

## Supermarket cashier simulation

In this assignment you demonstrate effective and efficient use of Sets, Maps and Queues from the Java Collection Framework.



A national Supermarket franchise 'Jambi' has asked you to perform some data analysis and simulation studies on their customers purchase statistics and the throughput performance of their cashier checkout stations. They wish to investigate the impact on waiting times and queue lengths of different queuing rules at the cashiers.

'Jambi' provides you with a 'jambi250\_8.xml' file that holds the purchase records of 250 customers visiting one of their supermarkets between 12:00pm and 15:00pm on a Saturday afternoon. The file provides for each customer the zip code of her residence, a list of products that this customer has bought from in the supermarket that afternoon and the time that he arrived at the check-out area. For each product, also a description and a unit-price are specified in the file.

Along with this assignment, a Java starter project is provided which includes some utility code, some skeleton application classes and unit tests to load the data from the xml files and drive the simulation. You may solve this assignment by extending/amending/modifying this starter project. In Figure 3 below you find the class diagram that explains some of the design of this starter project. The starter project also contains the jambi250\_8.xml file along with some other test examples in the resources folder.

### Reporting the results.

The Supermarket class in the starter project includes parts of two methods `printCustomerStatistics()` and `printSimulationResults()` that can be used to show a summary of the results after loading the data and after running a simulation scenario. These methods should report the following information at the console, similar to Figure 1 below.

`printCustomerStatistics()` shows:

1. Name of the input file and open and closing time of the simulation period.
2. Number of customers that have visited in that period, total number of items that they have bought, and number of different products available from the supermarket's product list.
3. Total revenue from all customers split by zip code and sorted ascending on zip code.  
(The total revenue you find by multiplying the number of items times the price per item and summing over all items bought by all customers within the zip code.)  
Also show per zip code the product that has been bought in highest total quantity by customers within the zip code.  
(If multiple products have the same maximum count for a given zip code, you report just one arbitrary most bought product for that zip code.)
4. The overall total revenue from all purchases in the file.

You can implement this functionality without any cashier simulation.

`printSimulationResults()` shows per Cashier in the scenario:

- a) the name of the cashier
- b) the number of customers that have been checked out by the cashier
- c) average waiting time per customer at the cashier, excluding own check-out time
- d) maximum waiting time of any customer at the given cashier (excluding check-out)

- e) maximum queue length at any time at the cashier. (The queue length includes the customer being served.)
- f) average check-out time of a customer at the cashier for his own items bought.
- g) total idle time of the cashier during which no customer was being handled.

Also show the overall totals across all cashiers or averages across all customers. (This is only relevant for multi-cashier simulations.)

### Three cashier configuration scenarios.

You are requested to study three cashier simulation scenarios:

1. The base scenario, which has one single (FIFO) cashier at the check-out area. This cashier serves all its customers on the basis of first-come-first-served.
2. The priority scenario, which has one single (PRIO) cashier at the check-out area. This cashier allows customers with no more than 5 items to jump the queue passing all other customers with more than five items (but not passing any other customers with less than 6 items).
3. The mixed scenario, which has two cashiers at the check-out area which each have their own queue. One cashier follows the FIFO rules, the other the PRIO rules. When a customer arrives at the check-out area he selects the cashier of his preference (based on the shortest expected pass-through time including waiting time and check-out time).

The layout of these three scenarios is illustrated by Figure 2 below. The `SupermarketMain` class in the starter project already provides code to setup and run these scenarios for you, but you first need to implement the `Cashier` concrete classes and configure them in the supermarket to make that work.

### Simulation algorithm.

The main engine of the simulation is the method `simulateCashiers()` in the `Supermarket` class. Basically, it simulates sequential arrival of all customers at the check-out area, one after the other at their given times. After each arriving customer, first all cashiers need to proceed their work up to the point of time when the new customer has arrived. Thereafter the new customer can decide which queue to join. This scheme is depicted in more detail in Figure 4. (If there is only one cashier, then the customer has no choice and the customer can only join the queue of that single cashier)

One important factor in the simulation is the actual time that it takes to scan all product items of one customer at the cashier terminal and complete the payment, royalty options and social chit/chat etc. In reality there will be some variation of work-pace for different customers, but for this study you may assume a uniform model:

check-out-time = 20 seconds (per customer) + 2 seconds per bought item.

(customers without any bought items have zero check-out-time)

Another important factor in the simulation of multiple cashiers is the decision of an arriving customer about which queue to join. In reality, customers might guess which queue might be the quickest based on the total amount of goods they see in front of them or may also include other personal factors of preference in their choice. In reality they also may decide to leave their queue at some point and join another one if check-out proceeds slower than their initial guess or if they remember about another item that was missing from their shopping list.

For this study you may ignore all these complicating factors. Just let the new arriving customer pick the queue that has the shortest expected 'pass-through' time for that customer:

my expected pass-through-time =

my expected waiting-time for completion of earlier, unfinished customers +

my expected check-out-time of my bought items.

For sake of simplicity we measure all times in seconds.

Key methods to complete implementation before a one-cashier simulation can be run are:

- `Cashier.add(customer)`
- `Cashier.expectedCheckOutTime(nrOfItems)` *(implement in `FIFOCashier`)*
- `Cashier.doTheWorkUntil(targetTime)` *(implement in `FIFOCashier`)*
- `Supermarket.simulateCashiers()`

For the implementation of the multi-cashier simulation you also require complete implementation of:

- `Cashier.expectedWaitingTime(customer)`
- `Customer.selectCashier(cashiers)`.

You may need to make some further assumptions when implementing these latter two methods. Those assumptions you should clarify in your report.

### How to get going with this assignment.

The starter project provides // TODO comments at every location where code is missing. You can deliver the requirements of the method `printCustomerStatistics()` without the need of completing any of the `Cashier` (sub)-classes or simulation methods mentioned above.

The `src/test` folder includes some tests that should help you to produce correct code. The names of the test methods start with a number: `tNNN`. The lower the number, the more fundamental is the test for your solution. Your approach should be to always first fix the failing unit tests with the lowest number, because unit tests with higher numbers may depend on correct implementation of functionality that is tested by a lower number.

(The tests themselves are not interdependent, but the tested functionality may be.)

Some of these tests are based on smaller data files `jambi1.xml`, `jambi2.xml` and `jambi5.xml`. Figure 1 below provides reference output for running test case `jambi50_7.xml`.

You are encouraged to add more tests as you find useful.

The starter project also includes a class `SupermarketGenerate` which you can use to generate more and even bigger test cases. You can also edit the xml files manually to construct specific test cases. The provided schema file `Supermarket.xsd` may be helpful to validate the format of your test files.

You are encouraged to exchange test case files with other students and compare results. Some results of the simulation may not deterministically reproduce across all correct solutions of your colleagues, because each of you may have made different assumptions.

### Your report

Besides the directives of the provided reporting guidelines, we expect the following commentaries in your report:

1. The console output of the `printCustomerStatistics()` and `printSimulationResults()` method for each of the three scenario's on the case of '`jambi250_8.xml`', similar to Figure 1 (which provides possible output of the smaller case '`jambi50_7.xml`').
2. Your commentary about reproducibility of the overall results of the mixed scenario. Some of these results are predetermined by the input data. I.e. the total number of customers handled by the cashiers should equal the total number of customers in the datafile.  
But what about the other numbers of average-wait-time, max-wait-time, max-queue-length, average-check-out-time and total idle time? Which are conserved? Which are somewhat depending on your implementation choices of the cashier simulation? Include your relevant code snippets and explain that dependency.

- Provide at least three additional code snippets from other methods in your solution and explain your approach.

## Basis of grading

Some parts of this assignment are more difficult than others...

You earn 'sufficient' if you succeed in calculating the revenues per zip code, and the simulation statistics of both scenarios with only one cashier.

You earn 'good' if beyond that you succeed in determining the most bought products per zipcode.

You earn 'excellent' if you also succeed to implement the mixed scenario.

## Figures

Customer Statistics of 'jambi50\_7.xml' between 12:00 and 12:30  
 50 customers have shopped 369 items out of 25 different products  
 Revenues and most bought product per zip-code:  
 1013AD:75,70(Verse scharreleieren 4 stuks), 1013BK:26,47(Coca Cola Zero 1.5L), 1013JG:173,39(Bloemkool)  
 1013KN:332,66(Cashew noten 300g), 1014BE:43,18(Campina halfvolle melk 1L), 1014CL:86,38(Coca Cola Zero 1.5L)  
 1014LO:29,27(Douwe Egberts snelfilter 500g), 1015CF:76,37(Croissant), 1015DM:116,95(Robijn stralend wit)  
 1015LI:9,96(Gourmet tonijn 100g), 1015MP:82,71(Kip kilo knaller), 1016DG:35,23(Kaiser broodje)  
 1016EN:119,73(Campina magere yoghurt 1.5L), 1016MJ:54,41(Multivruchtsap 2L), 1016NQ:4,75(Gourmet tonijn 100g)  
 1017FO:45,85(Kip kilo knaller), 1017NK:138,31(Eendeborst 500g), 1017OR:100,39(Croissant)  
 Total Revenue=1551,71

Scenario Simulation results:  
 Cashiers: n-customers: avg-wait-time: max-wait-time: max-queue-length: avg-check-out-time: idle-time:  
 FIFO 50 84,7 169 6 34,8 62  
 overall: 50 84,7 169 6 34,8 62

Scenario Simulation results:  
 Cashiers: n-customers: avg-wait-time: max-wait-time: max-queue-length: avg-check-out-time: idle-time:  
 PRIO 50 69,5 263 6 34,8 62  
 overall: 50 69,5 263 6 34,8 62

Scenario Simulation results:  
 Cashiers: n-customers: avg-wait-time: max-wait-time: max-queue-length: avg-check-out-time: idle-time:  
 FIFO 25 2,7 26 2 33,8 954  
 PRIO 25 2,1 18 2 35,7 908  
 overall: 50 2,4 26 2 34,8 1862

Figure 1: Sample output of customer statistics and three cashier simulation scenarios's





Base Scenario	Priority Scenario	Mixed Scenario
FIFO 	PRIO 	FIFO  PRIO 

Figure 2: Three Cashier simulation scenario's

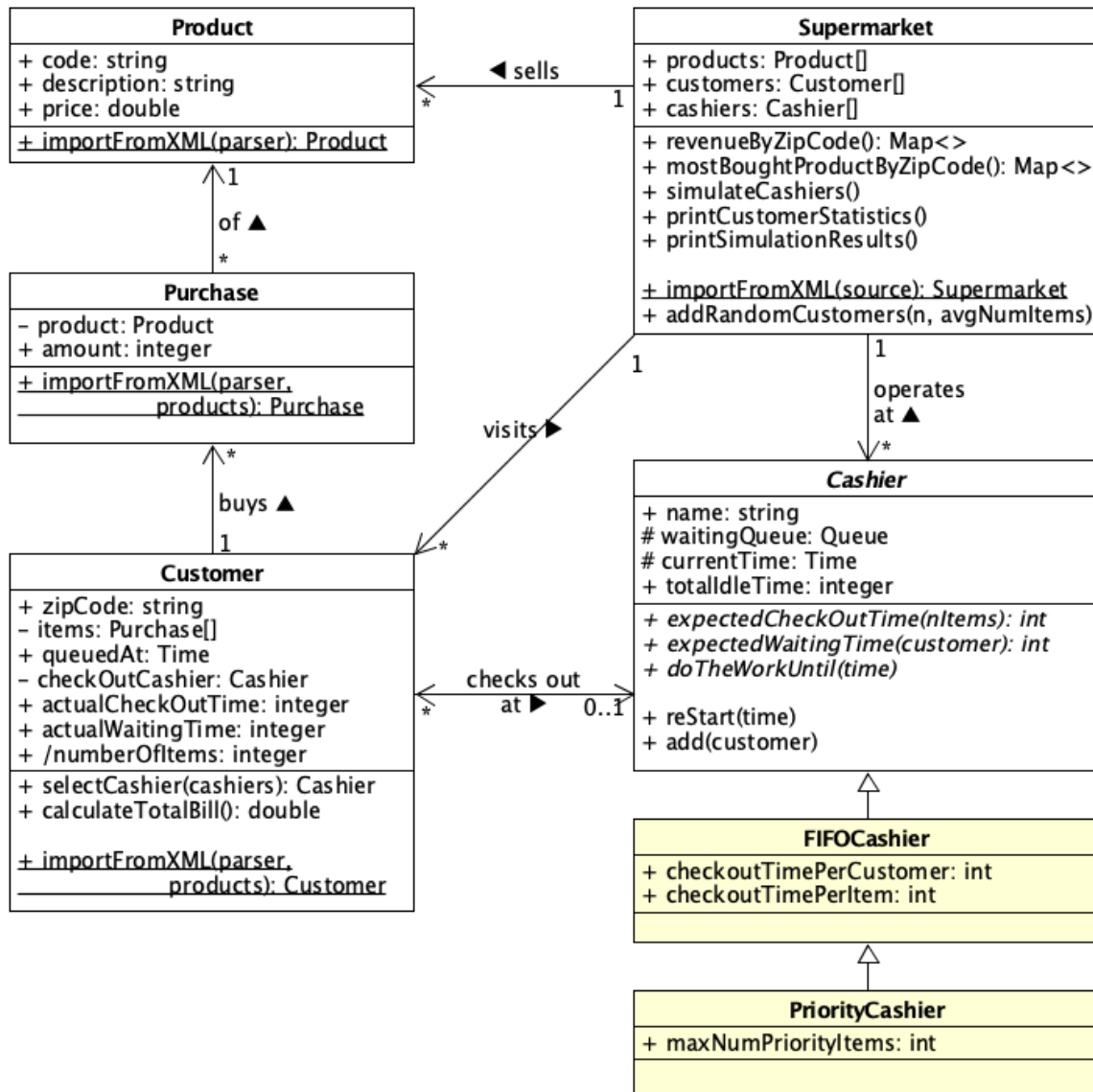


Figure 3: Supermarket class diagram

Some notes:

1. The white classes are provided in the starter project, but essential parts of the implementation are missing. That is indicated with // TODO comments.
2. The yellow classes are not provided in the starter project. Those you should create yourselves. Start with FIFOCashier, then PriorityCashier.  
As you add these Cashier subclasses, also their unit tests will be activated.
3. You should not need to worry about the XML import and export methods. They will do their job once you have completed the class definitions.

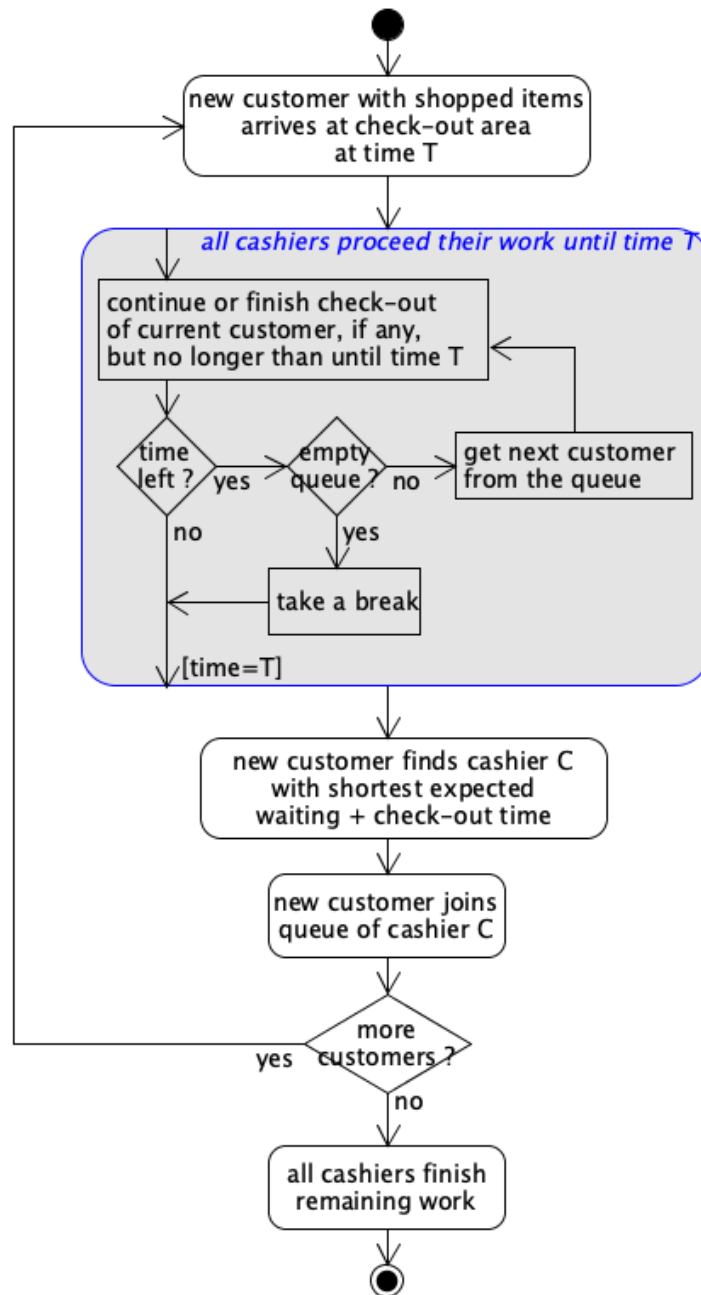


Figure 4: Cashier simulation flow diagram

Note:

The gray box in this diagram suggests an algorithm for partly progressing the work of the Cashiers until some target time T, such that appropriate statistics can be collected on the fly about waiting times, queue lengths and idle times. You are anticipated to implement this algorithm in the method `Cashier.doTheWorkUntil(targetTime)`.

The proposed algorithm follows an approach of tracking the work of the Cashiers customer by customer. You could also implement an alternate algorithm tracking the work of the Cashiers second by second. You are free to implement your own choice of solution and change provided code, whatever is required to compute all requested results.