Computer & Systems Engineering Department

CSE 223: Programming ||

Producer-Consumer Simulation

Contributors:

	Name	ID
1	Adel Mahmoud Mohamed Abdelrahman	20010769
2	Mohamed Hassan Sadek Abdelhamid	20011539
3	Mahmoud Attia Mohamed Abdelaziz Zian	20011810
4	Mahmoud Ali Ahmed Ali	20011811

How to run:

- Make sure you downloaded NodeJs and Angular-CLI.
- Extract the compressed project folder.
- Back-end part: Open the “Producer-consumer-Backend” folder using IntelliJ IDE, and run the ProducerConsumerBackendApplication.java class on port 9080. you can change the port from the project resources at application.properties (“server.port = …..”) if the 9080 port was already used in your device but in this case, you will need to change it in all http requests in the angular folder.
- Front-end part: open the “producer-consumer-frontend” folder using visual studio IDE, then open the terminal of the IDE, and write:
    - npm install in the terminal.
    - ng serve --open in the terminal to open the project, on port “http://localhost:4200/”. Note: if you needed to change port 4200 as it was already in use then you will need to change it in the Producer-Consumer-Backend folder in the controller class.

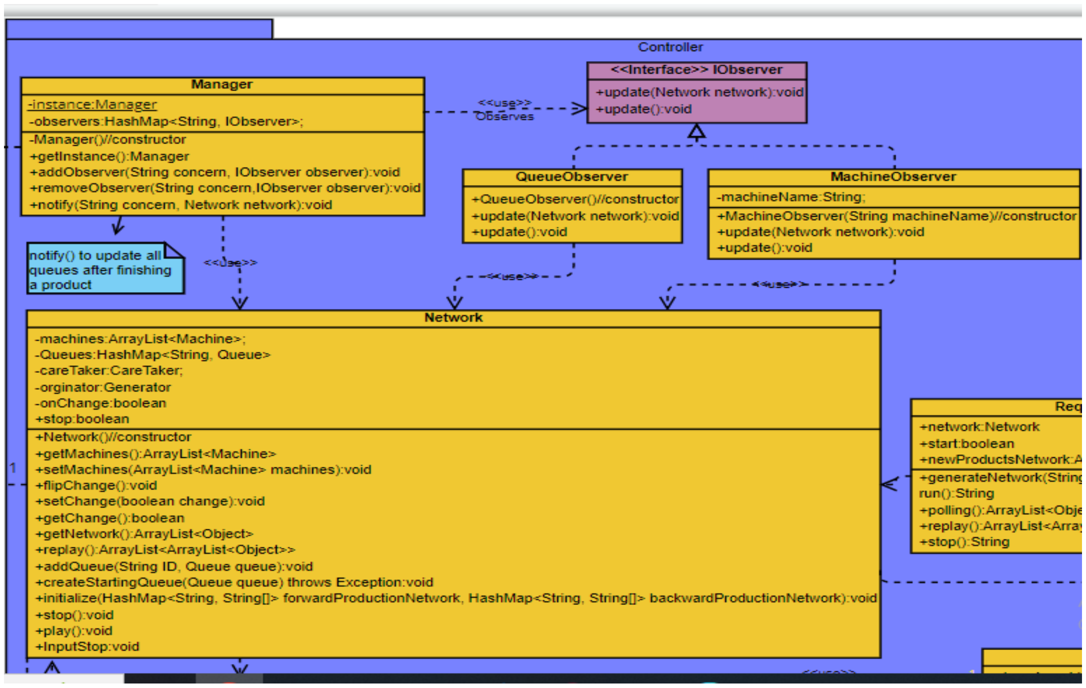
UML diagram:

https://drive.google.com/file/d/1CxZbJAS9B4kF3JgUjZnGSZPclNfNDkTh/view?usp=share_link

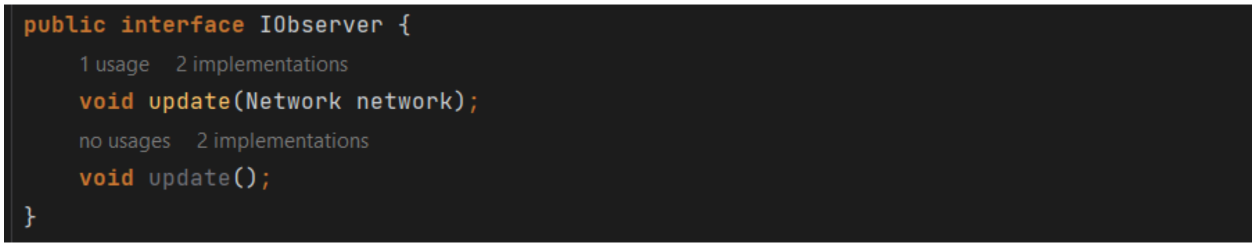
Design Patterns:

1. Observer Design Pattern:

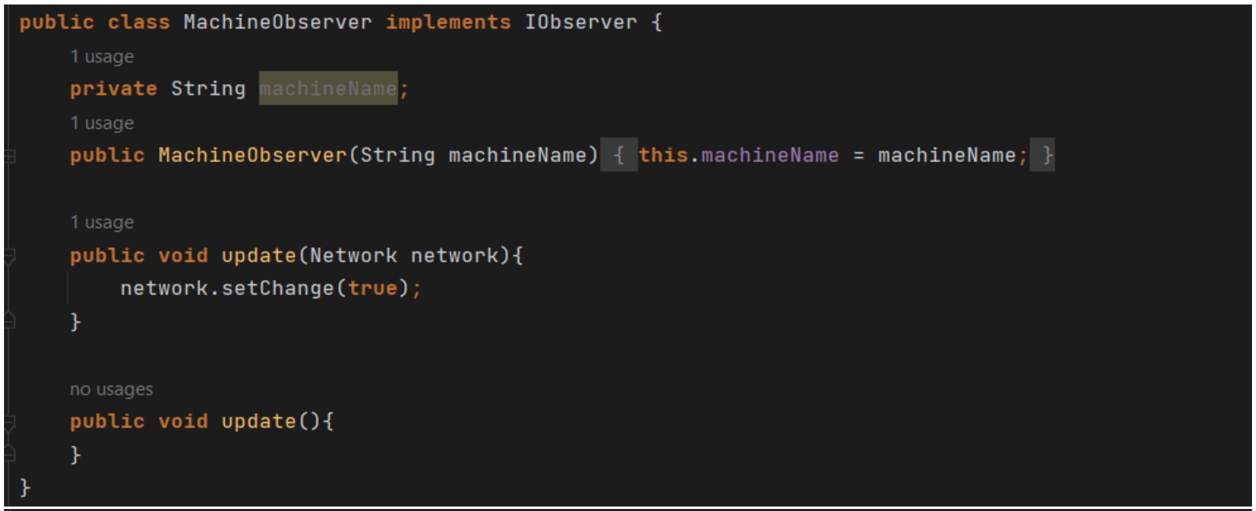
The intent of this design pattern is to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. In our code, we implement it by creating the Interface called "IObserver". Then, two classes called Machine Observer and Queue Observer are the concrete classes that implement the "IObserver" interface. There is a class called "Manager" which observes the changes on the machine and queues, then by calling the function "notify" the queues are notified and updated automatically. The goal of this is that we want to notify and update the state of the machines in the network once the state of any queue has changed and vice versa, and here's the UML for that DP.


Controller
<<Interface>> IObserver
+update(Network network):void
+update():void
Manager
-instance:Manager
-observers:HashMap<String, IObserver>;
-Manager()//constructor
+getInstance():Manager
+addObserver(String concern, IObserver observer):void
+removeObserver(String concern,IObserver observer):void
+notify(String concern, Network network):void
<<use>>
Observes
QueueObserver
+QueueObserver()//constructor
+update(Network network):void
+update():void
MachineObserver
-machineName:String;
+MachineObserver(String machineName)//constructor
+update(Network network):void
+update():void
notify() to update all queues after finishing a product
<<use>>
<<use>>
<<use>>
Network
-machines:ArrayList<Machine>;
-Queues:HashMap<String, Queue>
-careTaker:CareTaker;
-orginator:Generator
-onChange:boolean
+stop:boolean
+Network()//constructor
+getMachines():ArrayList<Machine>
+setMachines(ArrayList<Machine> machines):void
+flipChange():void
+setChange(boolean change):void
+getChange():boolean
+getNetwork():ArrayList<Object>
+replay():ArrayList<ArrayList<Object>>
+addQueue(String ID, Queue queue):void
+createStartingQueue(Queue queue) throws Exception:void
+initialize(HashMap<String, String[]> forwardProductionNetwork, HashMap<String, String[]> backwardProductionNetwork):void
+stop():void
+play():void
+InputStop:void
Req
+network:Network
+start:boolean
+newProductsNetwork:A
+generateNetwork(String
run():String
+polling():ArrayList<Obje
+replay():ArrayList<Array
+stop():String
1
1

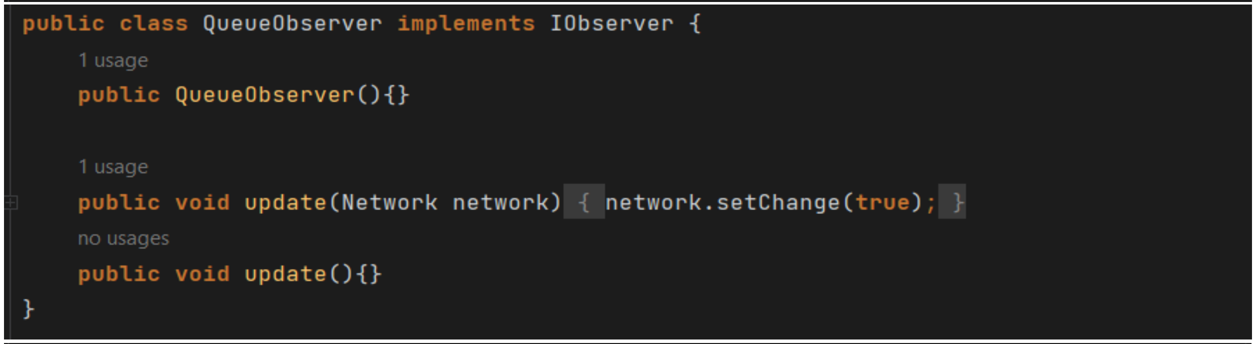
Code snippets:



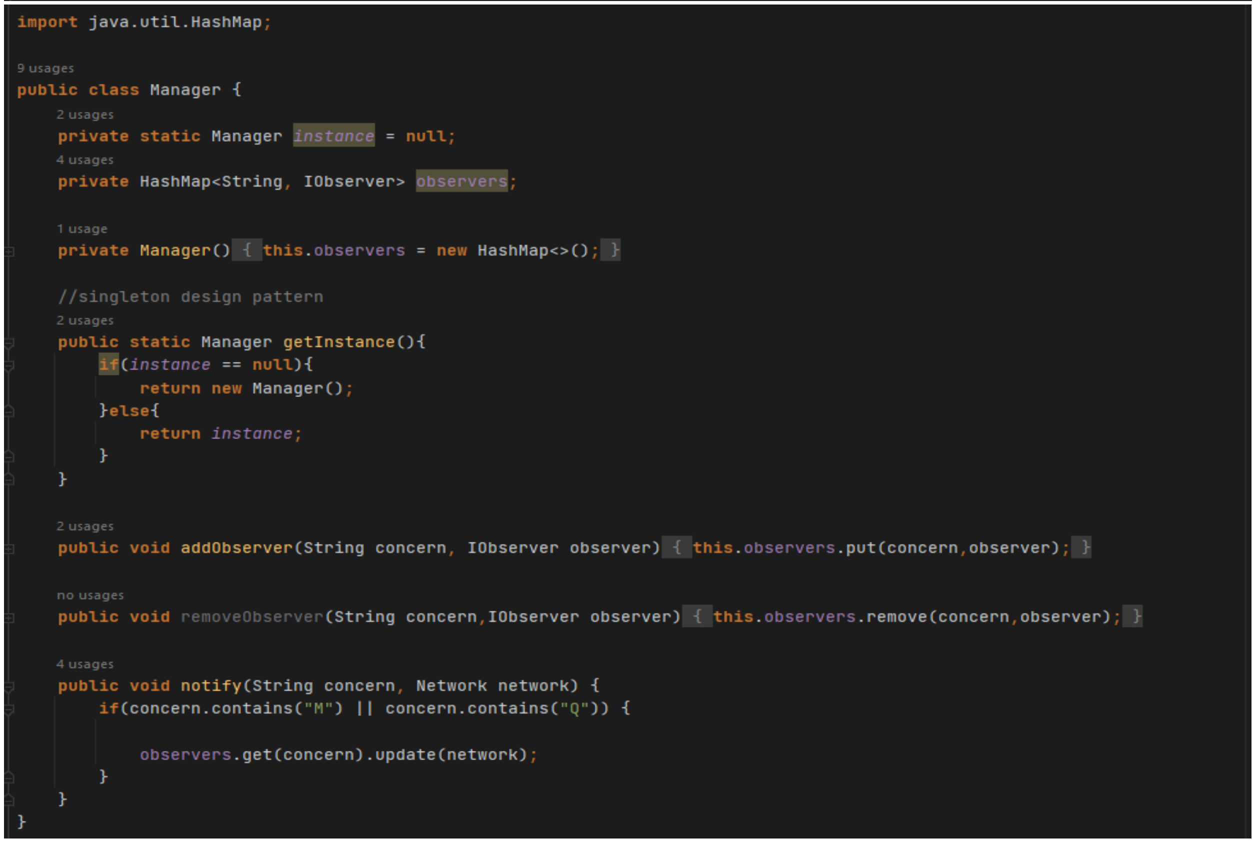
```
public interface IObserver {
    1 usage   2 implementations
    void update(Network network);
    no usages   2 implementations
    void update();
}
```



```
public class MachineObserver implements IObserver {
    1 usage
    private String machineName;
    1 usage
    public MachineObserver(String machineName) { this.machineName = machineName; }

    1 usage
    public void update(Network network){
        network.setChange(true);
    }

    no usages
    public void update(){
    }
}
```

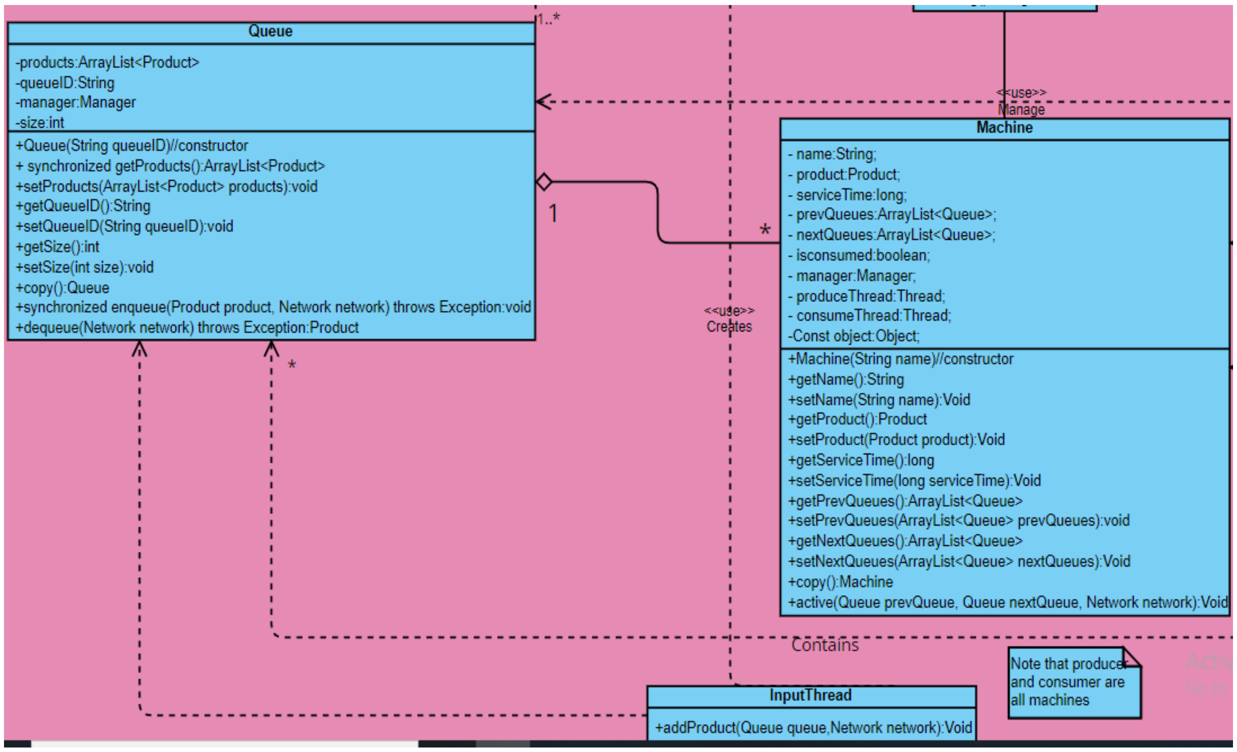


```
public class QueueObserver implements IObserver {
    1 usage
    public QueueObserver(){}

    1 usage
    public void update(Network network) { network.setChange(true); }
    no usages
    public void update(){}
}
```

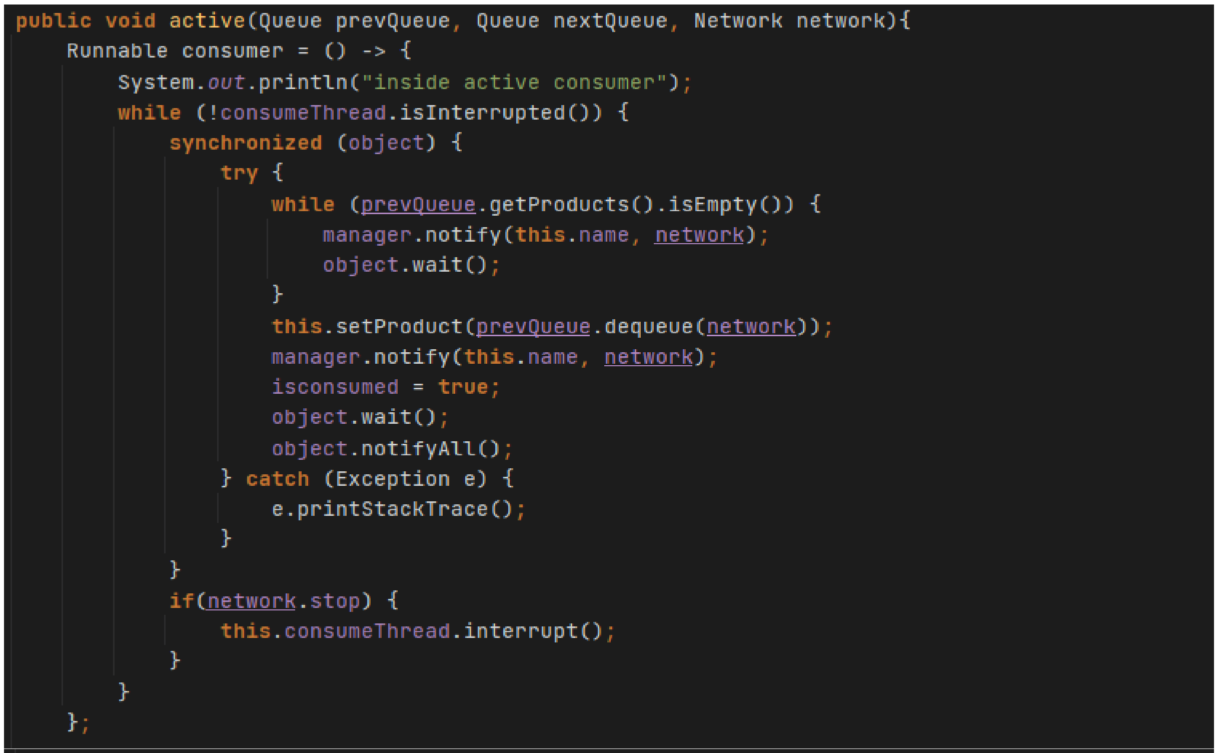


```
import java.util.HashMap;

9 usages
public class Manager {
    2 usages
    private static Manager instance = null;
    4 usages
    private HashMap<String, IObserver> observers;

    1 usage
    private Manager() { this.observers = new HashMap<>(); }

    //singleton design pattern
    2 usages
    public static Manager getInstance(){
        if(instance == null){
            return new Manager();
        }else{
            return instance;
        }
    }

    2 usages
    public void addObserver(String concern, IObserver observer) { this.observers.put(concern,observer); }

    no usages
    public void removeObserver(String concern,IObserver observer) { this.observers.remove(concern,observer); }

    4 usages
    public void notify(String concern, Network network) {
        if(concern.contains("M") || concern.contains("Q")) {

            observers.get(concern).update(network);
        }
    }
}
```

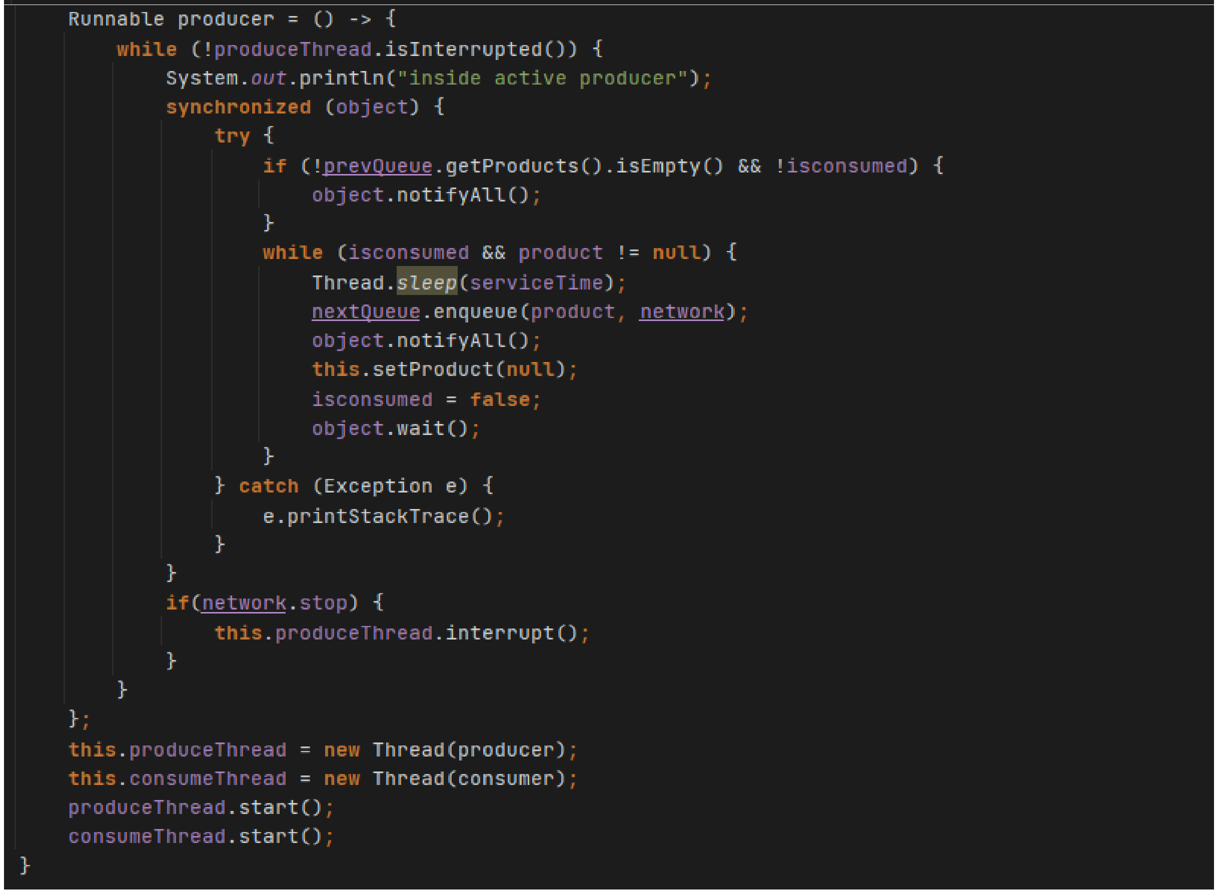
2. **Producer Consumer Design Pattern:**
   The intent is to organize the communication between the producer and consumer objects **(the queues and machines here in our context)** and control the multithreaded flow in the network. Briefly, we used it by creating two classes which are “Queue” and “Machine”. The Network contains a list of machines and queues in a certain order. There may be a multi-level of machines and queues the process started by queuing the products to be produced in “Q0”, then each machine takes a single product only and the rest wait in the queue until any of the machines finished, then the first product in the queue will enter the first available machine and, the second entered the next machine and so on in all levels until all products gathered in the “Qn”(i.e. Last level of the network) and here’s the UML for that DP.


Queue
-products:ArrayList<Product>
-queueID:String
-manager:Manager
-size:int
+Queue(String queueID)//constructor
+ synchronized getProducts():ArrayList<Product>
+setProducts(ArrayList<Product> products):void
+getQueueID():String
+setQueueID(String queueID):void
+getSize():int
+setSize(int size):void
+copy():Queue
+synchronized enqueue(Product product, Network network) throws Exception:void
+dequeue(Network network) throws Exception:Product
1..*
1
*
*
<<use>>
Manage
<<use>>
Creates
Machine
- name:String;
- product:Product;
- serviceTime:long;
- prevQueues:ArrayList<Queue>;
- nextQueues:ArrayList<Queue>;
- isconsumed:boolean;
- manager:Manager;
- produceThread:Thread;
- consumeThread:Thread;
-Const object:Object;
+Machine(String name)//constructor
+getName():String
+setName(String name):Void
+getProduct():Product
+setProduct(Product product):Void
+getServiceTime():long
+setServiceTime(long serviceTime):Void
+getPrevQueues():ArrayList<Queue>
+setPrevQueues(ArrayList<Queue> prevQueues):void
+getNextQueues():ArrayList<Queue>
+setNextQueues(ArrayList<Queue> nextQueues):Void
+copy():Machine
+active(Queue prevQueue, Queue nextQueue, Network network):Void
Contains
Note that producer and consumer are all machines
InputThread
+addProduct(Queue queue,Network network):Void

Code Snippets:

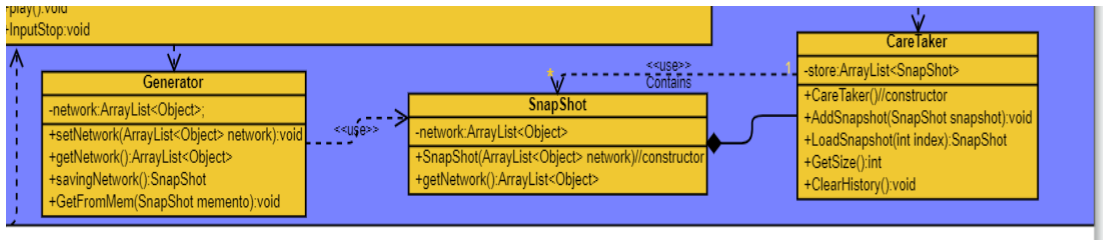


```java
public void active(Queue prevQueue, Queue nextQueue, Network network){
    Runnable consumer = () -> {
        System.out.println("inside active consumer");
        while (!consumeThread.isInterrupted()) {
            synchronized (object) {
                try {
                    while (prevQueue.getProducts().isEmpty()) {
                        manager.notify(this.name, network);
                        object.wait();
                    }
                    this.setProduct(prevQueue.dequeue(network));
                    manager.notify(this.name, network);
                    isconsumed = true;
                    object.wait();
                    object.notifyAll();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
            if(network.stop) {
                this.consumeThread.interrupt();
            }
        }
    };
```

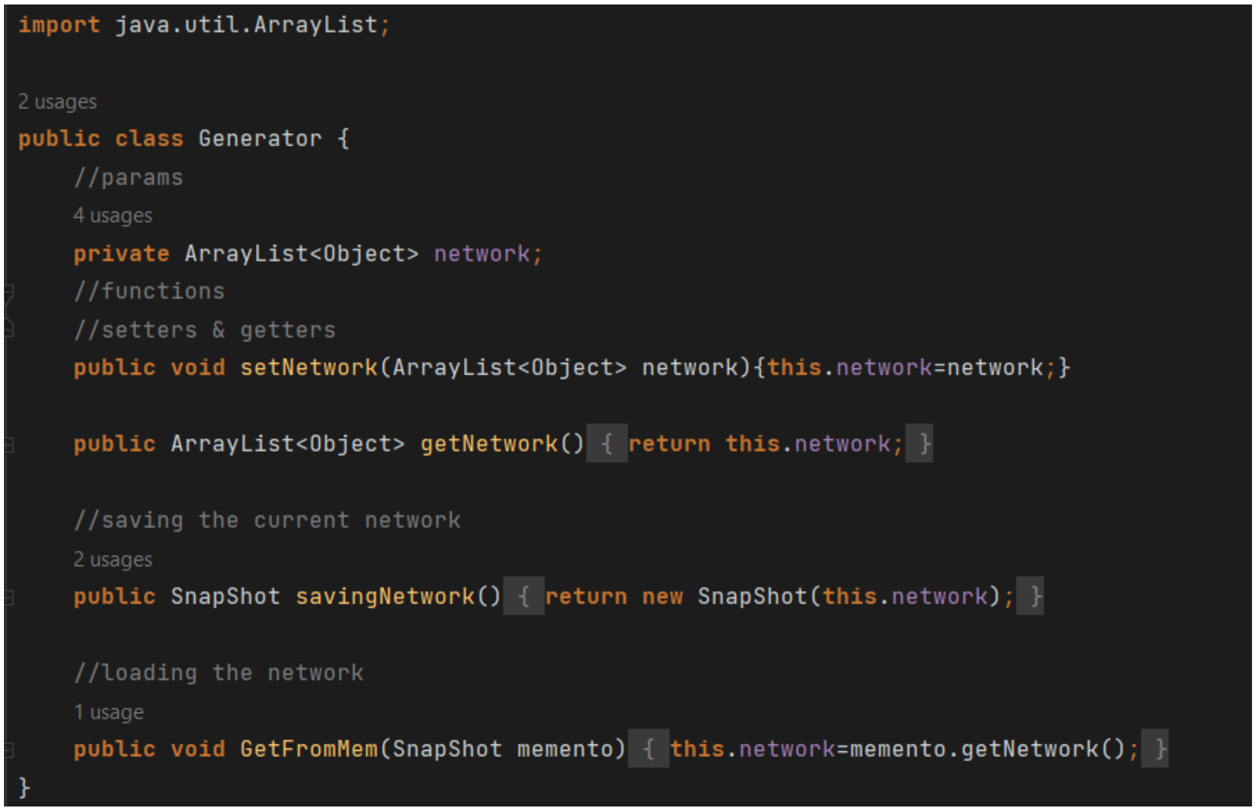


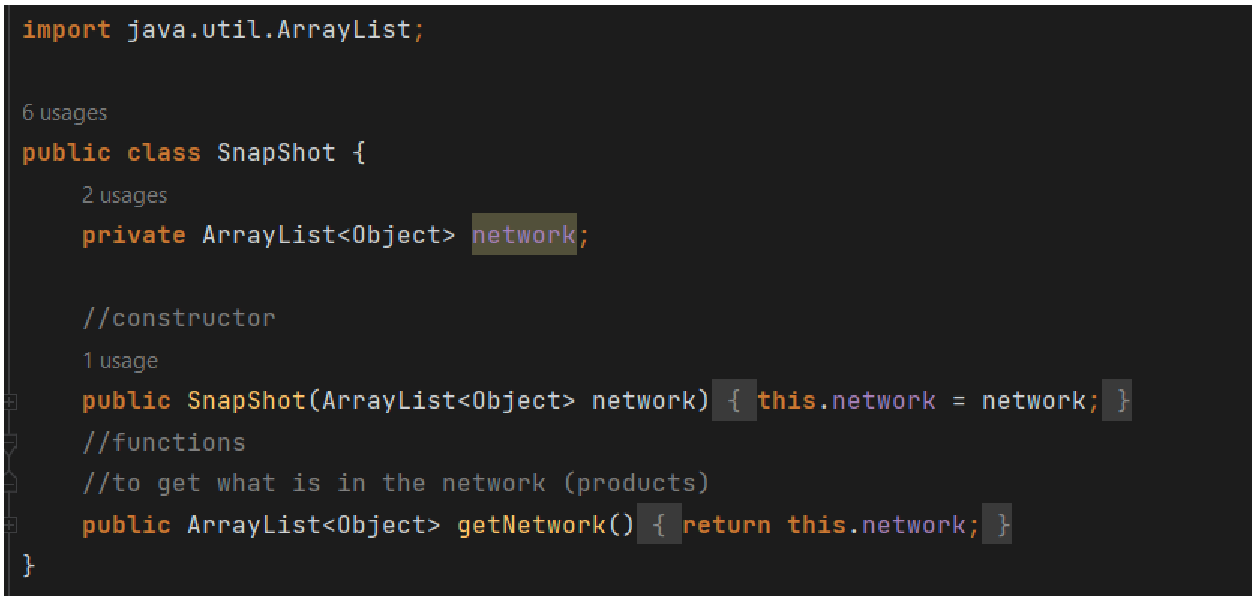
```java
    Runnable producer = () -> {
        while (!produceThread.isInterrupted()) {
            System.out.println("inside active producer");
            synchronized (object) {
                try {
                    if (!prevQueue.getProducts().isEmpty() && !isconsumed) {
                        object.notifyAll();
                    }
                    while (isconsumed && product != null) {
                        Thread.sleep(serviceTime);
                        nextQueue.enqueue(product, network);
                        object.notifyAll();
                        this.setProduct(null);
                        isconsumed = false;
                        object.wait();
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
            if(network.stop) {
                this.produceThread.interrupt();
            }
        }
    };
    this.produceThread = new Thread(producer);
    this.consumeThread = new Thread(consumer);
    produceThread.start();
    consumeThread.start();
}
```

3. **SnapShot:**
   We use this design pattern in order to restore(keep a memento or in other words take a snapshot) of the network each time a change of the state occurs to help us replay the simulation from the beginning until the exact moment we've stopped at. The class "Generator", "CareTaker" and "SnapShot" are the main elements to apply this design pattern. "Generator" communicates with "SnapShot" either to store the current states of the network or get the previous states of the current network. "CareTaker" is our storage for the networks it keeps a list of mementos and supports the different operations on this list like adding a memento to the list getting all mementos etc.

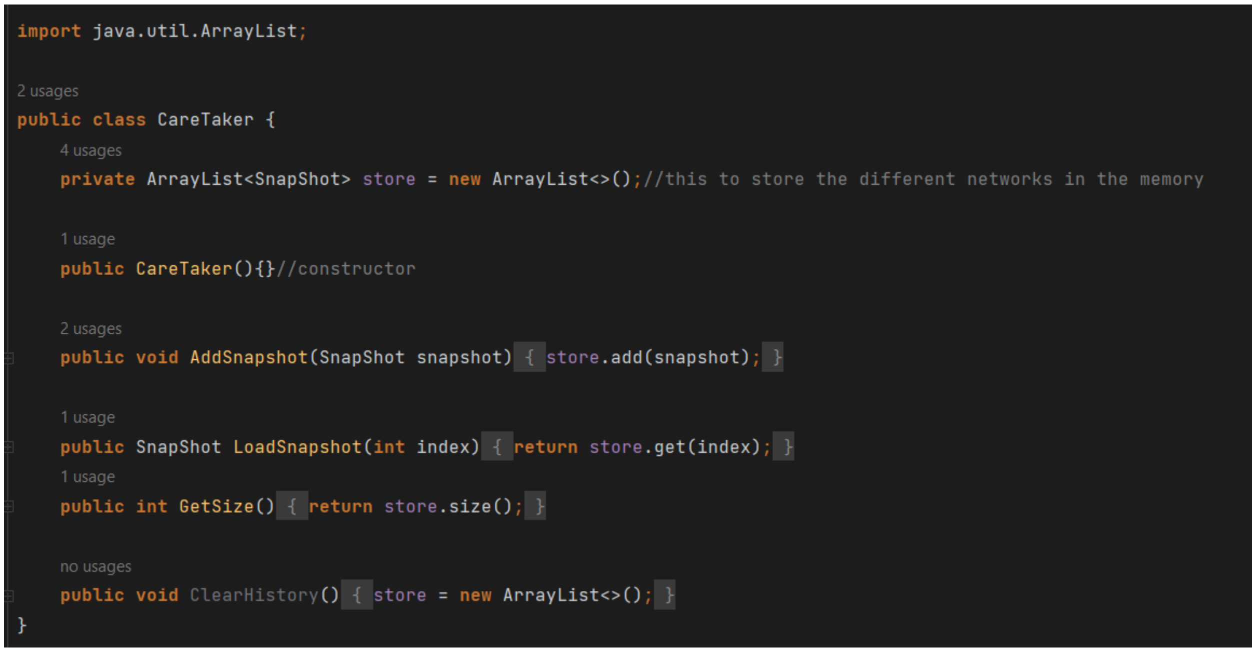

+play():void
+InputStop:void
Generator
-network:ArrayList<Object>;
+setNetwork(ArrayList<Object> network):void
+getNetwork():ArrayList<Object>
+savingNetwork():SnapShot
+GetFromMem(SnapShot memento):void
<<use>>
SnapShot
-network:ArrayList<Object>
+SnapShot(ArrayList<Object> network)//constructor
+getNetwork():ArrayList<Object>
*
<<use>>
Contains
1
CareTaker
-store:ArrayList<SnapShot>
+CareTaker()//constructor
+AddSnapshot(SnapShot snapshot):void
+LoadSnapshot(int index):SnapShot
+GetSize():int
+ClearHistory():void

Code Snippets:


```
import java.util.ArrayList;

2 usages
public class Generator {
    //params
    4 usages
    private ArrayList<Object> network;
    //functions
    //setters & getters
    public void setNetwork(ArrayList<Object> network){this.network=network;}

    public ArrayList<Object> getNetwork() { return this.network; }

    //saving the current network
    2 usages
    public SnapShot savingNetwork() { return new SnapShot(this.network); }

    //loading the network
    1 usage
    public void GetFromMem(SnapShot memento) { this.network=memento.getNetwork(); }
}
```

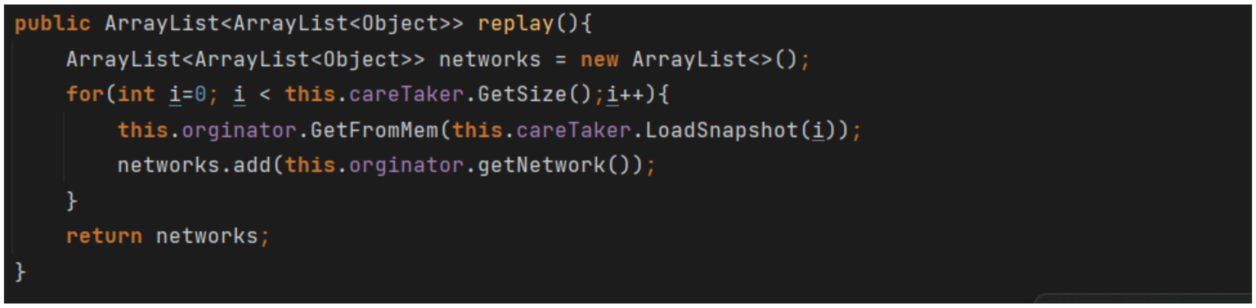


```
import java.util.ArrayList;

6 usages
public class SnapShot {
    2 usages
    private ArrayList<Object> network;

    //constructor
    1 usage
    public SnapShot(ArrayList<Object> network) { this.network = network; }
    //functions
    //to get what is in the network (products)
    public ArrayList<Object> getNetwork() { return this.network; }
}
```



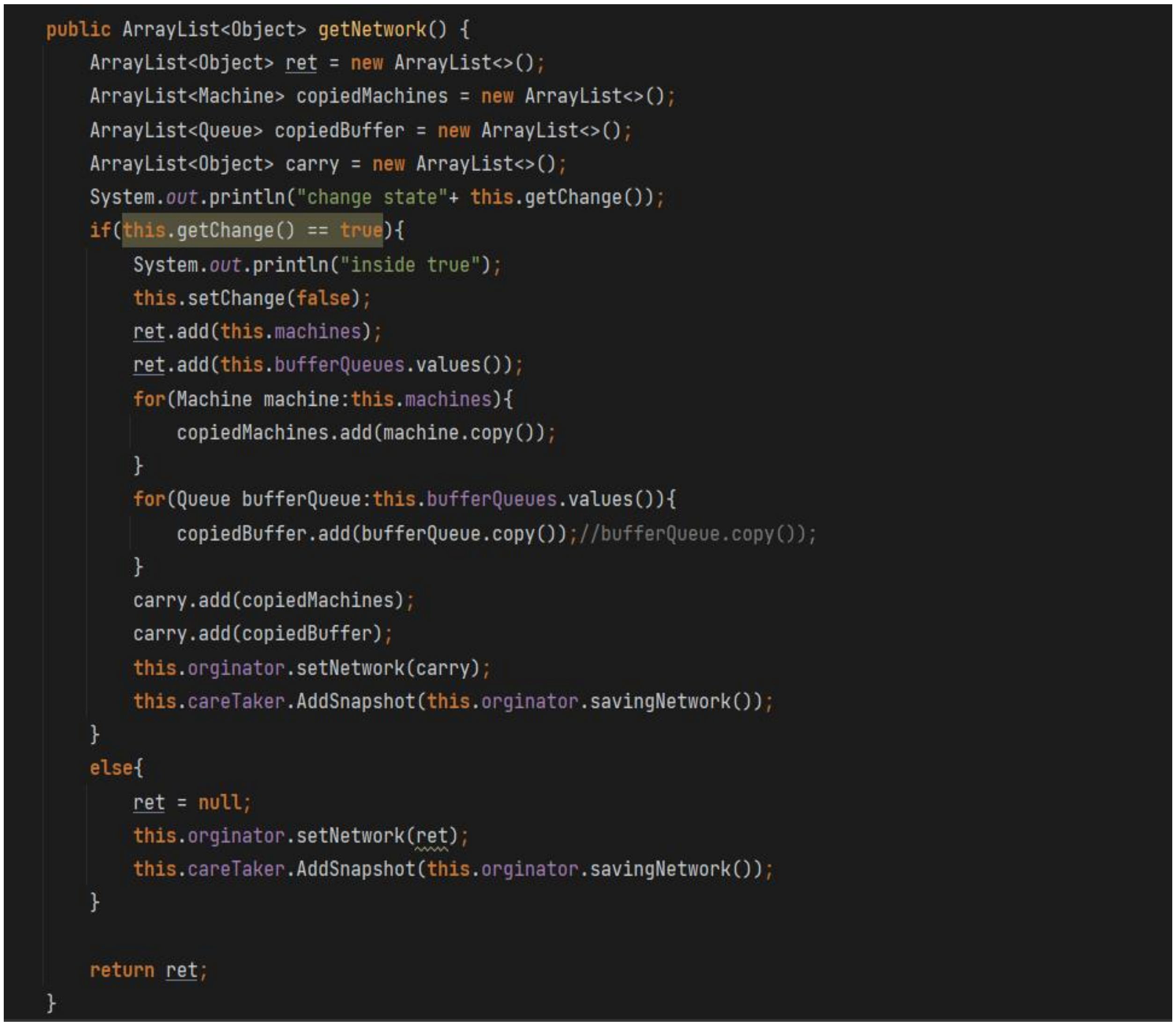
```
import java.util.ArrayList;

2 usages
public class CareTaker {
    4 usages
    private ArrayList<SnapShot> store = new ArrayList<>();//this to store the different networks in the memory

    1 usage
    public CareTaker(){}//constructor

    2 usages
    public void AddSnapshot(SnapShot snapshot) { store.add(snapshot); }

    1 usage
    public SnapShot LoadSnapshot(int index) { return store.get(index); }
    1 usage
    public int GetSize() { return store.size(); }

    no usages
    public void ClearHistory() { store = new ArrayList<>(); }
}
```

replay function in the network function:



```
public ArrayList<ArrayList<Object>> replay(){
    ArrayList<ArrayList<Object>> networks = new ArrayList<>();
    for(int i=0; i < this.careTaker.GetSize();i++){
        this.orginator.GetFromMem(this.careTaker.LoadSnapshot(i));
        networks.add(this.orginator.getNetwork());
    }
    return networks;
}
```

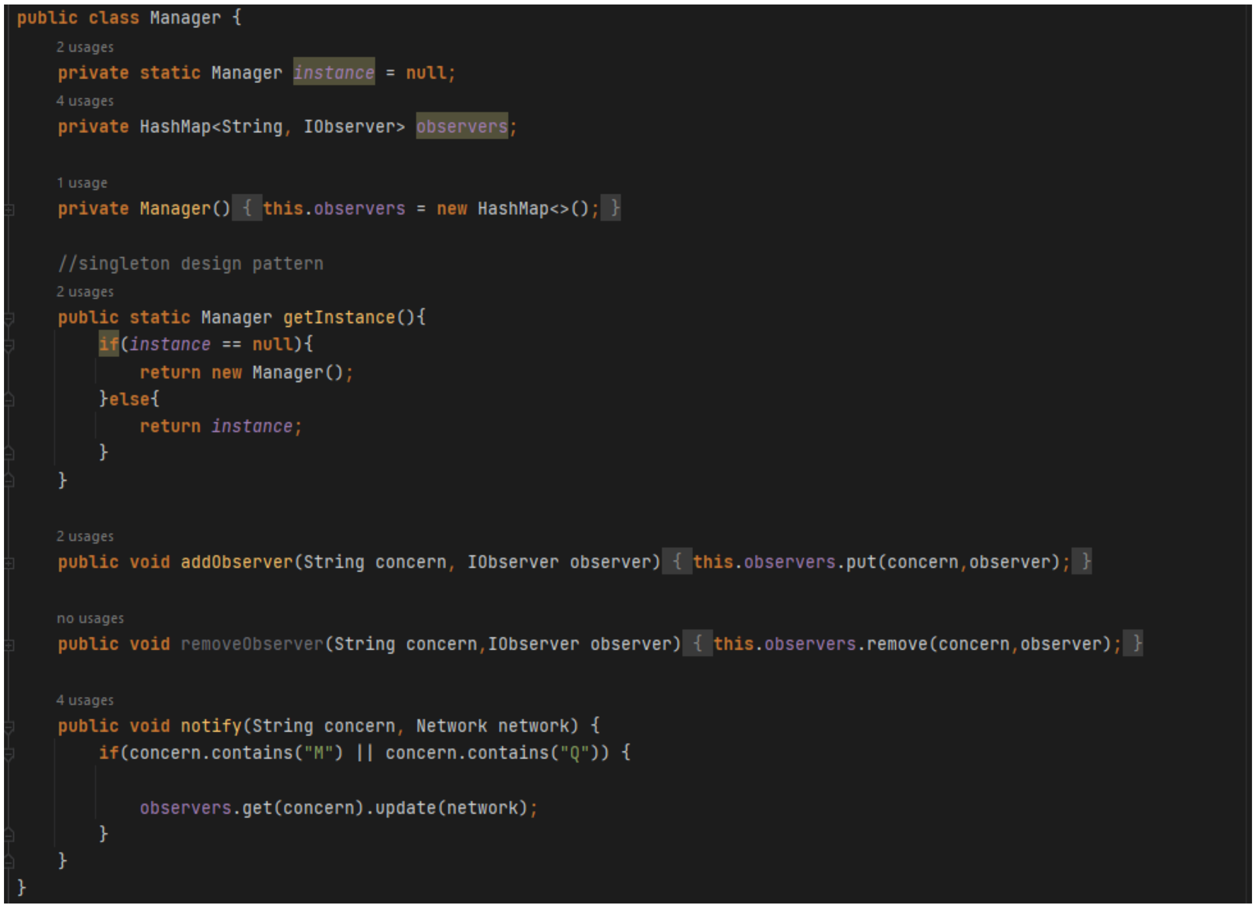
getNetwork function in the network class:



```java
public ArrayList<Object> getNetwork() {
    ArrayList<Object> ret = new ArrayList<>();
    ArrayList<Machine> copiedMachines = new ArrayList<>();
    ArrayList<Queue> copiedBuffer = new ArrayList<>();
    ArrayList<Object> carry = new ArrayList<>();
    System.out.println("change state"+ this.getChange());
    if(this.getChange() == true){
        System.out.println("inside true");
        this.setChange(false);
        ret.add(this.machines);
        ret.add(this.bufferQueues.values());
        for(Machine machine:this.machines){
            copiedMachines.add(machine.copy());
        }
        for(Queue bufferQueue:this.bufferQueues.values()){
            copiedBuffer.add(bufferQueue.copy());//bufferQueue.copy());
        }
        carry.add(copiedMachines);
        carry.add(copiedBuffer);
        this.orginator.setNetwork(carry);
        this.careTaker.AddSnapshot(this.orginator.savingNetwork());
    }
    else{
        ret = null;
        this.orginator.setNetwork(ret);
        this.careTaker.AddSnapshot(this.orginator.savingNetwork());
    }

    return ret;
}
```

4. **Singleton:**
   we used it in order to create a single instance from “Manager” Class to reuse it after creation-because it’s kind of a huge class so it costs a lot creating a new instance every now and then-to be able to manage the resources such as “AddingObserver”, ”RemovingObserver” and also notify the whole network if there is any change in the states.

Code Snippets:



```
public class Manager {
    2 usages
    private static Manager instance = null;
    4 usages
    private HashMap<String, IObserver> observers;

    1 usage
    private Manager() { this.observers = new HashMap<>(); }

    //singleton design pattern
    2 usages
    public static Manager getInstance(){
        if(instance == null){
            return new Manager();
        }else{
            return instance;
        }
    }

    2 usages
    public void addObserver(String concern, IObserver observer) { this.observers.put(concern,observer); }

    no usages
    public void removeObserver(String concern,IObserver observer) { this.observers.remove(concern,observer); }

    4 usages
    public void notify(String concern, Network network) {
        if(concern.contains("M") || concern.contains("Q")) {

            observers.get(concern).update(network);
        }
    }
}
```

System Design:

- Architecture: Each machine must be between two queues (queue to store the products until the machine has finished and queue to put the products after manufacturing) and connected with the lines
- Components and their interfaces: Queues (Rectangle Form) and Machines (Circular Form)
- Data: Number of products waiting in each Queue and the queues connected with machines (incoming and outcoming)
- Link for UML:https://drive.google.com/file/d/1CxZbJAS9B4kF3JgUjZnGSZPclNfNDkTh/view?usp=share_link

Design decisions:

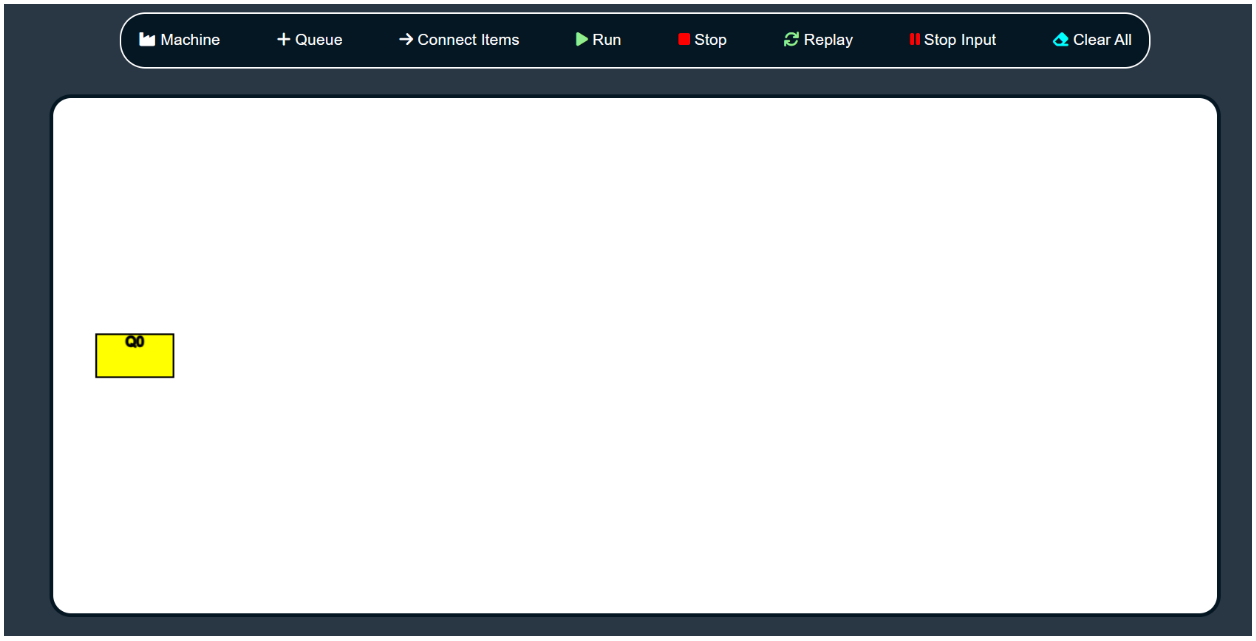
- We have used the “MVC” design in order to facilitate the communication among the three layers:
    - Model: Controlling the data.
    - View: Related with input from the user in the project UI.
    - Controller: which takes the Model layer and manipulates the data to output the suitable results to users through the View interface.
- The rate of the input products in Q0 is generated randomly-as it wasn’t required in the lab requirements to have the user enter a certain number of products to be served- by picking a random number from 40 up to 600.
- The queues are colored in yellow and the machines in gray.
- The service time of each machine is picked randomly and remains as it is during the whole process, and we’ve had it displayed in each machine.
- The network must be closed hence it starts by Q0 (which is added by default) and it must end with a destination queue for all the products to be accumulated till the simulation ends.

How the system works:

- The number of products is random and initially the default queue gets a random input rate of the products and through this queue, the products are distributed over the first level of machine(s).
- Each machine in the network has a random service time that remains the same till the process ends.
- Each machine serves one product at a time and the speed of service is according to the machine’s service time.
- And from the first level machine(s) the products are transferred along the network up to the second level and so on until the products are completely serviced and then they’re stored in the final buffer.
- The Observer DP is used to for notifying the network elements each state change and according to that other states are updated.
- The memento DP is used to keep a memento of the network each state change for the replay feature.
- The concurrency DP is used to control the multi-threaded flow along the network.

User Guide:

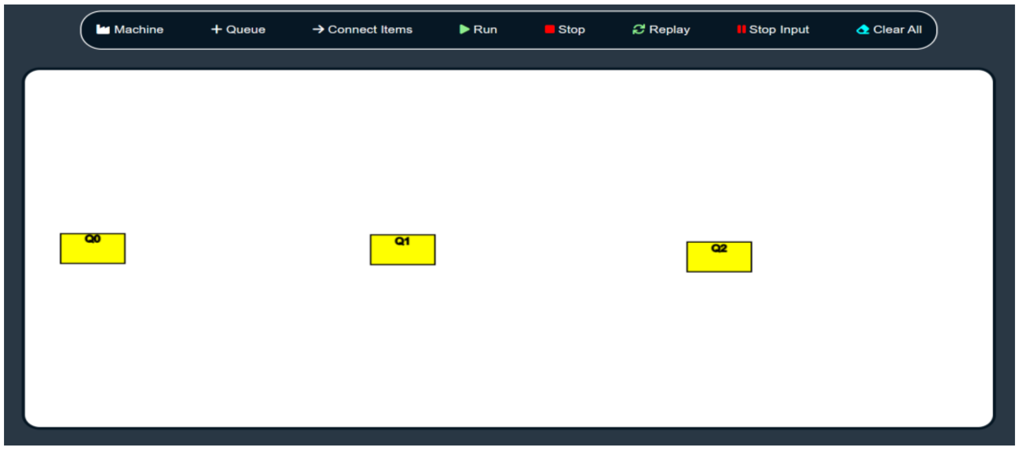
App design:



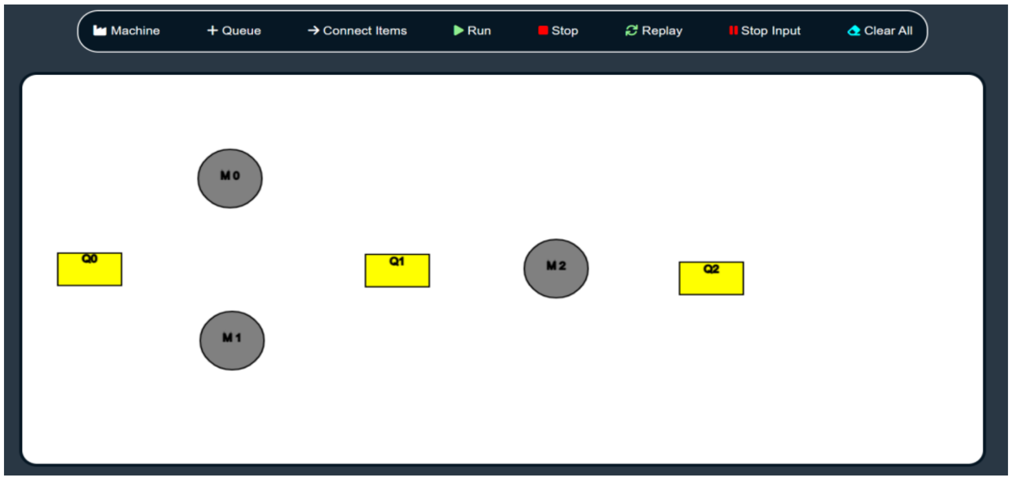
- The queue Q0 is added by default initially.
- Machine button: to add a new machine.
- +Queue button: to add a new queue.
- Connect items: to connect a queue to a machine in any direction. To connect between a queue and a machine first press the button and click on the start shape and hold, dragging the arrow to the destination shape.
- Run button: to start the simulation.
- Each M has a random service time and can serve one product at a time.
- Each M changes its color to the color of the product being processed.
- Each Q displays the number of products waiting.
- Stop button: to stop the simulation.
- Replay button: to replay the simulation.
- Stop input button: to stop products arriving at Q0.
- Clear all button: to clear all the network in order to add a new network.
- In case a machine is not connected to at least one queue at both directions this error message appears.

Snapshots of the UI

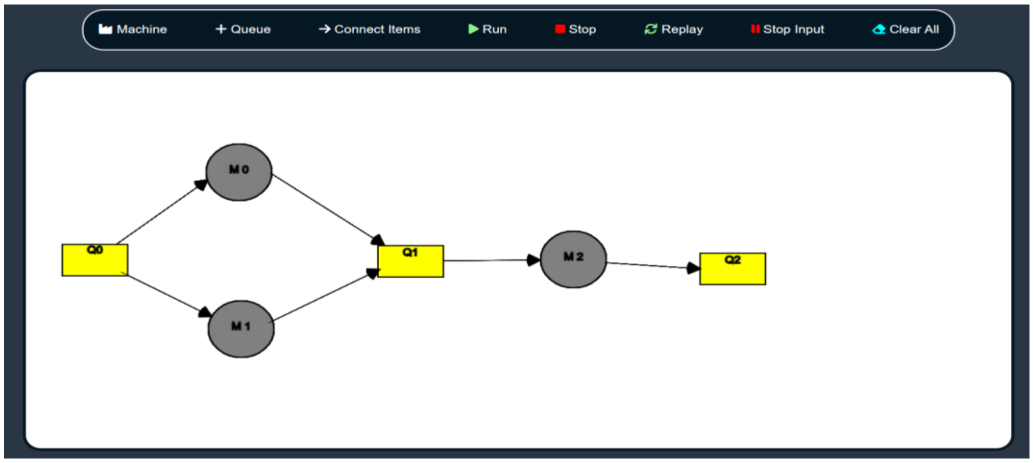
- Adding Q1, Q2, Q3:



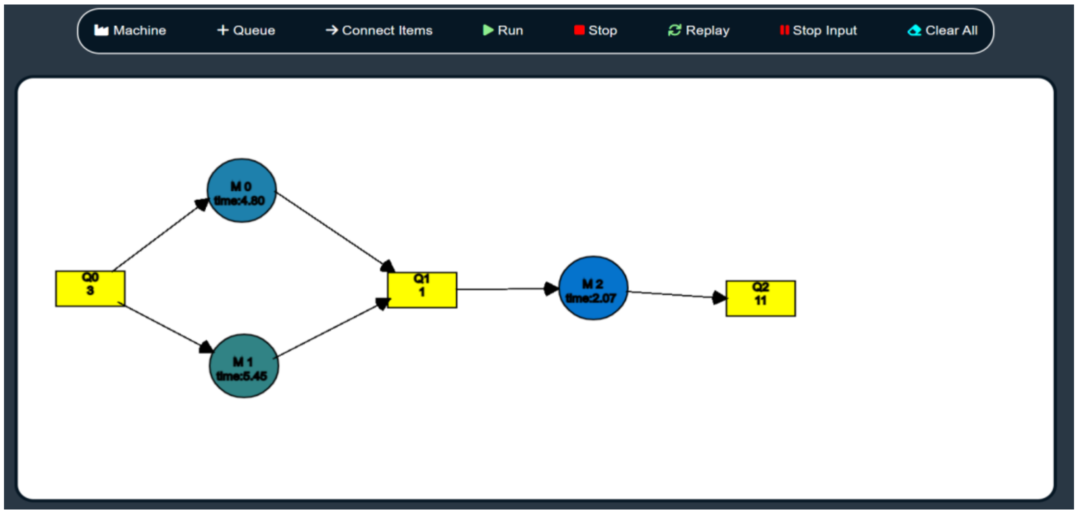
- Adding M0, M1, M2:



- Connecting the machines and queues:



- Here is a screenshot during the simulation:



- For better understanding please watch this video:

Producer Consumer Video-click here