Teammates Names

Omar Tammam - 20011016

Joseph Shokry Soliman - 20010439

Osama Elsayed Belal - 20010269

Marwan Essam eldien Rashad - 20011859

1.1 GitHub Repositories

- **Back-End Repository**

On main branch :

https://github.com/marwanesam22/Producer-Consumer

Front-End Repository

On main branch :

https://github.com/OmarTammam25/Producer-Consumer

1.2 Instruction to download codes

- **Downloading codes from GitHub repositories**

1. Open your Git Bash terminal.
2. Cloning Back-End files to your folder.
3. Cloning Front-End files to your folder.

1.3 Instruction to Run Back-End Server

- **Running Back-End codes**

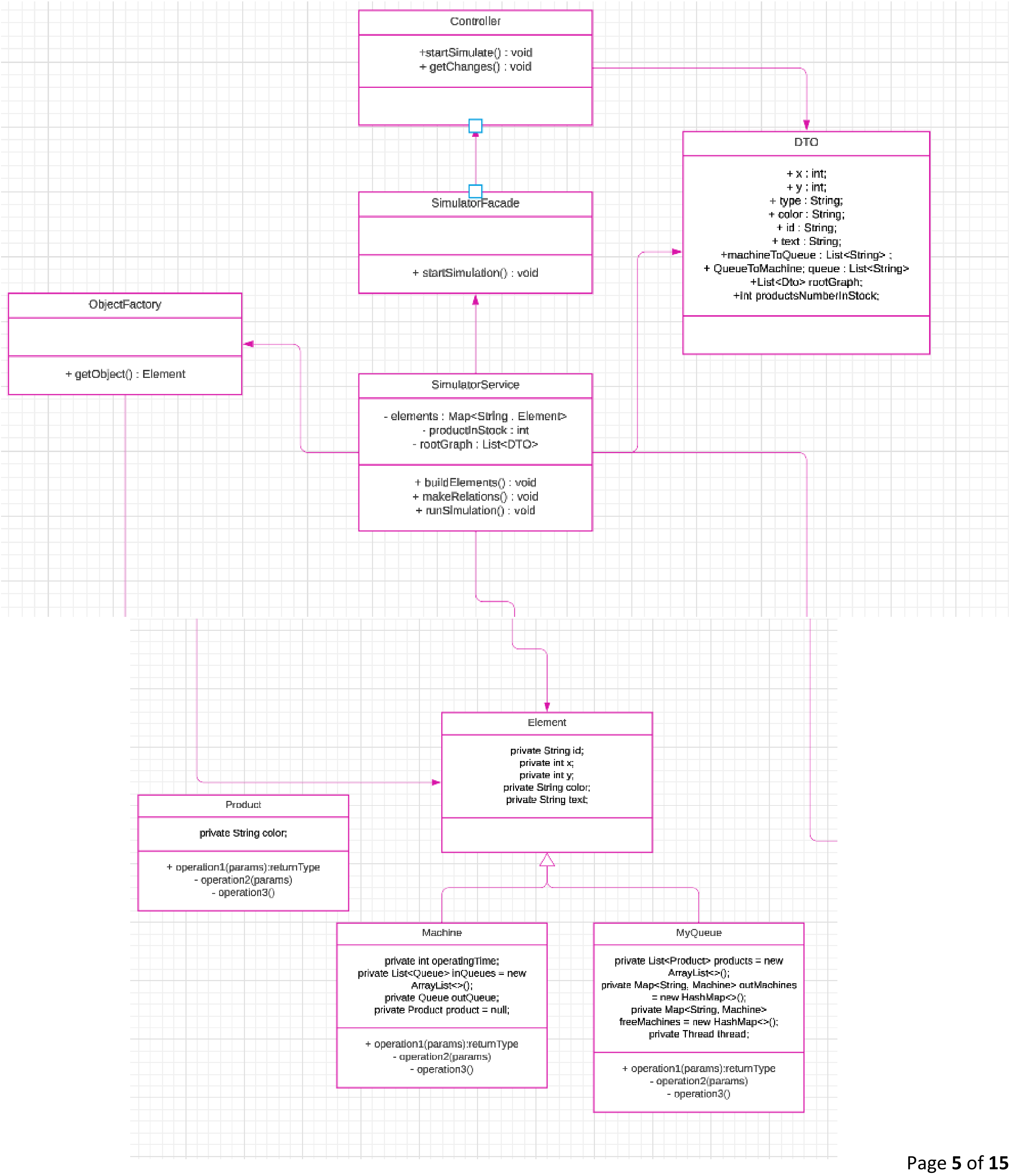
1. Open Back-End files to your favorite IDE to be run.
2. Use the normal Run button in your IDE.

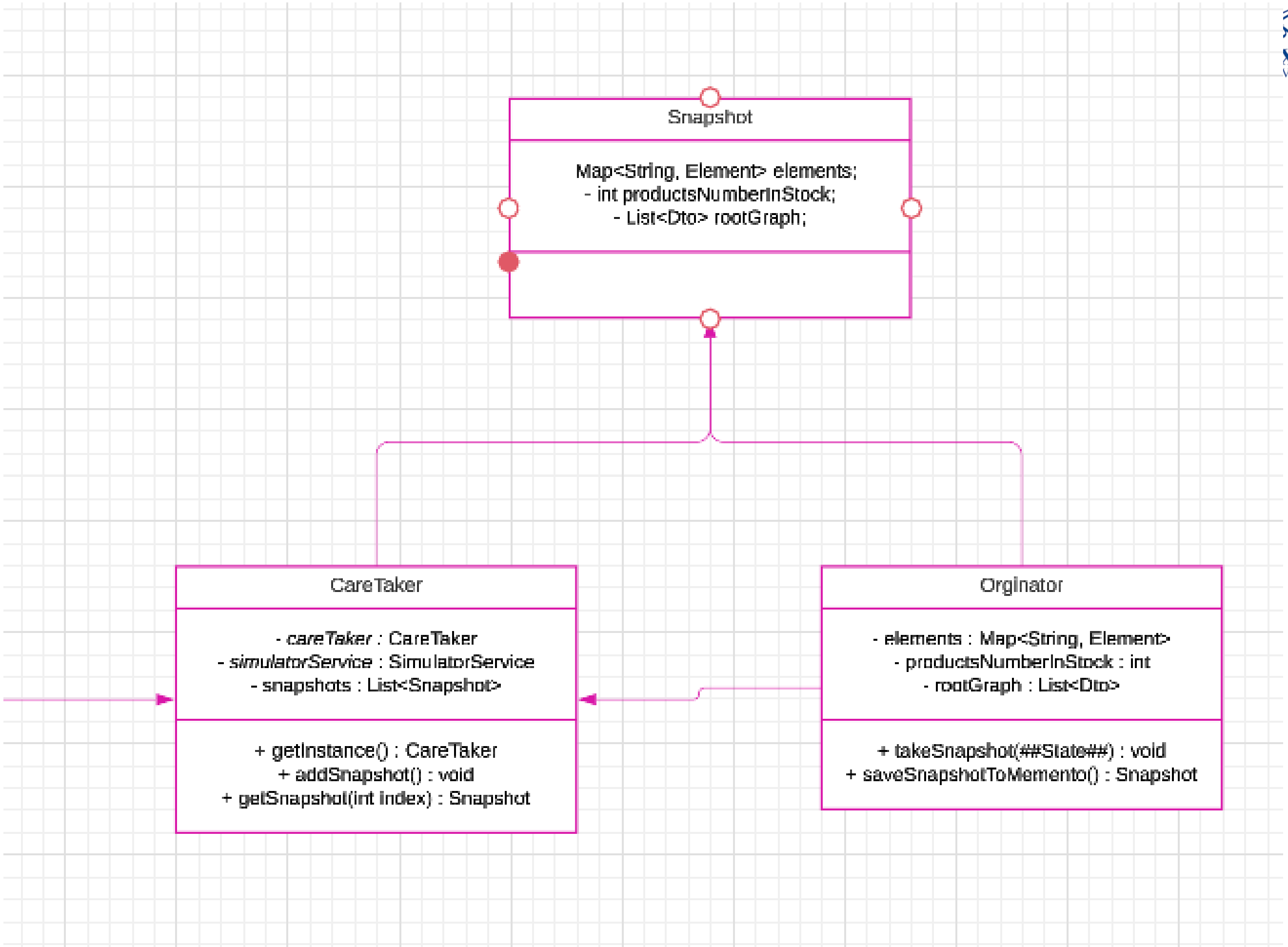
1.4 Instruction to Run Front-End Server

- **Running Front-End codes**

  1. Install Node.js from the official website
  2. Open your Command Prompt.
  3. In your Command Prompt "npm install -g @angular/cli" .
  4. Open Front-End files to your favorite IDE to be run.
  5. Running Angular server from prompt "ng serve --open" .

2.1 UML snippets




Snapshot
Map<String, Element> elements;
- int productsNumberInStock;
- List<Dto> rootGraph;
CareTaker
- *careTaker* : CareTaker
- *simulatorService* : SimulatorService
- snapshots : List<Snapshot>
+ getInstance() : CareTaker
+ addSnapshot() : void
+ getSnapshot(int index) : Snapshot
Orginator
- elements : Map<String, Element>
- productsNumberInStock : int
- rootGraph : List<Dto>
+ takeSnapshot(##State##) : void
+ saveSnapshotToMemento() : Snapshot

3.1 Observer Design Pattern

We used observer design pattern between Machines(Observable) and queues (Observers) as when any machine is ready it notifies its subscribers that “I am ready to get products “and when it is busy it notifies its subscribers “I am busy now ” not to send products.

3.2 Snapshot Design Pattern

We used Snapshot Design pattern to

1) save states of products, machines and queues to send snaps to view to run the simulation of the products.
2) Restimulate the process happened as we store the snaps then run it sequentially with 200ms delay to see the whole replayed process and according to optimization of time taken back-end server doesn't send snaps again to view but view store them to run them again when replay simulation action is fired.

3.3 Concurrency

Producer-Consumer :

We used producer-consumer concurrency design pattern to to synchronize functionalities of queues and machines.

- Products are moving to machines through queue to wait till machine permit them to be processed.
- Machines takes its product from queue when it is free.

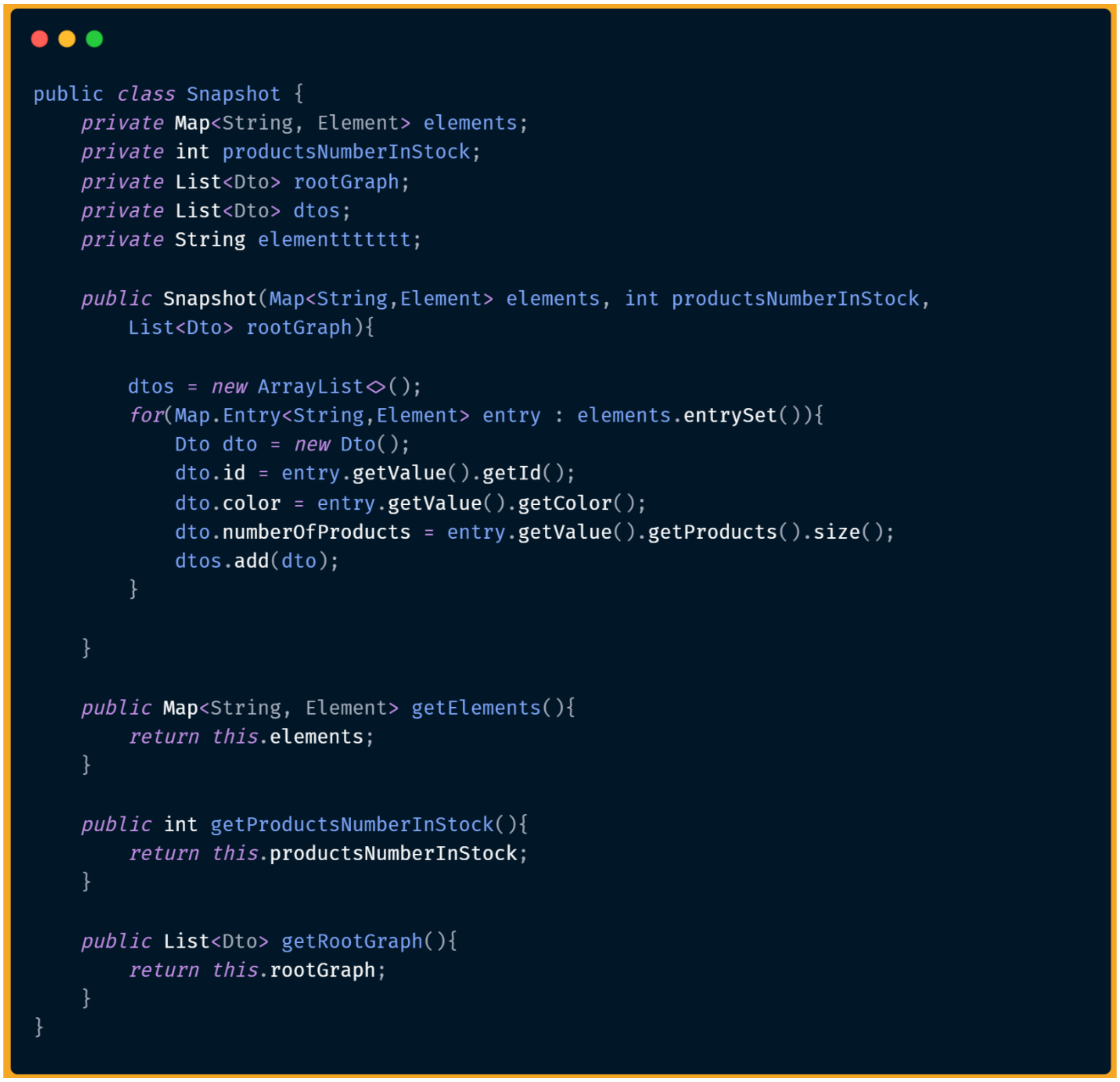
3.4 Façade design pattern

We used façade design pattern to encapsulate the implementation of the simulator service from the controller to facilitate the communication between the controller and the service.

3.5 Code Snippets

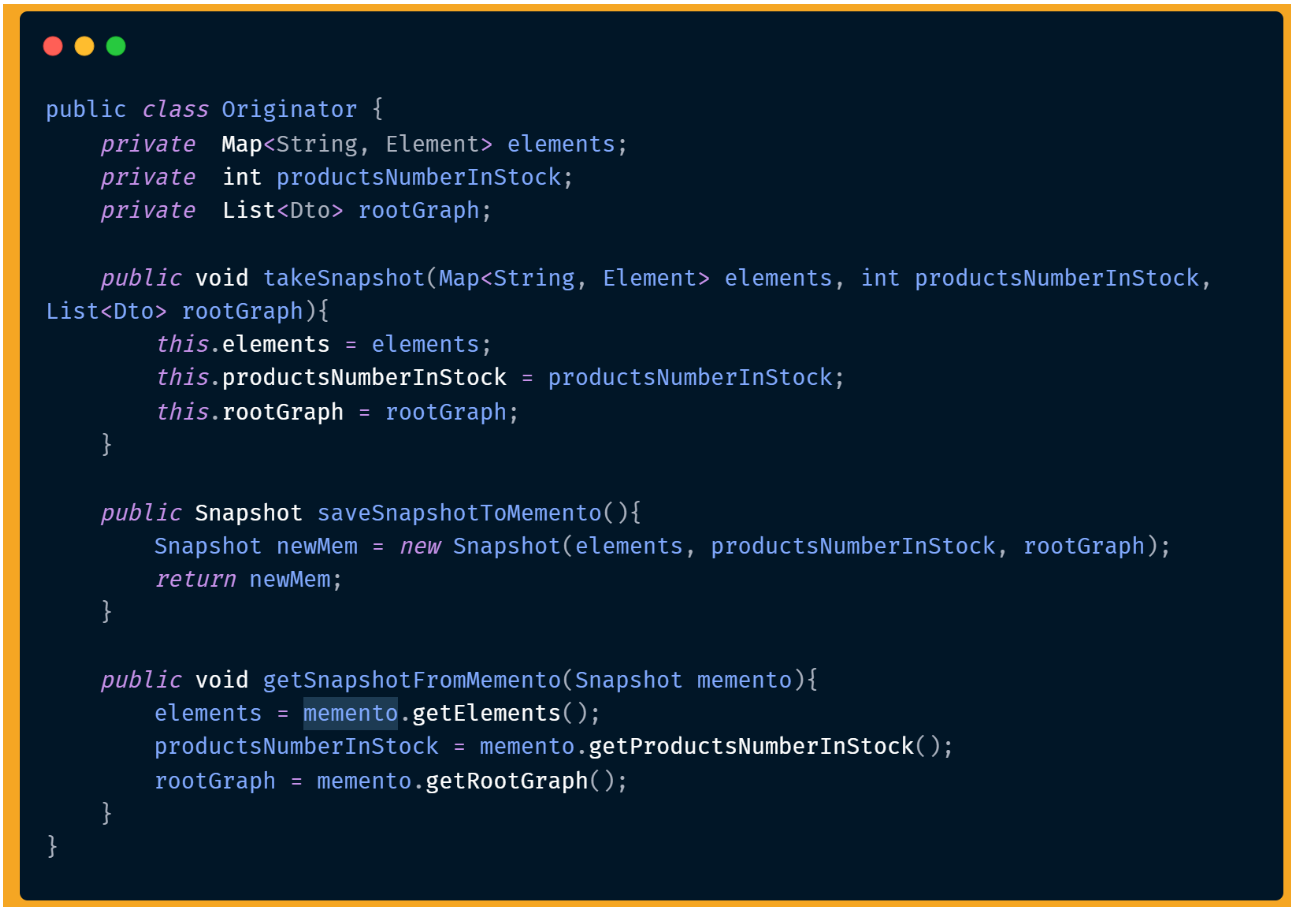
1) Snapshot design pattern code snippets

Snapshot Class (Memento)



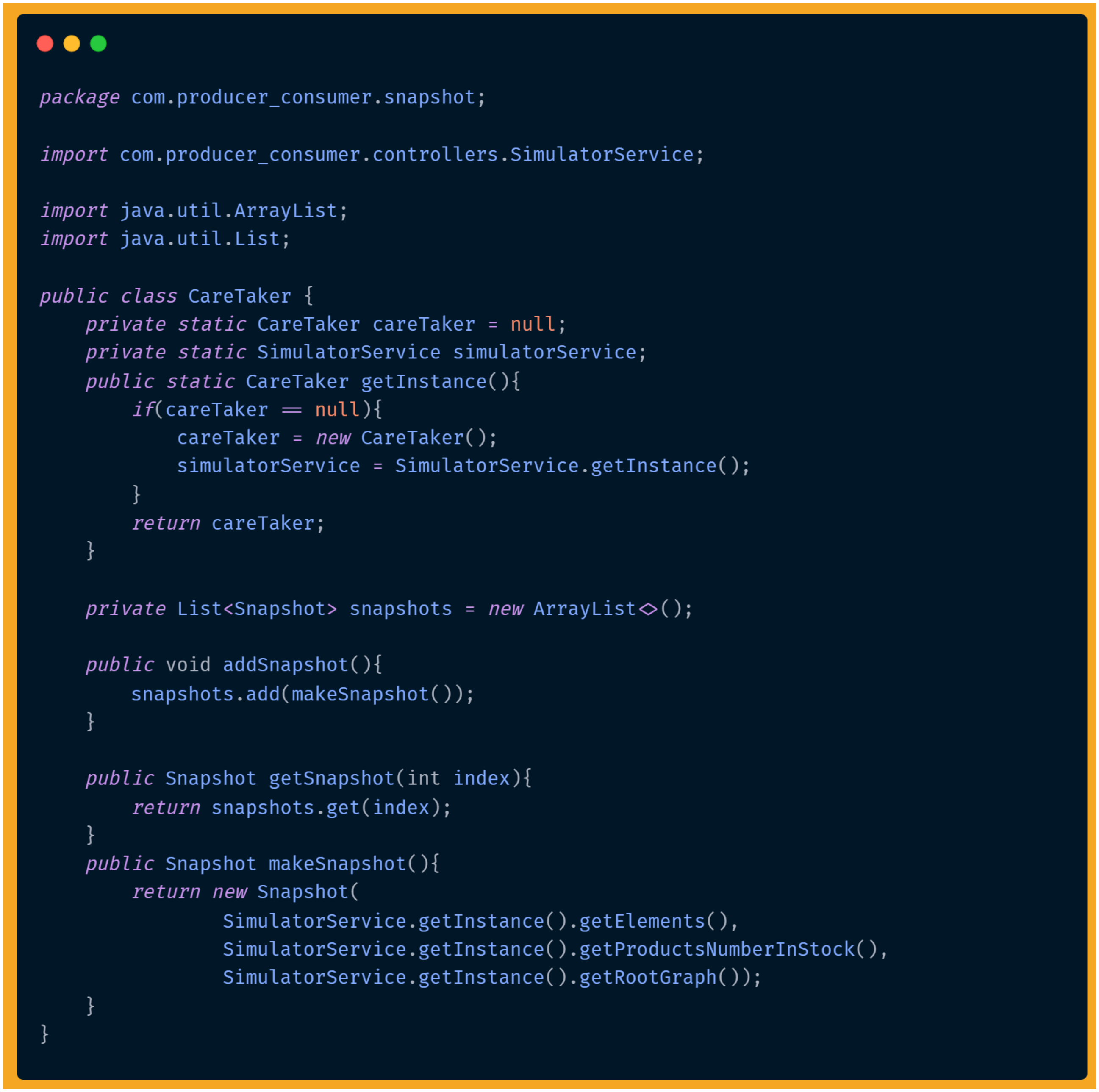
```java
public class Snapshot {
    private Map<String, Element> elements;
    private int productsNumberInStock;
    private List<Dto> rootGraph;
    private List<Dto> dtos;
    private String elementtttttt;

    public Snapshot(Map<String,Element> elements, int productsNumberInStock,
        List<Dto> rootGraph){

        dtos = new ArrayList<>();
        for(Map.Entry<String,Element> entry : elements.entrySet()){
            Dto dto = new Dto();
            dto.id = entry.getValue().getId();
            dto.color = entry.getValue().getColor();
            dto.numberOfProducts = entry.getValue().getProducts().size();
            dtos.add(dto);
        }

    }

    public Map<String, Element> getElements(){
        return this.elements;
    }

    public int getProductsNumberInStock(){
        return this.productsNumberInStock;
    }

    public List<Dto> getRootGraph(){
        return this.rootGraph;
    }
}
```

Originator Class



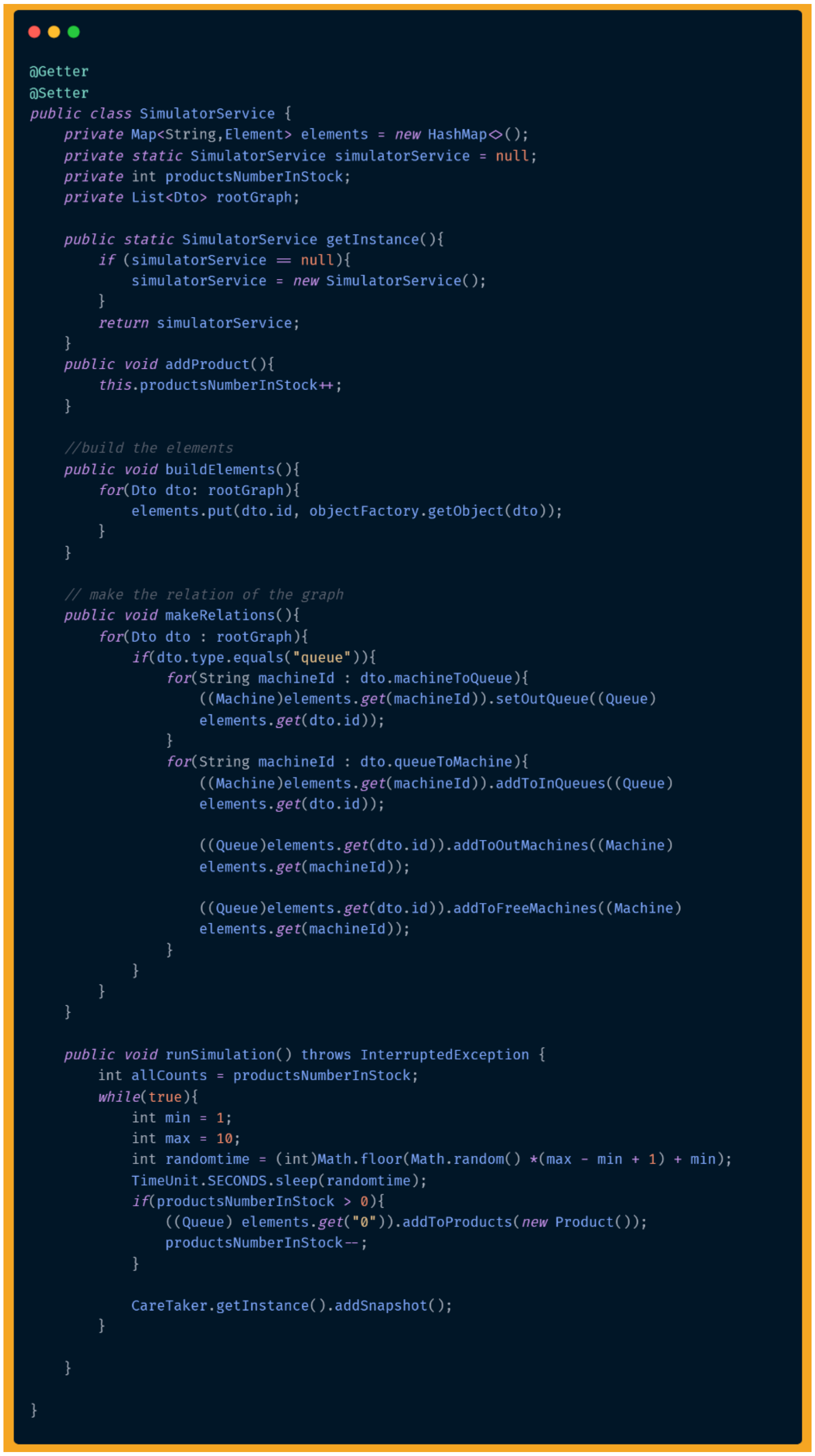
```
public class Originator {
    private  Map<String, Element> elements;
    private  int productsNumberInStock;
    private  List<Dto> rootGraph;

    public void takeSnapshot(Map<String, Element> elements, int productsNumberInStock,
List<Dto> rootGraph){
        this.elements = elements;
        this.productsNumberInStock = productsNumberInStock;
        this.rootGraph = rootGraph;
    }

    public Snapshot saveSnapshotToMemento(){
        Snapshot newMem = new Snapshot(elements, productsNumberInStock, rootGraph);
        return newMem;
    }

    public void getSnapshotFromMemento(Snapshot memento){
        elements = memento.getElements();
        productsNumberInStock = memento.getProductsNumberInStock();
        rootGraph = memento.getRootGraph();
    }
}
```

CareTaker Class



```java
package com.producer_consumer.snapshot;

import com.producer_consumer.controllers.SimulatorService;

import java.util.ArrayList;
import java.util.List;

public class CareTaker {
    private static CareTaker careTaker = null;
    private static SimulatorService simulatorService;
    public static CareTaker getInstance(){
        if(careTaker == null){
            careTaker = new CareTaker();
            simulatorService = SimulatorService.getInstance();
        }
        return careTaker;
    }

    private List<Snapshot> snapshots = new ArrayList<>();

    public void addSnapshot(){
        snapshots.add(makeSnapshot());
    }

    public Snapshot getSnapshot(int index){
        return snapshots.get(index);
    }
    public Snapshot makeSnapshot(){
        return new Snapshot(
                SimulatorService.getInstance().getElements(),
                SimulatorService.getInstance().getProductsNumberInStock(),
                SimulatorService.getInstance().getRootGraph());
    }
}
```

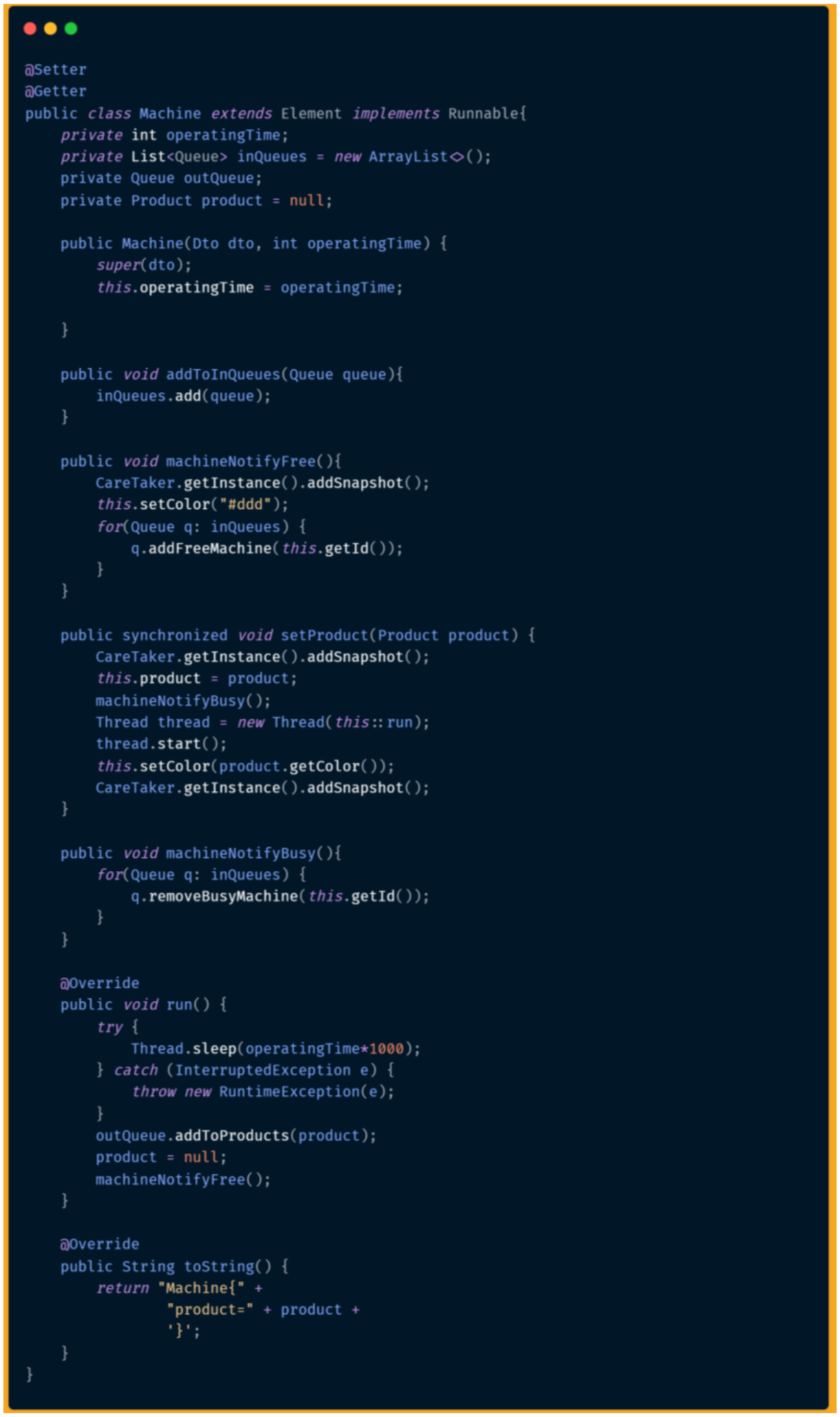
2) Producer – Consumer design pattern



```
@Getter
@Setter
public class SimulatorService {
    private Map<String,Element> elements = new HashMap<>();
    private static SimulatorService simulatorService = null;
    private int productsNumberInStock;
    private List<Dto> rootGraph;

    public static SimulatorService getInstance(){
        if (simulatorService == null){
            simulatorService = new SimulatorService();
        }
        return simulatorService;
    }
    public void addProduct(){
        this.productsNumberInStock++;
    }

    //build the elements
    public void buildElements(){
        for(Dto dto: rootGraph){
            elements.put(dto.id, objectFactory.getObject(dto));
        }
    }

    // make the relation of the graph
    public void makeRelations(){
        for(Dto dto : rootGraph){
            if(dto.type.equals("queue")){
                for(String machineId : dto.machineToQueue){
                    ((Machine)elements.get(machineId)).setOutQueue((Queue)
                    elements.get(dto.id));
                }
                for(String machineId : dto.queueToMachine){
                    ((Machine)elements.get(machineId)).addToInQueues((Queue)
                    elements.get(dto.id));

                    ((Queue)elements.get(dto.id)).addToOutMachines((Machine)
                    elements.get(machineId));

                    ((Queue)elements.get(dto.id)).addToFreeMachines((Machine)
                    elements.get(machineId));
                }
            }
        }
    }

    public void runSimulation() throws InterruptedException {
        int allCounts = productsNumberInStock;
        while(true){
            int min = 1;
            int max = 10;
            int randomtime = (int)Math.floor(Math.random() *(max - min + 1) + min);
            TimeUnit.SECONDS.sleep(randomtime);
            if(productsNumberInStock > 0){
                ((Queue) elements.get("0")).addToProducts(new Product());
                productsNumberInStock--;
            }

            CareTaker.getInstance().addSnapshot();
        }

    }

}
```

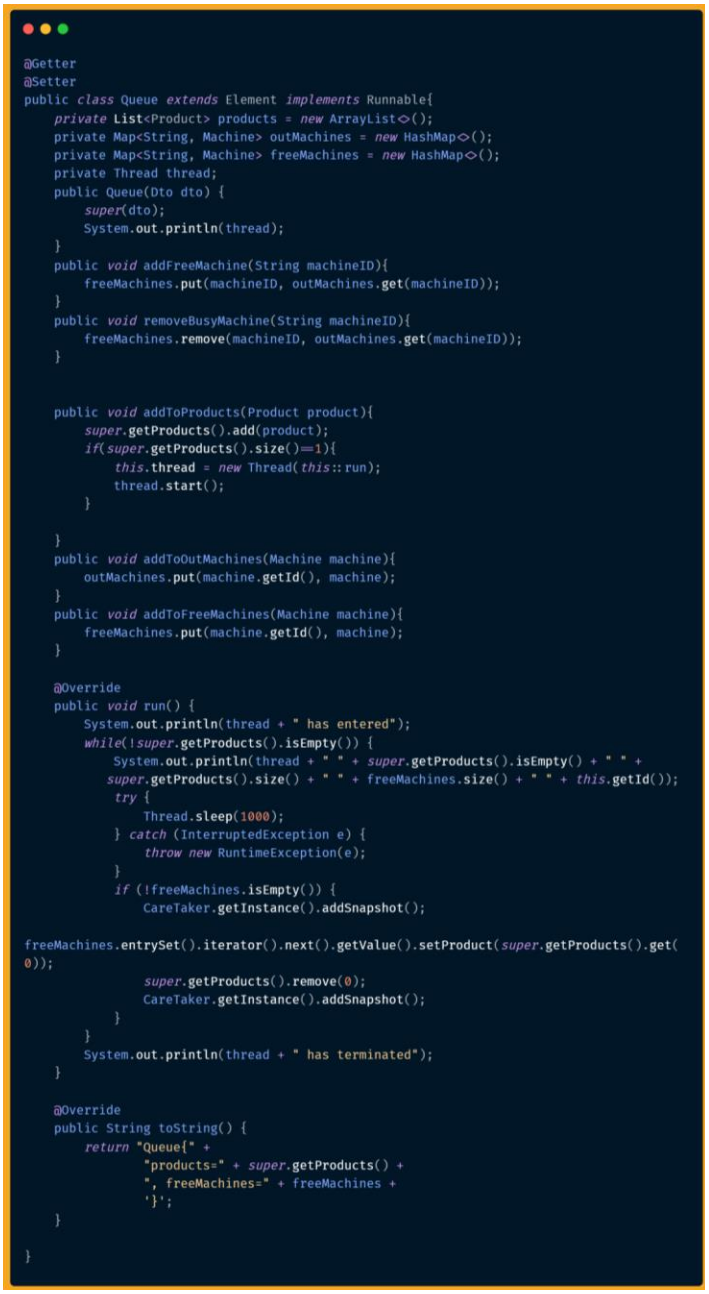
3) Observer design pattern

Machine (Observable)



```java
@Setter
@Getter
public class Machine extends Element implements Runnable{
    private int operatingTime;
    private List<Queue> inQueues = new ArrayList<>();
    private Queue outQueue;
    private Product product = null;

    public Machine(Dto dto, int operatingTime) {
        super(dto);
        this.operatingTime = operatingTime;

    }

    public void addToInQueues(Queue queue){
        inQueues.add(queue);
    }

    public void machineNotifyFree(){
        CareTaker.getInstance().addSnapshot();
        this.setColor("#ddd");
        for(Queue q: inQueues) {
            q.addFreeMachine(this.getId());
        }
    }

    public synchronized void setProduct(Product product) {
        CareTaker.getInstance().addSnapshot();
        this.product = product;
        machineNotifyBusy();
        Thread thread = new Thread(this::run);
        thread.start();
        this.setColor(product.getColor());
        CareTaker.getInstance().addSnapshot();
    }

    public void machineNotifyBusy(){
        for(Queue q: inQueues) {
            q.removeBusyMachine(this.getId());
        }
    }

    @Override
    public void run() {
        try {
            Thread.sleep(operatingTime*1000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        outQueue.addToProducts(product);
        product = null;
        machineNotifyFree();
    }

    @Override
    public String toString() {
        return "Machine{" +
                "product=" + product +
                '}';
    }
}
```

Queues (Observers)

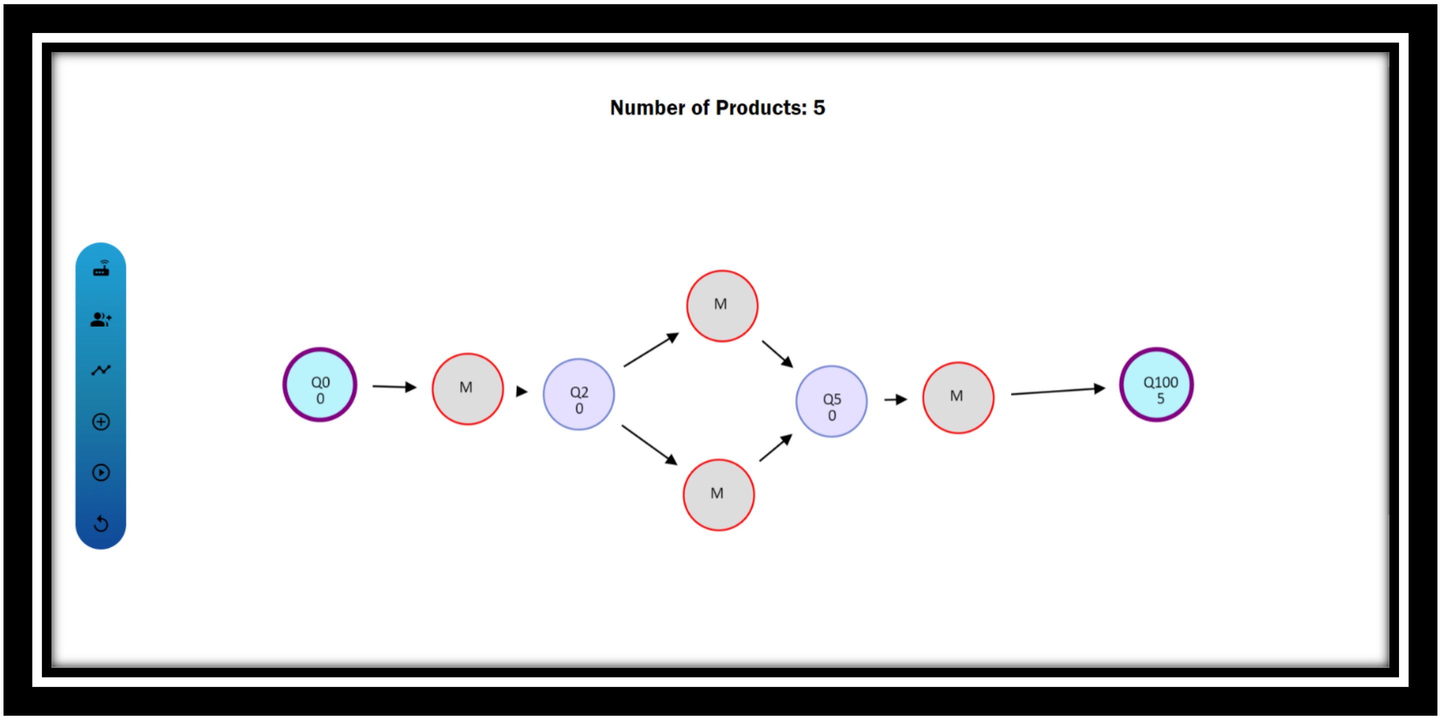


```
@Getter
@Setter
public class Queue extends Element implements Runnable{
    private List<Product> products = new ArrayList<>();
    private Map<String, Machine> outMachines = new HashMap<>();
    private Map<String, Machine> freeMachines = new HashMap<>();
    private Thread thread;
    public Queue(Dto dto) {
        super(dto);
        System.out.println(thread);
    }
    public void addFreeMachine(String machineID){
        freeMachines.put(machineID, outMachines.get(machineID));
    }
    public void removeBusyMachine(String machineID){
        freeMachines.remove(machineID, outMachines.get(machineID));
    }


    public void addToProducts(Product product){
        super.getProducts().add(product);
        if(super.getProducts().size()==1){
            this.thread = new Thread(this::run);
            thread.start();
        }

    }
    public void addToOutMachines(Machine machine){
        outMachines.put(machine.getId(), machine);
    }
    public void addToFreeMachines(Machine machine){
        freeMachines.put(machine.getId(), machine);
    }

    @Override
    public void run() {
        System.out.println(thread + " has entered");
        while(!super.getProducts().isEmpty()) {
            System.out.println(thread + " " + super.getProducts().isEmpty() + " " +
           super.getProducts().size() + " " + freeMachines.size() + " " + this.getId());
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            if (!freeMachines.isEmpty()) {
                CareTaker.getInstance().addSnapshot();

freeMachines.entrySet().iterator().next().getValue().setProduct(super.getProducts().get(
0));
                super.getProducts().remove(0);
                CareTaker.getInstance().addSnapshot();
            }
        }
        System.out.println(thread + " has terminated");
    }

    @Override
    public String toString() {
        return "Queue{" +
                "products=" + super.getProducts() +
                ", freeMachines=" + freeMachines +
                '}';
    }

}
```

4.1 Decisions

- Used Konva library for drawing.
- Operating time in machine [4 - 10] seconds.
- Front sever sends request each 200ms.

5.1 UI snippets


Number of Products: 5

5.2 User Guide

User can

1) Create product on runtime by pressing on the product icon button.
2) Add Queues by pressing on the queue icon.
3) Create machine by pressing on the machine icon button
4) Restimulate the process by pressing on replay icon button
5) Start simulation by pressing on the simulate button.
6) Connect the machine and queues by pressing on the line icon button.