Name	ID
Mohamed Abdelrahman Anwar	20011634
Hussein Mohamed Mansour	20010499
Amr Ahmed Abd ElAzim	20011037
Hussien Khaled Hussien Elsaid	20010494

Programming 2

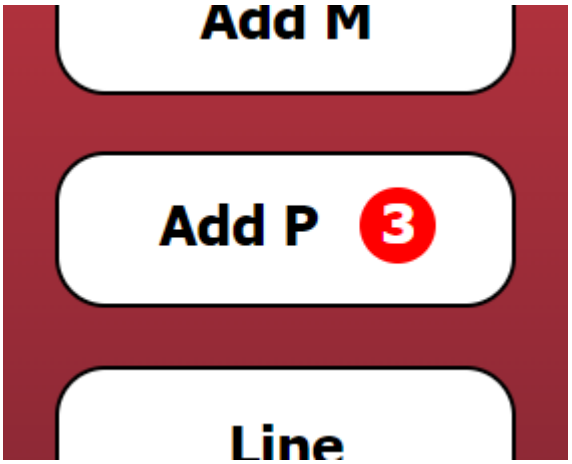
Assignment 5 Report

Problem Statement:

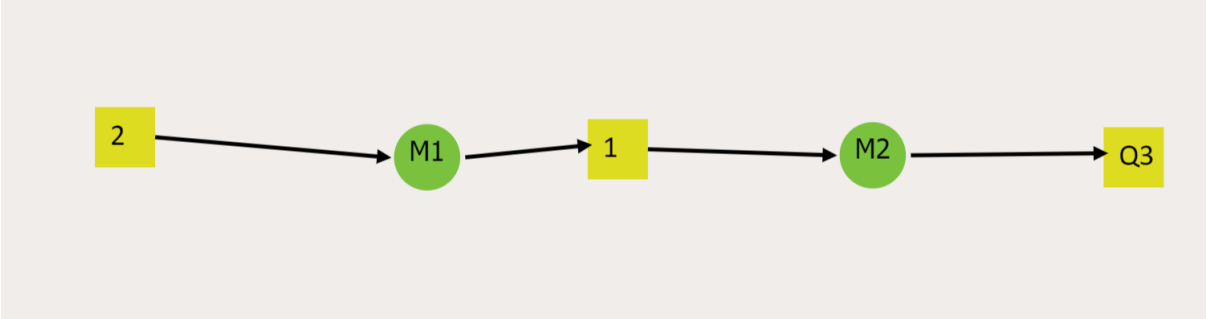
An assembly line that produces different products consists of different processing machines Ms that are responsible for processing the product at different stages and queue Qs to handle product movement between different processing stages. In this assignment, we will develop a simulation program to simulate this production line as a queuing network.

Design choices:

- Whenever the user wants to add a product he clicks on the Add P button and for visibility the number of products added it is declared beside it in a red box.



- The queues and machines are draggable at the first before connecting them with an arrow.
- Before running, the machines and queues are in default names and color. After running, the queue name is replaced with the number of elements in each queue.

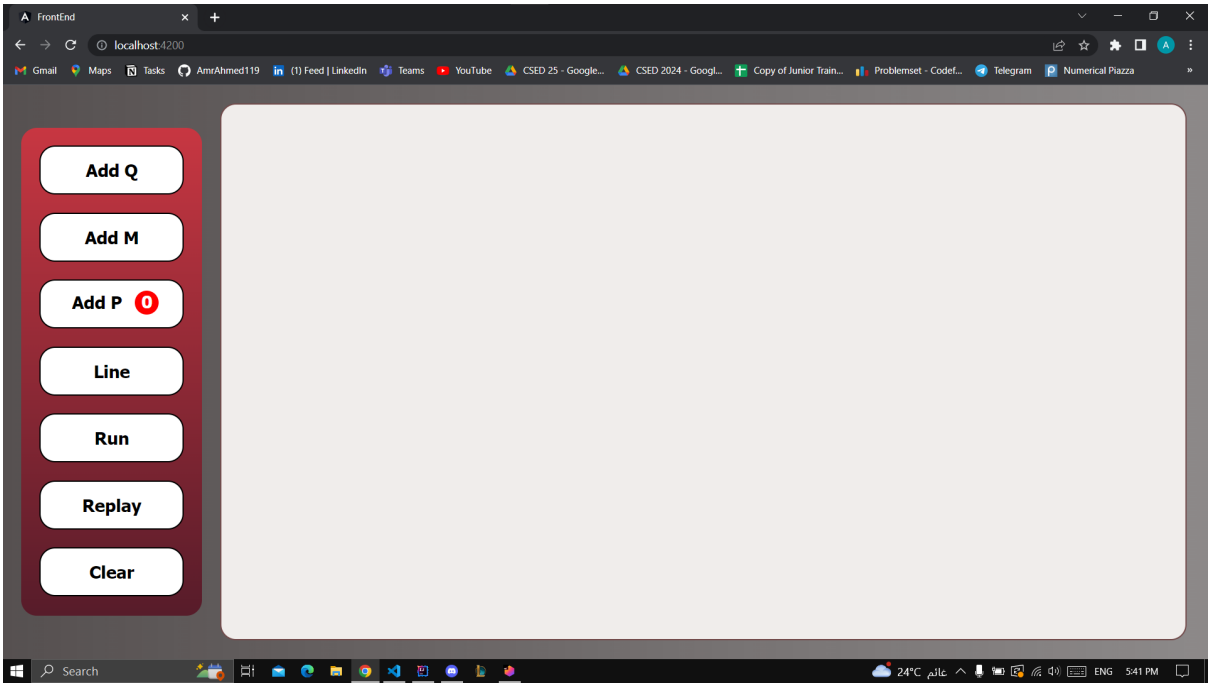


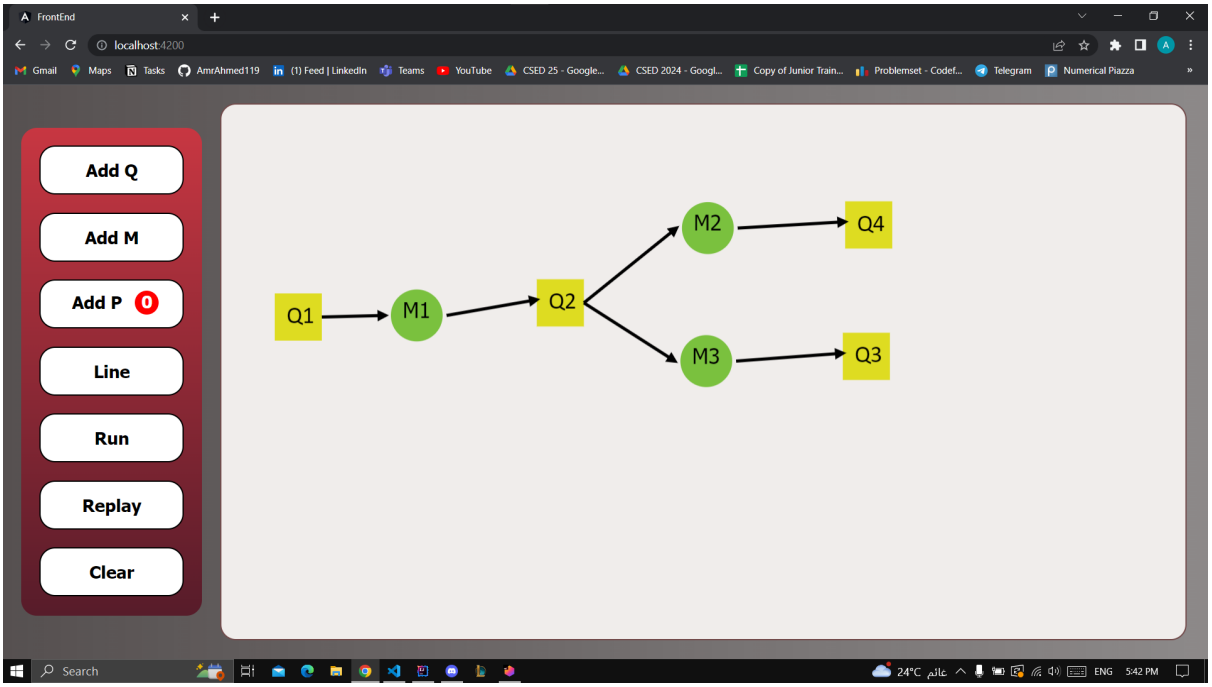
- When the machine is working, the machine color is toggling between the default color and the generated color for the product.
- We have used Konva library for drawing shapes.

UI Snapshots:

The main user interface contains two main windows, the left screen which contains all the buttons needed corresponds to an action.

The main board where all the queues and the machines are placed.

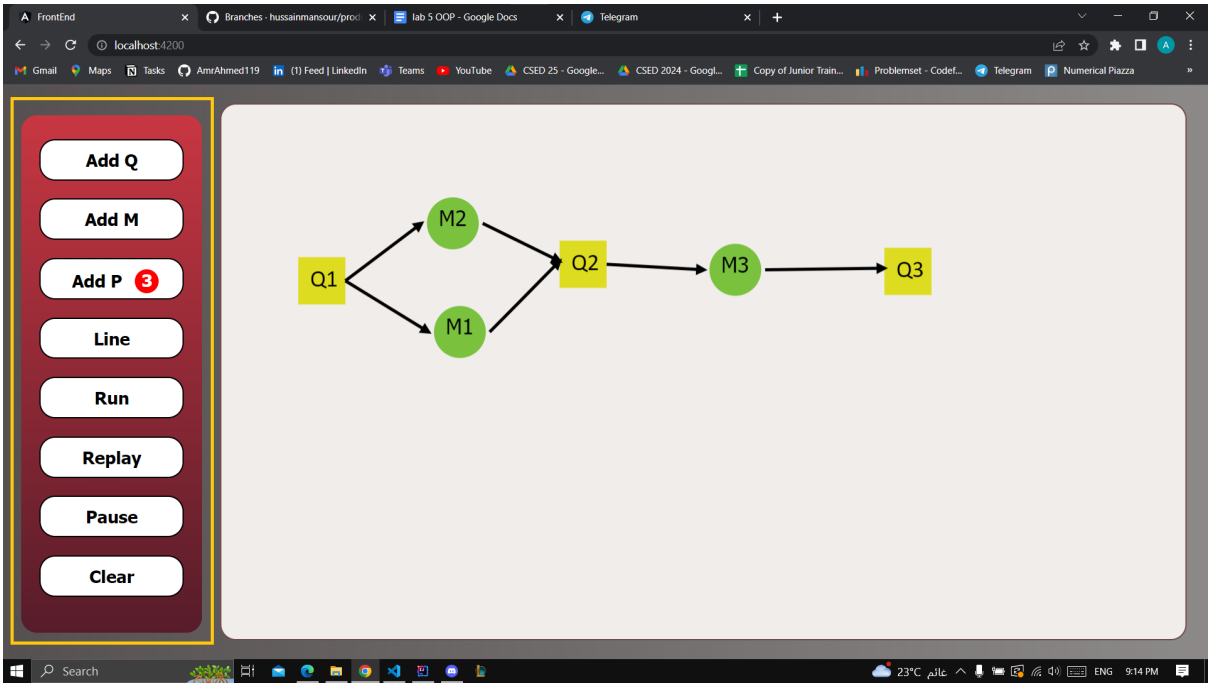




This is the shape when queues and machines are added.

How system works:

- you can add a queue from the Add Q button.
- you can add a machine from the Add M button.
- you can add products from the Add p button and the number of products added is beside the button.
- To connect the machine and queue you click on the line button and then click the machine and queue you want to connect and an arrow will be directed from source to target you selected.
- to run this system and start simulation press the run button.
- to replay this system press the system button.
- to clear the system to start a new system press clear button.
- To pause the source queue Q1 from producing products into machines press the pause button.



Design patterns used:

1. **producer consumer pattern:** in which it is used to coordinate the asynchronous production and consumption of information.

```
    public void notifyQueueSize(MyQueue queue){
        JSONObject json = new JSONObject();
        json.put("id",queue.getId());
        json.put("size",queue.getSize());
        json.put("operation","updateQueueSize");
        webSocketService.notifyFrontEnd(json.toString());
    }

    public void notifyMachineRunning(){
        JSONObject json = new JSONObject();
        json.put("id",this.getId());
        var array = new JSONArray();
        array.put(product.getColor().getRed());
        array.put(product.getColor().getGreen());
        array.put(product.getColor().getBlue());
        json.put("color",array);
        json.put("operation","machineRunning");
        webSocketService.notifyFrontEnd(json.toString());
    }

    public void notifyMachineFinished(){
        JSONObject json = new JSONObject();
        json.put("operation","machineFinished");
        json.put("id",this.getId());
        webSocketService.notifyFrontEnd(json.toString());
    }

}
```

```
                    System.out.println("jjjjjjjjjjjjjjjjjj");
                    nextQueue.put(product);
                    notifyQueueSize(nextQueue);
                    notifyAllObservers();
                    product = null;
                    //------------------------
                    // break;
                }
            }
        }
    };
    System.out.println("CREATED THREAD");
    consumerThread = new Thread(consumer);
    consumerThread.start();
    //producerThread.start();
    }

public void producer() {
    if (product != null && !running){
        System.out.println("jjjjjjjjjjjjjjjjjj");
        nextQueue.put(product);
        notifyQueueSize(nextQueue);
        notifyAllObservers();
        product = null;
    }
    //consumerThread.start();
}


public void notifyQueueSize(MyQueue queue){
    JSONObject json = new JSONObject();
    json.put("id", queue.getId());
```

```java
public MyQueue getNextQueue() { return nextQueue; }

public void setNextQueue(MyQueue nextQueue) { this.nextQueue = nextQueue; }


public void notifyAllObservers() { nextQueue.update(); }

public void activate(){

    Runnable consumer = () -> {
        while(true) {
            for (MyQueue queue : prevQueues) {
                if (!queue.isEmpty()) {
                    product = queue.get();
                    if (product == null) continue;
                    running = true;
                    notifyMachineRunning();
                    notifyQueueSize(queue);
                    System.out.println(serviceTime);
                    try {
                        Thread.sleep(serviceTime);
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                    running = false;
                    notifyMachineFinished();
                    // producerThread.start();
                    //-------------------
                    System.out.println("jjjjjjjjjjjjjjjjjj");
                    nextQueue.put(product);
```

```java
    public String getId() { return id; }
    public void setId(String id) { this.id = id; }
    public Product getProduct() { return product; }
    public void setProduct(Product product) { this.product = product; }
    public Thread getProducerThread() { return producerThread; }
    public void setProducerThread(Thread producerThread) { this.producerThread = produce
    public Thread getConsumerThread() { return consumerThread; }
    public void setConsumerThread(Thread consumerThread) { this.consumerThread = consume
    public List<MyQueue> getPrevQueues() { return prevQueues; }
    public void setPrevQueues(List<MyQueue> prevQueues) { this.prevQueues = prevQueues;
    public void addQueueBefore(MyQueue q) { this.prevQueues.add(q); }
    public int getServiceTime() { return serviceTime; }
    public void setServiceTime(int serviceTime) { this.serviceTime = serviceTime; }
    public boolean isRunning() { return running; }
    public void setRunning(boolean running) { this.running = running; }
```

```java
package com.example.producerconsumerbe.Service.Model;


import ...

public class Machine {

    private String id;
    private boolean running;
    private Product product;
    private Thread producerThread;
    private Thread consumerThread;
    private List<MyQueue> prevQueues;
    private MyQueue nextQueue;
    private int serviceTime;

    private final WebSocketService webSocketService;

    public Machine(WebSocketService webSocketService){
        this.webSocketService = webSocketService;
        this.id = UUID.randomUUID().toString();
        this.running = false;
        this.serviceTime = ThreadLocalRandom.current()
                .nextInt(SERVICE_TIME.MINIMUM.time, SERVICE_TIME.MAXIMUM.time);
        this.prevQueues = new ArrayList<>();

    }

    public String getId() { return id; }

    public void setId(String id) { this.id = id; }
```

2. **observer:** define one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

```java
package com.example.producerconsumerbe.Service.Model;

public interface Observer {
    void update();
}
```

```java
package com.example.producerconsumerbe.Service.Model;

import ...

public class MyQueue implements Observer {

    private String id;
    private final Queue<Product> queue;
    private List<Machine> forwardMachines;
    private int size;
    private boolean locker;

    public MyQueue() {
        this.forwardMachines = new ArrayList<>();
        queue = new LinkedList<>();
        size = 0;
        locker = false;
    }

    public boolean isLocker() { return locker; }

    public void toggleLocker() { this.locker = !locker; }

    public void put(Product product) {

        synchronized (this) {
            queue.add(product);
            this.notify();
            this.size = queue.size();
        }
    }
```

```java
package com.example.producerconsumerbe.Service.Model;

import ...

public class MyQueue implements Observer {

    private String id;
    private final Queue<Product> queue;
    private List<Machine> forwardMachines;
    private int size;
    private boolean locker;

    public MyQueue() {
        this.forwardMachines = new ArrayList<>();
        queue = new LinkedList<>();
        size = 0;
        locker = false;
    }

    public boolean isLocker() { return locker; }

    public void toggleLocker() { this.locker = !locker; }

    public void put(Product product) {

        synchronized (this) {
            queue.add(product);
            this.notify();
            this.size = queue.size();
        }
    }
```

3. **snapshot:** which is used to restore the previous state of the system.

```java
package com.example.producerconsumerbe.Service.Snapshot;

import ...

public class Originator {

    private List<Pair<Long,Product>> products;

    public Originator() {
        this.products = new ArrayList<>();
    }

    public List<Pair<Long,Product>> getProducts() {
        return products;
    }

    public void setProducts(List<Pair<Long,Product>> products) {
        this.products = products;
    }

    public Memento saveToMemento(){
        return new Memento(products);
    }

    public void getStateFromMemento(Memento memento){
        this.products = memento.getState();
    }
}
```

```java
package com.example.producerconsumerbe.Service.Snapshot;

import ...

public class Memento {

    private final List<Pair<Long,Product>> products;

    public Memento(List<Pair<Long,Product>> products){
        this.products = products;
    }

    public List<Pair<Long,Product>> getState(){
        return products;
    }
}
```

```java
package com.example.producerconsumerbe.Service.Snapshot;

import java.util.Stack;

public class CareTaker {

    private final Stack<Memento> mementoStack;

    public CareTaker() { mementoStack = new Stack<>(); }

    public void add(Memento memento) { mementoStack.push(memento); }

    public Memento getLast() { return mementoStack.pop(); }

    public int getSize() { return mementoStack.size(); }

    public void clear() { mementoStack.clear(); }

}
```

4. **prototype:** used to clone product as all products have the same properties but differ in their number only.

```java
package com.example.producerconsumerbe.Service.Model;

import ...

public class Product implements Cloneable{

    private final Color color;
    private final String id;

    public Product() {
        this.id = UUID.randomUUID().toString();
        this.color = RandomColorGenerator.generateColor();
    }

    public Color getColor() { return color; }
    public String getId() { return id; }


    @Override
    public Product clone() {
        try {
            return (Product) super.clone();
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }

    @Override
    public String toString(){
        ArrayList<Integer> rgb = new ArrayList<>();
        rgb.add(color.getRed());
```

How-To-Document:

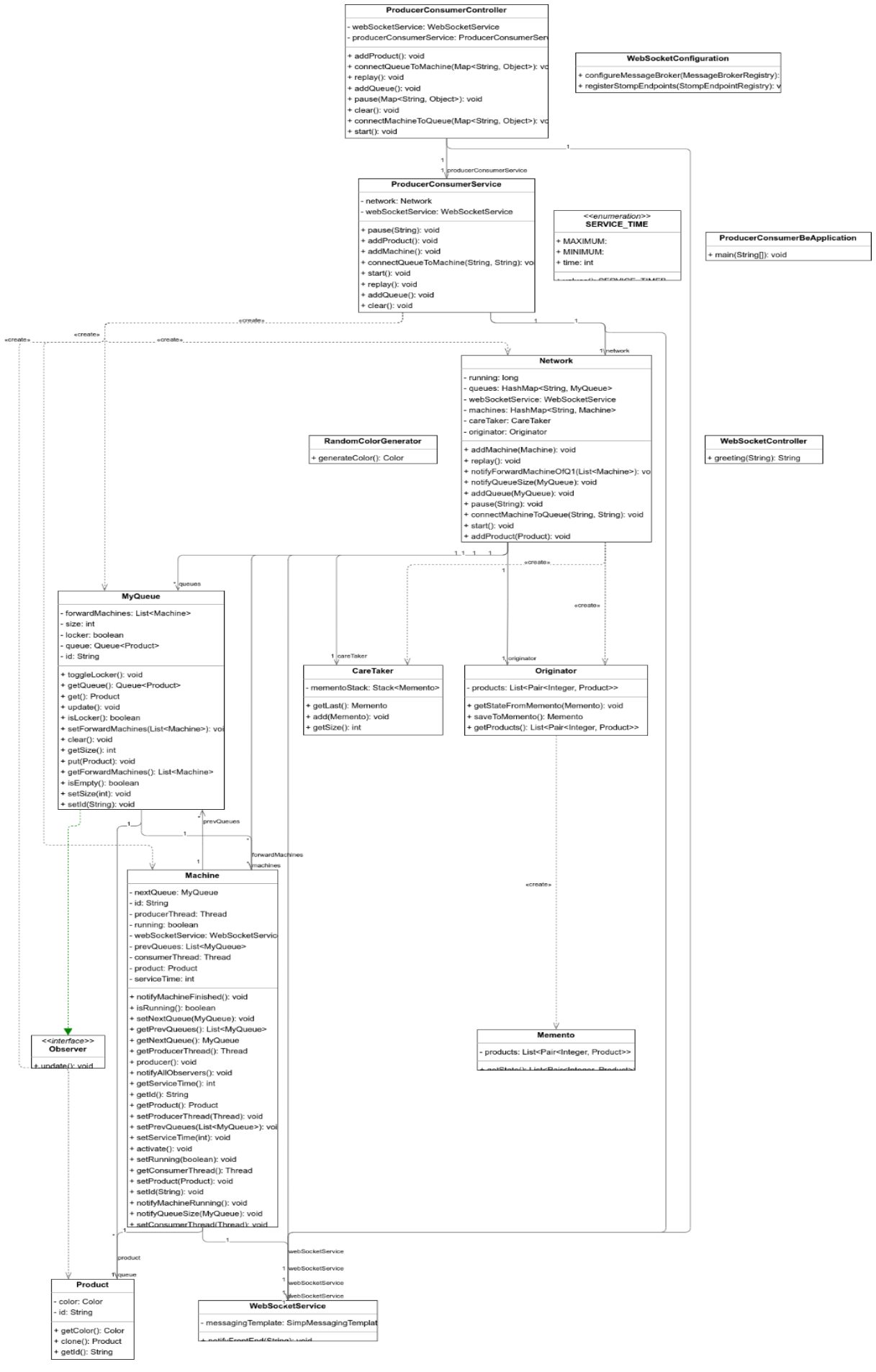
- download the zip file.
- open the back end project on any ide
- check the server port if it is available and if not you can change it in the application.resoucres file and type server.port = chosenPort.

- then you can run the back end project.
- open the front end project on any ide.
- first install the ngModules by typing npm install.
- then install knova library by typing

  npm install konva ng2-konva --save

- then type ng serve –open then the browser will open.

System design and UML:


ProducerConsumerController
- webSocketService: WebSocketService
- producerConsumerService: ProducerConsumerSer
+ addProduct(): void
+ connectQueueToMachine(Map<String, Object>): vo
+ replay(): void
+ addQueue(): void
+ pause(Map<String, Object>): void
+ clear(): void
+ connectMachineToQueue(Map<String, Object>): vo
+ start(): void
WebSocketConfiguration
+ configureMessageBroker(MessageBrokerRegistry):
+ registerStompEndpoints(StompEndpointRegistry): v
1
1 producerConsumerService
ProducerConsumerService
- network: Network
- webSocketService: WebSocketService
+ pause(String): void
+ addProduct(): void
+ addMachine(): void
+ connectQueueToMachine(String, String): vo
+ start(): void
+ replay(): void
+ addQueue(): void
+ clear(): void
<<enumeration>>
SERVICE_TIME
+ MAXIMUM:
+ MINIMUM:
+ time: int
ProducerConsumerBeApplication
+ main(String[]): void
«create»
«create»
«create»
«create»
1 network
Network
- running: long
- queues: HashMap<String, MyQueue>
- webSocketService: WebSocketService
- machines: HashMap<String, Machine>
- careTaker: CareTaker
- originator: Originator
+ addMachine(Machine): void
+ replay(): void
+ notifyForwardMachineOfQ1(List<Machine>): vo
+ notifyQueueSize(MyQueue): void
+ addQueue(MyQueue): void
+ pause(String): void
+ connectMachineToQueue(String, String): void
+ start(): void
+ addProduct(Product): void
RandomColorGenerator
+ generateColor(): Color
WebSocketController
+ greeting(String): String
* queues
«create»
«create»
MyQueue
- forwardMachines: List<Machine>
- size: int
- locker: boolean
- queue: Queue<Product>
- id: String
+ toggleLocker(): void
+ getQueue(): Queue<Product>
+ get(): Product
+ update(): void
+ isLocker(): boolean
+ setForwardMachines(List<Machine>): voi
+ clear(): void
+ getSize(): int
+ put(Product): void
+ getForwardMachines(): List<Machine>
+ isEmpty(): boolean
+ setSize(int): void
+ setId(String): void
1 careTaker
1 originator
CareTaker
- mementoStack: Stack<Memento>
+ getLast(): Memento
+ add(Memento): void
+ getSize(): int
Originator
- products: List<Pair<Integer, Product>>
+ getStateFromMemento(Memento): void
+ saveToMemento(): Memento
+ getProducts(): List<Pair<Integer, Product>>
prevQueues
forwardMachines
machines
«create»
Machine
- nextQueue: MyQueue
- id: String
- producerThread: Thread
- running: boolean
- webSocketService: WebSocketServic
- prevQueues: List<MyQueue>
- consumerThread: Thread
- product: Product
- serviceTime: int
+ notifyMachineFinished(): void
+ isRunning(): boolean
+ setNextQueue(MyQueue): void
+ getPrevQueues(): List<MyQueue>
+ getNextQueue(): MyQueue
+ getProducerThread(): Thread
+ producer(): void
+ notifyAllObservers(): void
+ getServiceTime(): int
+ getId(): String
+ getProduct(): Product
+ setProducerThread(Thread): void
+ setPrevQueues(List<MyQueue>): voi
+ setServiceTime(int): void
+ activate(): void
+ setRunning(boolean): void
+ getConsumerThread(): Thread
+ setProduct(Product): void
+ setId(String): void
+ notifyMachineRunning(): void
+ notifyQueueSize(MyQueue): void
+ setConsumerThread(Thread): void
<<interface>>
Observer
+ update(): void
Memento
- products: List<Pair<Integer, Product>>
product
queue
webSocketService
webSocketService
webSocketService
webSocketService
Product
- color: Color
- id: String
+ getColor(): Color
+ clone(): Product
+ getId(): String
WebSocketService
- messagingTemplate: SimpMessagingTemplat
+ notifyFrontEnd(String): void