exams@york.ac.uk as soon as possible – and *before* the deadline if feasible.

Marking

Your submission will be marked as follows. Partial solutions are acceptable, and will be awarded partial marks.

Code correctness (27 marks) Your code will be tested by an automated process (a collection of unit tests) to check whether it works correctly for various values of the input parameters. There are 54 functional tests, and for each of them that you pass, 0.5 marks will be awarded. Marks are awarded entirely on the criterion whether the output values of your code match the conditions specified on this assignment sheet.

Coding style (10 marks) Your source code will be read by the examiner. Marks will be awarded to reflect whether your code is readable, is appropriately structured, and follows the coding style conventions mentioned above. Marks may be subtracted for code that is functional but is implemented in a non-transparent or overly complicated way.

Source code documentation (5 marks) These marks are awarded for appropriate Javadoc documentation, see 3(a).

Test plan (6 marks) These marks will be awarded for the test plan, as outlined in 3(b).

Analysis (6 marks) These marks will be awarded for the analysis as outlined in 3(c).

In total, 54 marks are available. Your raw mark out of 54 will be moderated and scaled to the University scale 0–100 in consultation with the department’s Assessment Committee. This scaled mark will contribute 70% towards your final mark for the module.

Academic integrity

To this project, the established Academic Integrity rules apply, as set out in the University’s regulations. In particular, you must not copy program code or the text of the documentation from fellow students, or from other public or non-public sources. You must also not make your solution (program code or documentation) available to fellow students before the deadline.

You are allowed to re-use example code from the lectures or practicals (though, where you use it, you should include a comment to that effect in your source).