**Customer.java**

**Pre-Condition**: list of orders != null

Name != null

Eating Time > 0

Priority > 0

**Post-Condition**: Customer must leave after having correct order

The event should end if there are no customers waiting or in the café

**Invariance**: tablesOccupied must be less than or equal to numTables

All the customers waiting outside must get into the coffee shop

Any Customer cannot place two orders

Priority must be given to each customer

Customer must not get order before cook completes the whole order

Customer must not leave café before finishing the order

Cannot place order outside defined food types

Number of customers must not exceed specified numbers

**Exception**: InterruptedException

**Pseudo** **Code**:

run()

Event : CustomerStarting

while(true)

if(Order is not placed yet)

if(tablesOccupied >= numTables)

this.wait();

else

AcquireLock(availability)

tableOccupied++;

ReleaseLock(availability)

Event : CustomerEnteredCoffeeShop

Add a new entry to ordersList

if(This customer has already placed any order)

interrupt the thread as Customer can’t place more than 1 order

else

list = null;

add this current order to the list and update the customerOrder List

if(there are any pending orders)

list = list of orders numbers

else

list = null;

add this current order number and priority to the list and update the pending order List (this list is treeMap, thus sorted by keys, i.e. by priorities)

Event: CustomerPlacedOrder

placedOrder = true

if(placedOrder)

wait for order to get ready

once order is ready i.e. notified by cook

check if order given is same as order received or not. If(yes)

Event: CustomerReceivedOrder

Thread.sleep(eatingTime);

Event: CustomerLeavingCoffeeShop

notifyAll();

**Cook.java**

**Pre-Condition**: Name != null

**Post-Condition**: There should not be any pending orders while terminating the program

**Invariance**: number of cooks must not exceed the specified number

Cook should not start the order before getting it

Cook should not complete the order before getting all foodItems ready by the machine

If interrupted at any point, should terminate the whole order

**Exception**: InterruptedException

**Pseudo Code**:

run()

Event: CookStarting

try{

while(true)

if(cook is available)

set availability = false

get list of orderNumbers by priority

if(list!= null)

order = first order in the list

Event: CookReceivedOrder

Event: CookStartedFood

for(Food f: list)

type = foodType

put the food in the machine

proceed to the next item in the machine

if(f is finished)

Event: CookFinishedFood

wait till all the items get cooked

if(interrupted by Machine class)

terminate //Event: CookEnding

else

update orderReceived of Customer class

notify Customer

Event: CookCompletedOrder

}

Catch(InterruptedException e){

Event: CookEnding

}

**Machine.java**

**Pre-Condition**: Food must not be empty

Cook must not be empty

Machine name must not be empty

Capacity should be greater than 0

**Post-Condition**: There should not be any food in any machine while terminating the program

Must return true as soon as Food is ready

**Invariance**: Should not place food in the machine if capacity is maximum

If(max capacity is reached) must interrupt the cook thread

Must wait while food is cooking

**Pseudo Code**:

Event: MachineStarting

boolean makeFood(Food food, Cook cook, int capacity)

type = food.getFoodType

for(Machine m: MachinesList)

if(m.machineFoodType.equals(type))

AcquireLock(count)

int num = get number of food inside machine

if(num >= capacity)

interrupt Cook thread.

return false

ReleaseLock(count)

Event: MachineStartingFood

Thread.sleep(food.cookTimeMS)

Event: MachineDoneFood

notify Cook class

return true

**Simulation.java**

**Invariance**: At any point Customers should not exceed specified number

At any point Cooks should not exceed specified number

Order must be randomly generated if randomOrder is true else standard order is generated

Customers inside café must be less than or equal to number of tables

**Pre-Condition**: None

**Post-Condition**: validate method returns true or false

**Pseudo Code**:

Has 3 Maps, viz.

customerAndOrder that keeps track of all the orders placed by every customer.

pendingOrders which is a sorted Map that keeps track of all the order numbers under each priority

ordersAndOderNumbers which is again a sorted Map that keeps track of all the orders ever placed.

main()

call runSimulation(numCustomers, numCooks, numTables, machineCapacity, randomOrders)

**Pre-Condition**: numCustomers > 0

numCooks > 0

numTable > 0

machineCapacity > 0

randomOrders true or false

**Post-Condition**: All events must be returned correctly

**Pseudo Code**:

runSimulation(int numCustomers, int numCooks, int numTable, int machineCapacity, int randomOrders)

//staring machines

e.g.

Machine Grill = new Machine("Grill", FoodType.burger, machineCapacity);

//Let cooks in

Thread[] cooks = new Thread[numCooks];

for(int i=0; i<cooks.length; i++){

cooks[i] = new Thread(new Cook("Cook"+(i+1)));

}

Starting customer and cook threads

try{

**if**(customers.length <= 0){

simulationFlag = **true**;

interrupt running threads like Cook

}

Join Cook threads

catch(){}

if(simulationFlag)

Shut down all the machines

End Simulation

**Synchronization**

The main concerns in this assignment is

1. Number of Customers at the café must not exceed number of tables in it.
2. Orders must be taken according to customers priority

Instance lock is used on the tablesOccupied variable as it does not have any method.

The methods getPendingOrders and removeFromPendingOrders are Syncronized static methods.