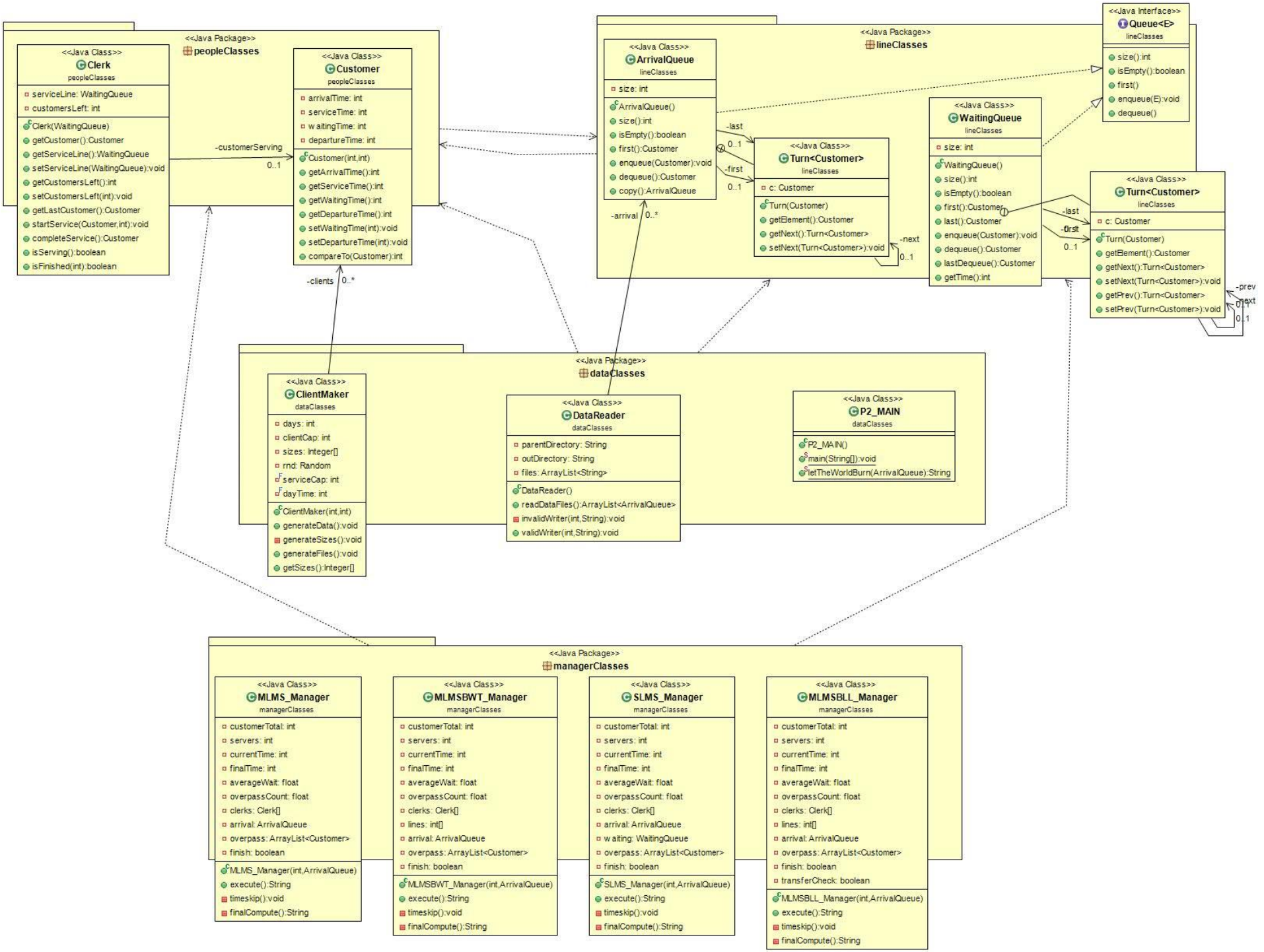
This program is built to test out different approaches for the service area of a store, supermarket, etc. There are 4 different approaches that function in different ways. The first approach is SLMS, which stands for Single Line Multiple Servers. As with all approaches, the "Multiple Servers" actually stands for "Variable Servers" since one server is an option. This first approach work by having all possible servers serve the same waiting line. This is similar to the service area in a pharmacy (e.g. Walgreen's). The second approach is MLMS or Multiple Lines Multiple Servers. From this one onwards each possible server is in charge of their own line. In this one people can't change lines, so it resembles a toll station (peaje). The third is MLMSBLL, which is Multiple Lines Multiple Servers Balanced Line Length. Customers are allowed to change lines here, to a line where they will benefit from the change (i.e. there are less people in front of them), but only when they're in the last position of a line; that makes it similar to a supermarket. The fourth and last one is MLMSBWT or Multiple Lines Multiple Servers Balanced Waiting Times. Similar to MLMS, Customers can't change lines; but it won't be necessary since this approach analyzes the time that each person in a line will take and automatically assign each arriving Customer to the best possible line for them. In theory, each approach should be better and more efficient than the one before.

To use the program, open GitBash in the project directory **p1_40354020_172**
Run *java -classpath ./bin dataClasses/P2_Main*
Enter the amount of days you want to test (D)
Enter the maximum amount of customers that can arrive in a day (CC)
The console should now show you each data file being created and processed
After being told the system is finished verify in the subdirectory **inputFiles** that the data files have been generated and in **outputFiles** for the results
To run the program again, delete all the files in **inputFiles** first, just to make sure you get the correct files

The system will generate up to D data files with up to CC entries on each. There may be some missing or invalid files, but there's no need to worry. The system will automatically find these and label them as such in the output files. Days will also have an arrival cap, the last time that a new Customer can arrive, and it is set to 4 times CC. This can be changed to cause smoother days or mass chaos.

A Testing class is available if you wish to try out different parts of the system. Sysout commands are spread across each approach to verify each event as it passes. You can uncomment these to see them. The last approach, MLMSBWT contains a commented implementation of the transfer event. This is simply to prove that a correctly implemented MLMSBWT will never need a transfer. You can uncomment these lines to see.


<<Java Package>>
peopleClasses
<<Java Class>>
Clerk
peopleClasses
serviceLine: WaitingQueue
customersLeft: int
Clerk(WaitingQueue)
getCustomer():Customer
getServiceLine():WaitingQueue
setServiceLine(WaitingQueue):void
getCustomersLeft():int
setCustomersLeft(int):void
getLastCustomer():Customer
startService(Customer,int):void
completeService():Customer
isServing():boolean
isFinished(int):boolean
-customerServing
0..1
<<Java Class>>
Customer
peopleClasses
arrivalTime: int
serviceTime: int
waitingTime: int
departureTime: int
Customer(int,int)
getArrivalTime():int
getServiceTime():int
getWaitingTime():int
getDepartureTime():int
setWaitingTime(int):void
setDepartureTime(int):void
compareTo(Customer):int
-clients
0..*
<<Java Package>>
lineClasses
<<Java Class>>
ArrivalQueue
lineClasses
size: int
ArrivalQueue()
size():int
isEmpty():boolean
first():Customer
enqueue(Customer):void
dequeue():Customer
copy():ArrivalQueue
-last
0..1
-first
0..1
-arrival
0..*
<<Java Class>>
Turn<Customer>
lineClasses
c: Customer
Turn(Customer)
getElement():Customer
getNext():Turn<Customer>
setNext(Turn<Customer>):void
-next
0..1
<<Java Class>>
WaitingQueue
lineClasses
size: int
WaitingQueue()
size():int
isEmpty():boolean
first():Customer
last():Customer
enqueue(Customer):void
dequeue():Customer
lastDequeue():Customer
getTime():int
<<Java Interface>>
Queue<E>
lineClasses
size():int
isEmpty():boolean
first()
enqueue(E):void
dequeue()
<<Java Class>>
Turn<Customer>
lineClasses
c: Customer
Turn(Customer)
getElement():Customer
getNext():Turn<Customer>
setNext(Turn<Customer>):void
getPrev():Turn<Customer>
setPrev(Turn<Customer>):void
-prev
0..1
-next
0..1
<<Java Package>>
dataClasses
<<Java Class>>
ClientMaker
dataClasses
days: int
clientCap: int
sizes: Integer[]
rnd: Random
serviceCap: int
dayTime: int
ClientMaker(int,int)
generateData():void
generateSizes():void
generateFiles():void
getSizes():Integer[]
<<Java Class>>
DataReader
dataClasses
parentDirectory: String
outDirectory: String
files: ArrayList<String>
DataReader()
readDataFiles():ArrayList<ArrivalQueue>
invalidWriter(int,String):void
validWriter(int,String):void
<<Java Class>>
P2_MAIN
dataClasses
P2_MAIN()
main(String[]):void
letTheWorldBurn(ArrivalQueue):String
<<Java Package>>
managerClasses
<<Java Class>>
MLMS_Manager
managerClasses
customerTotal: int
servers: int
currentTime: int
finalTime: int
averageWait: float
overpassCount: float
clerks: Clerk[]
arrival: ArrivalQueue
overpass: ArrayList<Customer>
finish: boolean
MLMS_Manager(int,ArrivalQueue)
execute():String
timeskip():void
finalCompute():String
<<Java Class>>
MLMSBWT_Manager
managerClasses
customerTotal: int
servers: int
currentTime: int
finalTime: int
averageWait: float
overpassCount: float
clerks: Clerk[]
lines: int[]
arrival: ArrivalQueue
overpass: ArrayList<Customer>
finish: boolean
MLMSBWT_Manager(int,ArrivalQueue)
execute():String
timeskip():void
finalCompute():String
<<Java Class>>
SLMS_Manager
managerClasses
customerTotal: int
servers: int
currentTime: int
finalTime: int
averageWait: float
overpassCount: float
clerks: Clerk[]
arrival: ArrivalQueue
waiting: WaitingQueue
overpass: ArrayList<Customer>
finish: boolean
SLMS_Manager(int,ArrivalQueue)
execute():String
timeskip():void
finalCompute():String
<<Java Class>>
MLMSBLL_Manager
managerClasses
customerTotal: int
servers: int
currentTime: int
finalTime: int
averageWait: float
overpassCount: float
clerks: Clerk[]
lines: int[]
arrival: ArrivalQueue
overpass: ArrayList<Customer>
finish: boolean
transferCheck: boolean
MLMSBLL_Manager(int,ArrivalQueue)
execute():String
timeskip():void
finalCompute():String