**Group members**

| | |
|---|---|
| • Mina Nabil | 22011290 |
| • Abdelrahman Khaled | 22010877 |
| • Omar Khaled | 22010962 |
| • Yaseen Asaad | 22011349 |

## Table of Contents:

---

## Introduction:

- **Project Overview:**

This project aims to develop an object-oriented queuing simulation program that models an assembly line producing various products. The assembly line consists of multiple processing machines (Ms) and queues (Qs) that manage the movement of products between different processing stages. The simulation will be implemented using **Java Spring Boot** and **React**, incorporating several design patterns such as **concurrency, snapshot, and observer** patterns to ensure efficient and effective simulation.

- **Key Features**

✓ **Random Input and Service Times**:
  - Products arrive at the initial queue (Q0) at a random rate.
  - Each machine processes products with a random service time and serves one product at a time.

✓ **Concurrency**:
  - Each machine operates on a separate thread, processing products independently.

✓ **Undo and Redo**:
  - Users can undo and redo actions within the simulation, allowing for flexible control and correction of operations.

✓ **Save and Load**:
  - Users can save the current state of the simulation and load it later, enabling them to pause and resume the simulation as needed.

✓ **Visual Indicators**:
  - Machines flash when they finish processing an item.
  - Products have unique colors that change the machine's color during processing, making it easy for users to follow the simulation.

✓ **Simulation Control**:

- Users can start a new simulation or replay the previous one using the snapshot pattern.

## User Guide:

### Project GitHub Repository:

**https://github.com/omar2004khaled/Producer-Consumer-Simulation**

### Instructions to Download and Run the Project

1. **Cloning the Project from GitHub:**
   - Open your Git Bash terminal.
   - Run the following command to clone the project:

     ```
     git clone https://github.com/omar2004khaled/Producer-Consumer-Simulation.git
     ```

2. **Running the Back-End Server:**
   - Open the **Back-End** project folder in your preferred IDE (e.g., Visual Studio Code, IntelliJ).
   - Run the project using the **Run** button or the terminal in your IDE.
3. **Running the Front-End Server:**
   - **Install Node.js**: Download and install Node.js from the official website.
   - **Install React Dependencies**:
     - Open your Command Prompt (or terminal) and navigate to the **Front-End** project folder.
     - Run the following command to install necessary dependencies:

       ```
       npm install
       ```

   - **Run the Front-End Server:**
     - In the terminal, run:

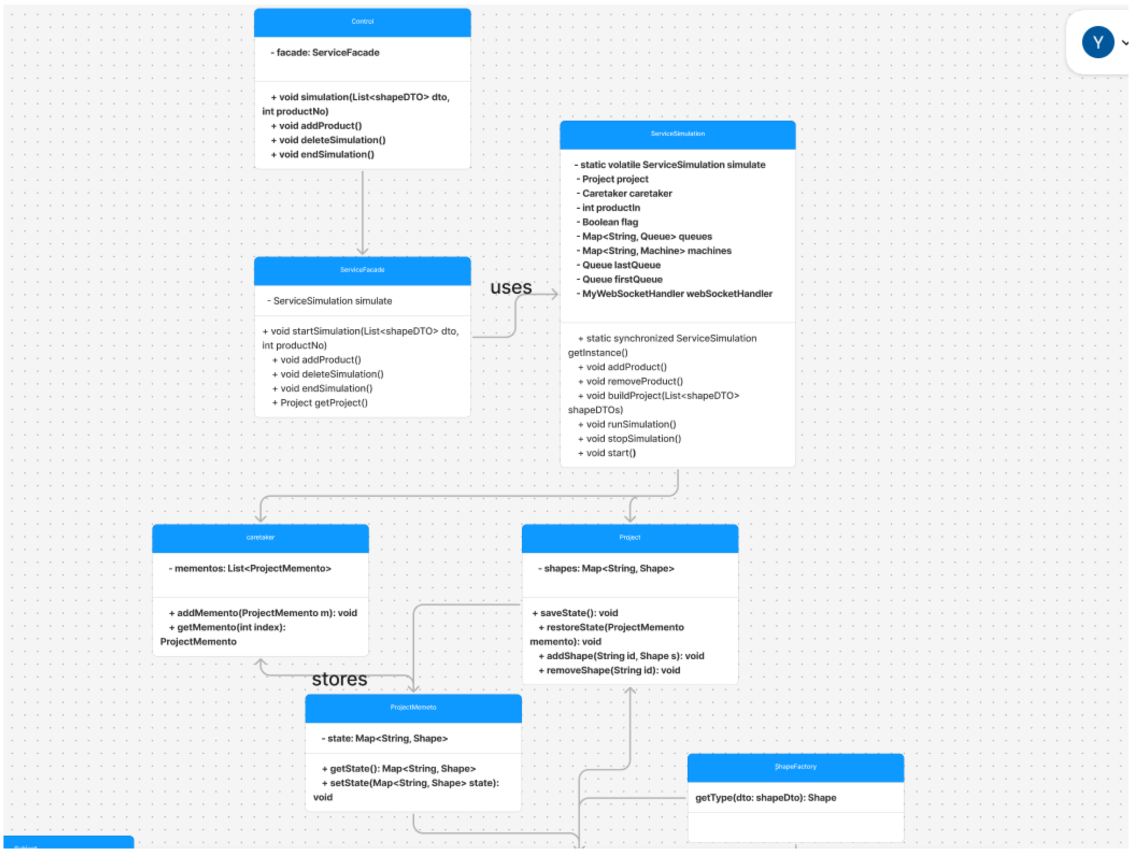
       ```
       npm run dev
       ```

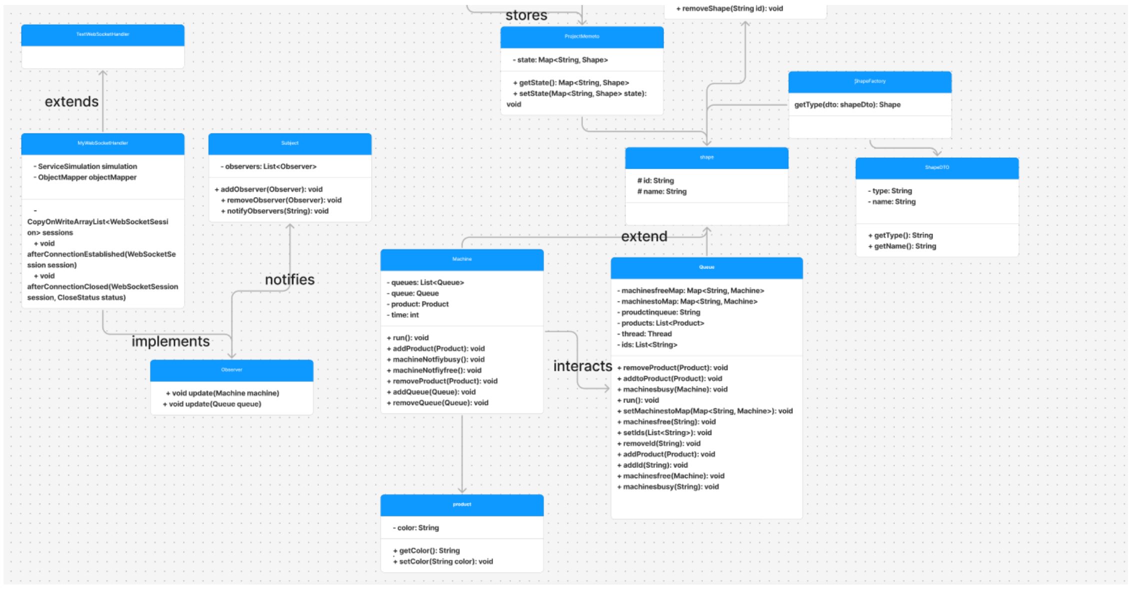
     - This will start the React development server, and the project should be running locally.
4. **Important Installations for the FrontEnd:**

   - npm install react-icons
   - npm install @xyflow/react
   - npm install @mui/icons-material @mui/material

Once these steps are completed, both the **Back-End** and **Front-End** servers will be running, Enjoy

# UML Diagram:


Control
- facade: ServiceFacade
+ void simulation(List<shapeDTO> dto, int productNo)
+ void addProduct()
+ void deleteSimulation()
+ void endSimulation()
ServiceFacade
- ServiceSimulation simulate
+ void startSimulation(List<shapeDTO> dto, int productNo)
+ void addProduct()
+ void deleteSimulation()
+ void endSimulation()
+ Project getProject()
uses
ServiceSimulation
- static volatile ServiceSimulation simulate
- Project project
- Caretaker caretaker
- int productIn
- Boolean flag
- Map<String, Queue> queues
- Map<String, Machine> machines
- Queue lastQueue
- Queue firstQueue
- MyWebSocketHandler webSocketHandler
+ static synchronized ServiceSimulation getInstance()
+ void addProduct()
+ void removeProduct()
+ void buildProject(List<shapeDTO> shapeDTOs)
+ void runSimulation()
+ void stopSimulation()
+ void start()
caretaker
- mementos: List<ProjectMemento>
+ addMemento(ProjectMemento m): void
+ getMemento(int index): ProjectMemento
Project
- shapes: Map<String, Shape>
+ saveState(): void
+ restoreState(ProjectMemento memento): void
+ addShape(String id, Shape s): void
+ removeShape(String id): void
stores
ProjectMemeto
- state: Map<String, Shape>
+ getState(): Map<String, Shape>
+ setState(Map<String, Shape> state): void
ShapeFactory
getType(dto: shapeDto): Shape


stores
+ removeShape(String id): void
TextWebSocketHandler
extends
MyWebSocketHandler
- ServiceSimulation simulation
- ObjectMapper objectMapper
- CopyOnWriteArrayList<WebSocketSession> sessions
+ void afterConnectionEstablished(WebSocketSession session)
+ void afterConnectionClosed(WebSocketSession session, CloseStatus status)
Subject
- observers: List<Observer>
+ addObserver(Observer): void
+ removeObserver(Observer): void
+ notifyObservers(String): void
notifies
implements
Observer
+ void update(Machine machine)
+ void update(Queue queue)
ProjectMemeto
- state: Map<String, Shape>
+ getState(): Map<String, Shape>
+ setState(Map<String, Shape> state): void
ShapeFactory
getType(dto: shapeDto): Shape
shape
# id: String
# name: String
ShapeDTO
- type: String
- name: String
+ getType(): String
+ getName(): String
extend
Machine
- queues: List<Queue>
- queue: Queue
- product: Product
- time: int
+ run(): void
+ addProduct(Product): void
+ machineNotfiybusy(): void
+ machineNotfiyfree(): void
+ removeProduct(Product): void
+ addQueue(Queue): void
+ removeQueue(Queue): void
interacts
Queue
- machinesfreeMap: Map<String, Machine>
- machinestoMap: Map<String, Machine>
- proudctinqueue: String
- products: List<Product>
- thread: Thread
- ids: List<String>
+ removeProduct(Product): void
+ addtoProduct(Product): void
+ machinesbusy(Machine): void
+ run(): void
+ setMachinestoMap(Map<String, Machine>): void
+ machinesfree(String): void
+ setIds(List<String>): void
+ removeId(String): void
+ addProduct(Product): void
+ addId(String): void
+ machinesfree(Machine): void
+ machinesbusy(String): void
product
- color: String
+ getColor(): String
+ setColor(String color): void

## Full UML diagram:

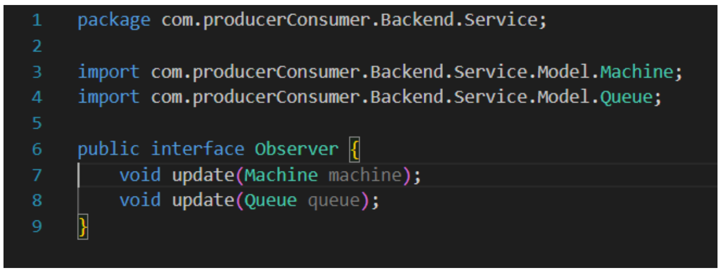
https://www.figma.com/board/CIyYvt7PkoY4VOvDyXGKSu/Producer-Consumer-Simulation?node-id=0-1&t=iphJ4nWdwMj6tgFy-1

# Design patterns:

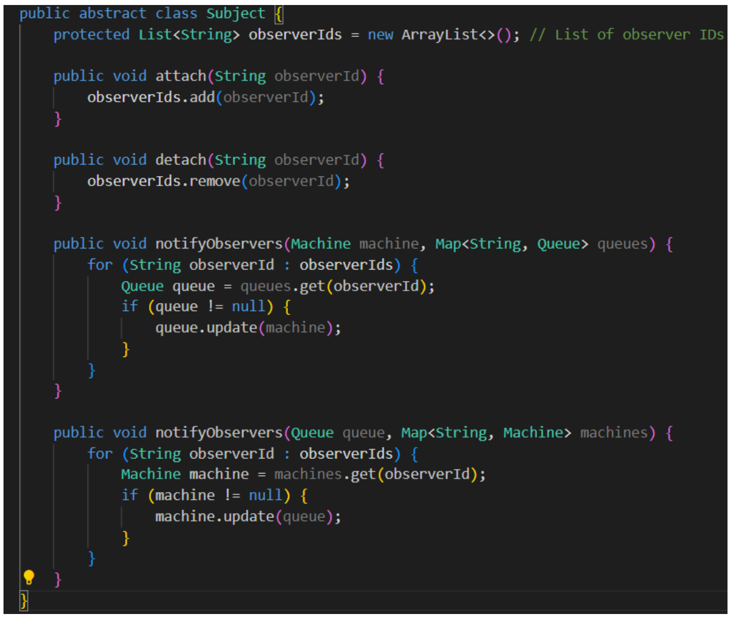
## 1. Observer Pattern:

- **Description:** The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. This pattern is useful for implementing distributed event-handling systems.
- **Implementation in Code:**
  - **Classes Involved:** Machine, Queue, Subject, Observer
  - **Methods:** notifyObservers(), addObserver(), removeObserver(), update()
  - **Usage:**
    - The Subject class maintains a list of observers and provides methods to add, remove, and notify observers.
    - The Machine and Queue classes implement the Observer interface and override the update() method to handle updates.
    - When a machine finishes processing a product, it notifies the queues that it is free using the notifyObservers() method. The queues then check if there are any products waiting to be processed and assign them to the free machine.

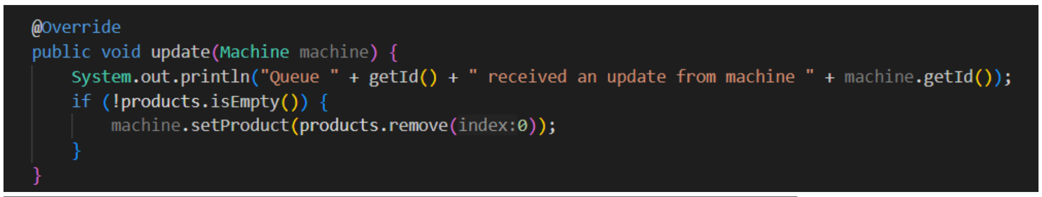
- Code



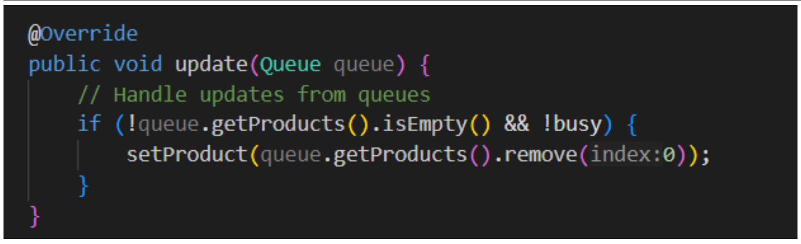
```java
package com.producerConsumer.Backend.Service;

import com.producerConsumer.Backend.Service.Model.Machine;
import com.producerConsumer.Backend.Service.Model.Queue;

public interface Observer {
    void update(Machine machine);
    void update(Queue queue);
}
```



```java
public abstract class Subject {
    protected List<String> observerIds = new ArrayList<>(); // List of observer IDs

    public void attach(String observerId) {
        observerIds.add(observerId);
    }

    public void detach(String observerId) {
        observerIds.remove(observerId);
    }

    public void notifyObservers(Machine machine, Map<String, Queue> queues) {
        for (String observerId : observerIds) {
            Queue queue = queues.get(observerId);
            if (queue != null) {
                queue.update(machine);
            }
        }
    }

    public void notifyObservers(Queue queue, Map<String, Machine> machines) {
        for (String observerId : observerIds) {
            Machine machine = machines.get(observerId);
            if (machine != null) {
                machine.update(queue);
            }
        }
    }
}
```



```java
@Override
public void update(Machine machine) {
    System.out.println("Queue " + getId() + " received an update from machine " + machine.getId());
    if (!products.isEmpty()) {
        machine.setProduct(products.remove(index:0));
    }
}
```



```java
@Override
public void update(Queue queue) {
    // Handle updates from queues
    if (!queue.getProducts().isEmpty() && !busy) {
        setProduct(queue.getProducts().remove(index:0));
    }
}
```

---

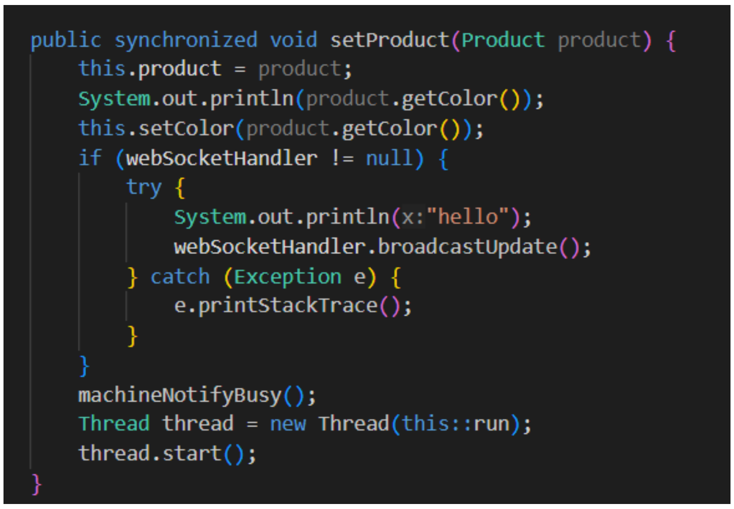
## 2. <u>**Concurrency Pattern**:</u>

### Producer-Consumer Pattern:

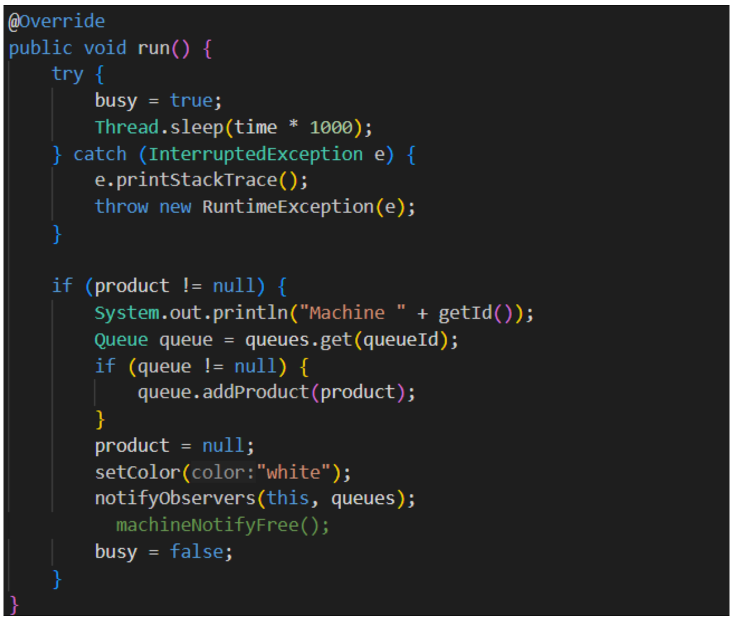
**Description:** The producer-consumer pattern is a concurrency pattern where producer threads generate data and place it into a buffer, and consumer threads take data from the buffer and process it. This pattern helps in coordinating the work between multiple threads.

- **Implementation in Code:**
  - **Classes Involved:** Machine, Queue
  - **Methods:** run(), setProduct(), addtoProduct()
  - **Usage:**
    - The Queue class acts as the buffer, holding products that need to be processed.
    - The Machine class acts as the consumer, taking products from the queue and processing them.
      - Each machine and queue runs on a separate thread. The Machine and Queue classes implement the Runnable interface, and new threads are created to handle their operations independently. This allows multiple machines to process products concurrently.

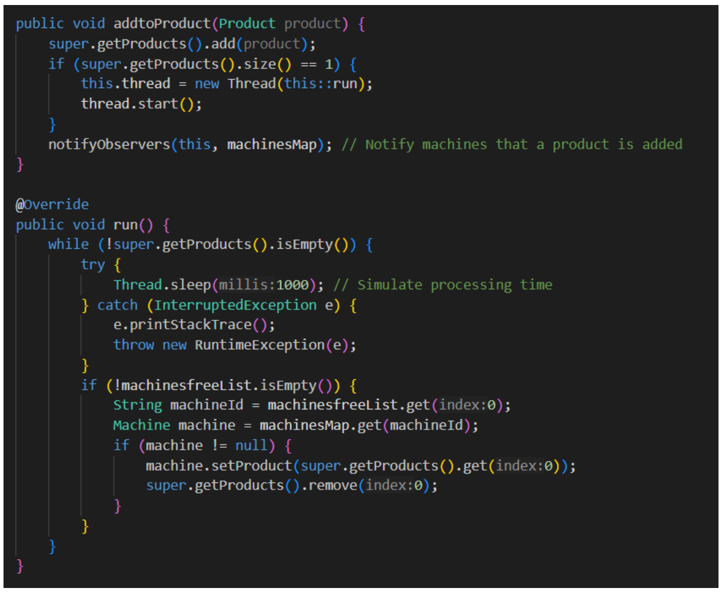
- **Code**



```java
public synchronized void setProduct(Product product) {
    this.product = product;
    System.out.println(product.getColor());
    this.setColor(product.getColor());
    if (webSocketHandler != null) {
        try {
            System.out.println(x:"hello");
            webSocketHandler.broadcastUpdate();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    machineNotifyBusy();
    Thread thread = new Thread(this::run);
    thread.start();
}
```



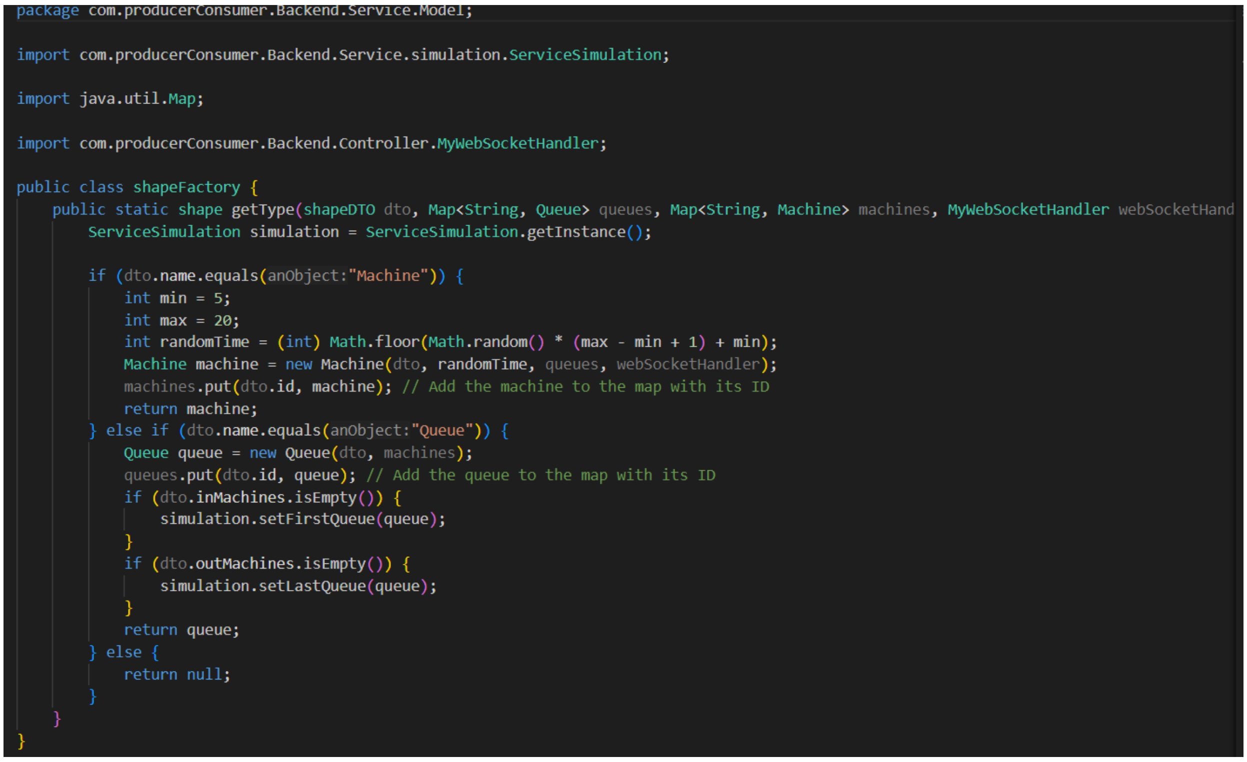
```java
@Override
public void run() {
    try {
        busy = true;
        Thread.sleep(time * 1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
        throw new RuntimeException(e);
    }

    if (product != null) {
        System.out.println("Machine " + getId());
        Queue queue = queues.get(queueId);
        if (queue != null) {
            queue.addProduct(product);
        }
        product = null;
        setColor(color:"white");
        notifyObservers(this, queues);
          machineNotifyFree();
        busy = false;
    }
}
```



```java
public void addtoProduct(Product product) {
    super.getProducts().add(product);
    if (super.getProducts().size() == 1) {
        this.thread = new Thread(this::run);
        thread.start();
    }
    notifyObservers(this, machinesMap); // Notify machines that a product is added
}

@Override
public void run() {
    while (!super.getProducts().isEmpty()) {
        try {
            Thread.sleep(millis:1000); // Simulate processing time
        } catch (InterruptedException e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        }
        if (!machinesfreeList.isEmpty()) {
            String machineId = machinesfreeList.get(index:0);
            Machine machine = machinesMap.get(machineId);
            if (machine != null) {
                machine.setProduct(super.getProducts().get(index:0));
                super.getProducts().remove(index:0);
            }
        }
    }
}
```

## 3. <u>**Factory Pattern**</u>:

- **Description**: when the exact types of objects to be created may vary or need to be determined at runtime, enabling flexibility and extensibility in object creation
- **Implementation in Code**:
  - **Classes Involved**: shapeFactory
  - **Methods**: getType()
  - **Usage**:
    - The shapeFactory class contains a static method getType() that returns a new instance of Machine or Queue based on the shapeDTO input.
    - This allows for the creation of different shapes (machines and queues) without specifying their exact classes.



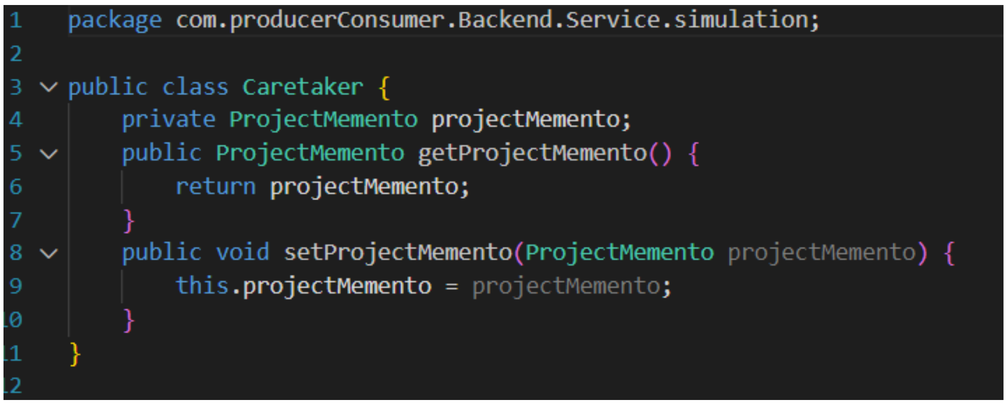
```java
package com.producerConsumer.Backend.Service.Model;

import com.producerConsumer.Backend.Service.simulation.ServiceSimulation;

import java.util.Map;

import com.producerConsumer.Backend.Controller.MyWebSocketHandler;

public class shapeFactory {
    public static shape getType(shapeDTO dto, Map<String, Queue> queues, Map<String, Machine> machines, MyWebSocketHandler webSocketHand
        ServiceSimulation simulation = ServiceSimulation.getInstance();

        if (dto.name.equals(anObject:"Machine")) {
            int min = 5;
            int max = 20;
            int randomTime = (int) Math.floor(Math.random() * (max - min + 1) + min);
            Machine machine = new Machine(dto, randomTime, queues, webSocketHandler);
            machines.put(dto.id, machine); // Add the machine to the map with its ID
            return machine;
        } else if (dto.name.equals(anObject:"Queue")) {
            Queue queue = new Queue(dto, machines);
            queues.put(dto.id, queue); // Add the queue to the map with its ID
            if (dto.inMachines.isEmpty()) {
                simulation.setFirstQueue(queue);
            }
            if (dto.outMachines.isEmpty()) {
                simulation.setLastQueue(queue);
            }
            return queue;
        } else {
            return null;
        }
    }
}
```

## 4. <u>**Snapshot Pattern**</u>:

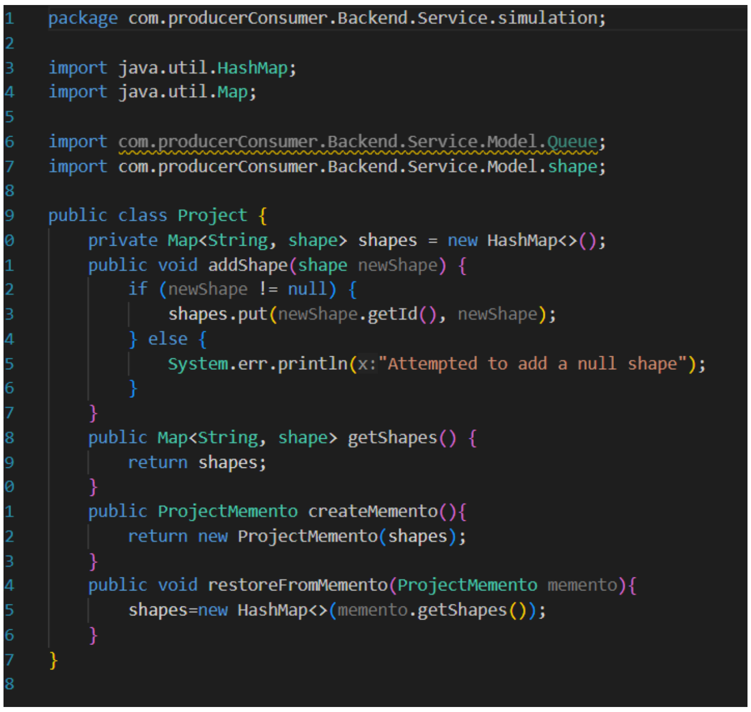
- **Description**: The snapshot pattern captures and externalizes an object's internal state so that the object can be restored to this state later. This pattern is useful for replay simulation.
- **Implementation in Code**:

- **ClassesInvolved**: Caretaker, Project, ProjectMemento, ServiceSimulation
- **Methods**: createMemento(), restoreFromMemento(), saveSate(), restoreState()
- **Usage**:
  - The Project class can create a memento of its current state using the createMemento() method.
  - The Caretaker class stores this memento.
  - The ServiceSimulation class uses the saveSate() and restoreState() methods to save and restore the state of the project, allowing users to replay the previous simulation.

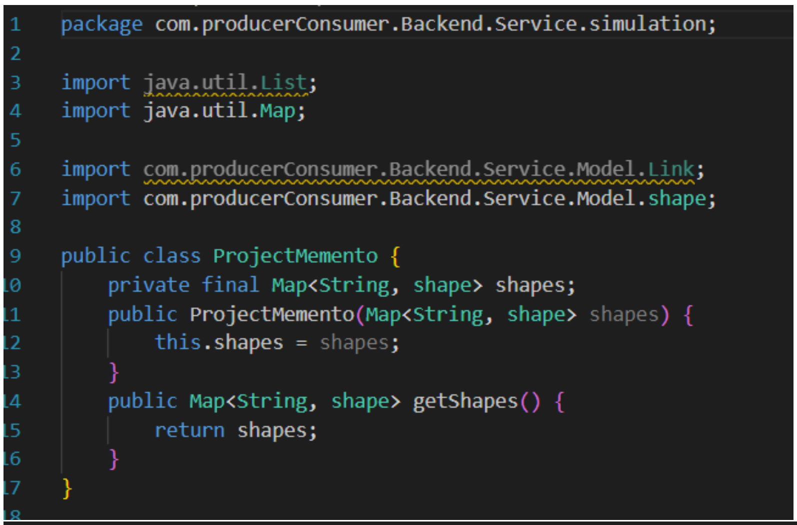
- Code



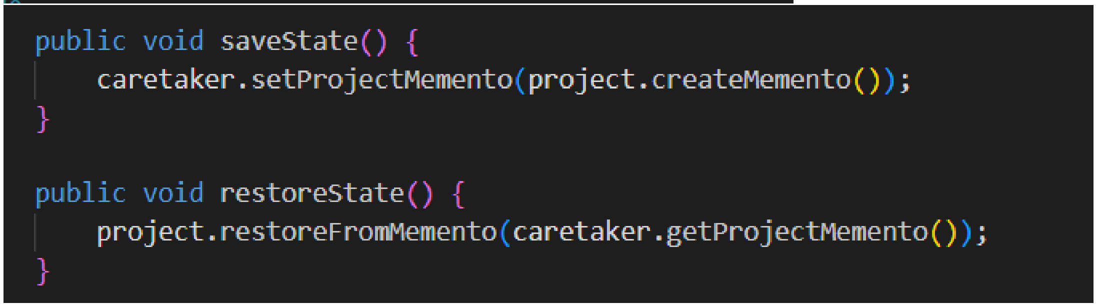
```java
package com.producerConsumer.Backend.Service.simulation;

public class Caretaker {
    private ProjectMemento projectMemento;
    public ProjectMemento getProjectMemento() {
        return projectMemento;
    }
    public void setProjectMemento(ProjectMemento projectMemento) {
        this.projectMemento = projectMemento;
    }
}
```



```java
package com.producerConsumer.Backend.Service.simulation;

import java.util.HashMap;
import java.util.Map;

import com.producerConsumer.Backend.Service.Model.Queue;
import com.producerConsumer.Backend.Service.Model.shape;

public class Project {
    private Map<String, shape> shapes = new HashMap<>();
    public void addShape(shape newShape) {
        if (newShape != null) {
            shapes.put(newShape.getId(), newShape);
        } else {
            System.err.println(x:"Attempted to add a null shape");
        }
    }
    public Map<String, shape> getShapes() {
        return shapes;
    }
    public ProjectMemento createMemento(){
        return new ProjectMemento(shapes);
    }
    public void restoreFromMemento(ProjectMemento memento){
        shapes=new HashMap<>(memento.getShapes());
    }
}
```



```java
package com.producerConsumer.Backend.Service.simulation;

import java.util.List;
import java.util.Map;

import com.producerConsumer.Backend.Service.Model.Link;
import com.producerConsumer.Backend.Service.Model.shape;

public class ProjectMemento {
    private final Map<String, shape> shapes;
    public ProjectMemento(Map<String, shape> shapes) {
        this.shapes = shapes;
    }
    public Map<String, shape> getShapes() {
        return shapes;
    }
}
```

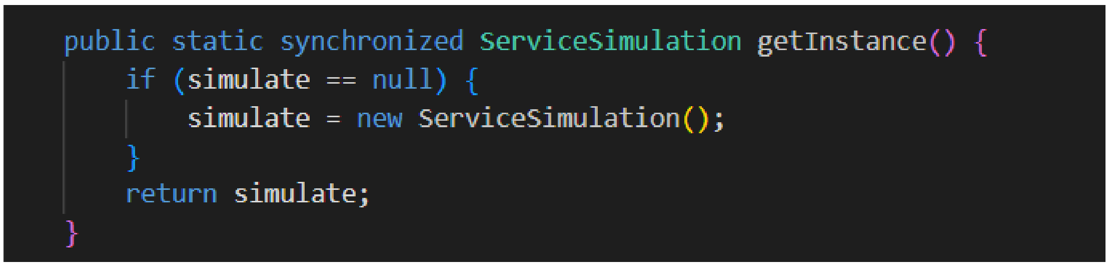


```java
public void saveState() {
    caretaker.setProjectMemento(project.createMemento());
}

public void restoreState() {
    project.restoreFromMemento(caretaker.getProjectMemento());
}
```

---

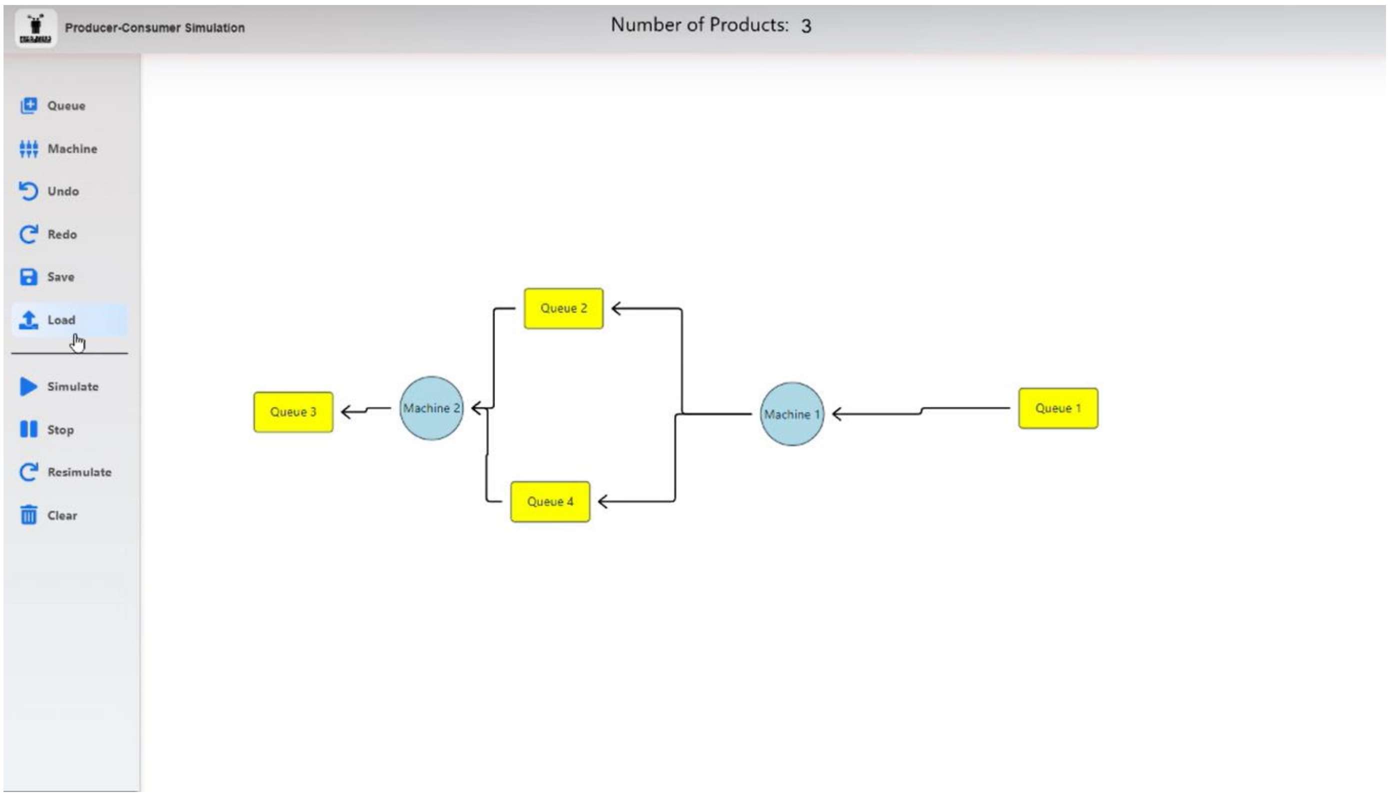
## 5. Singleton Pattern:

- **Description**: The singleton pattern ensures that a class has only one instance and provides a global point of access to it. This pattern is useful for managing shared resources or configurations.
- **Implementation in Code**:
  - **Classes Involved**: ServiceSimulation
  - **Methods**: getInstance()
  - **Usage**:
    - The ServiceSimulation class has a private constructor and a static method getInstance() that returns the single instance of the class.
    - This ensures that there is only one instance of the ServiceSimulation class, providing a global point of access to it.



```java
public static synchronized ServiceSimulation getInstance() {
    if (simulate == null) {
        simulate = new ServiceSimulation();
    }
    return simulate;
}
```

## User Guide:



### The interface consists of two main parts:

**1-Control Panel**: This panel contains buttons to control the simulation and manage saved states.

- **Queue**: This button likely adds a new queue to the simulation diagram.
- **Machine**: This button likely adds a new machine to the simulation diagram.
- **Undo**: Undoes the last action.
- **Redo**: Redoes the last undone action.
- **Save**: Saves the current simulation state to a file.
- **Load**: Loads a previously saved simulation state.
- **Simulate**: Starts the simulation.
- **Stop**: Pauses the simulation.
- **Resimulate**: Restarts the simulation from the beginning.
- **Clear:** Clears the entire simulation.

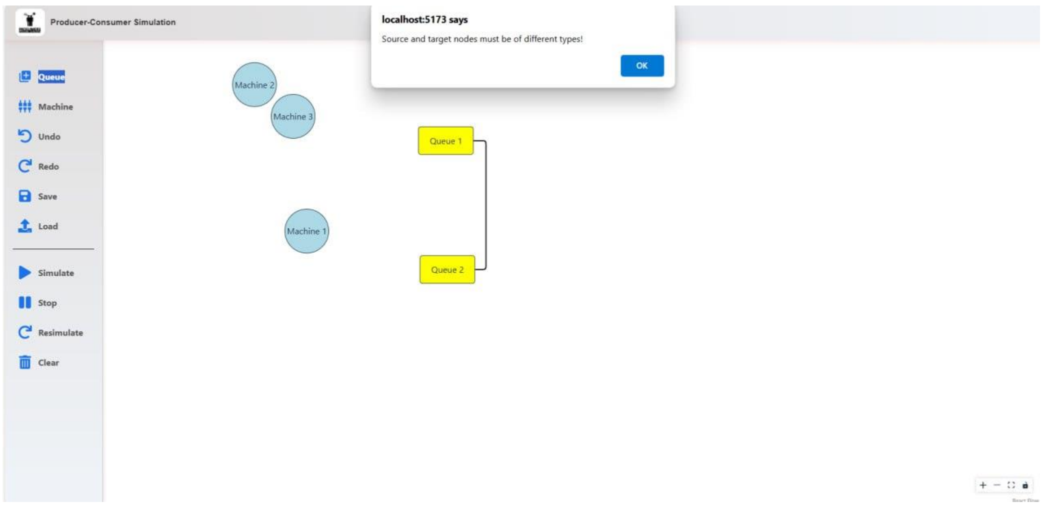
**2-Simulation Area** : This area displays the visual representation of the producer-consumer system. Yellow rectangles represent queues, and light-blue circles represent machines. Lines indicate the flow of products. The "Number of Products" displays the total number of products in the system.

## How to Use the Application:

**Setting up the Simulation**:Use the "Queue" and "Machine" buttons to add queues and machines to the simulation diagram. Arrange them to define the desired workflow.

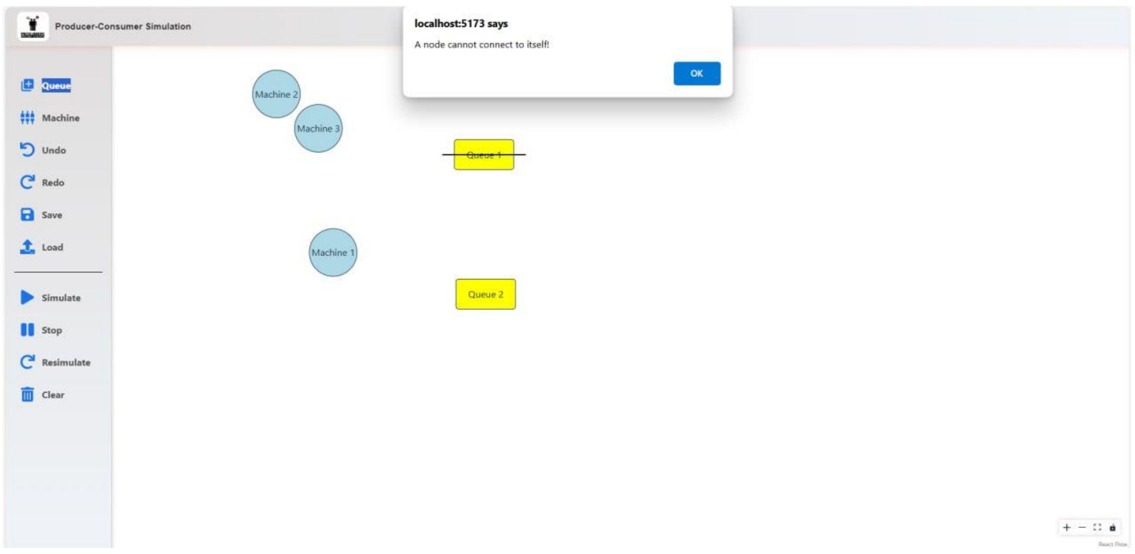
**Adding Products**: Before starting the simulation there is an input in the Top-center of the screen that should be filled with the products number

- Simulation must start and end with queue
- No Queue can be connected to another Queue, No Machine can be connected to another Machine


localhost:5173 says

Source and target nodes must be of different types!

- No shape (Machine, Queue) Can be connected to itself


localhost:5173 says

A node cannot connect to itself!

**Starting the Simulation**: Click the "Simulate" button to start the simulation. The application will likely visually show products moving through the queues and being processed by the machines.

- When a machine gets a product it glow with a random color until the consuming time end.
- WebSocket protocol was used to send notifications from backend to frontend, when a product is moved from a queue to a machine or from a machine to a queue.

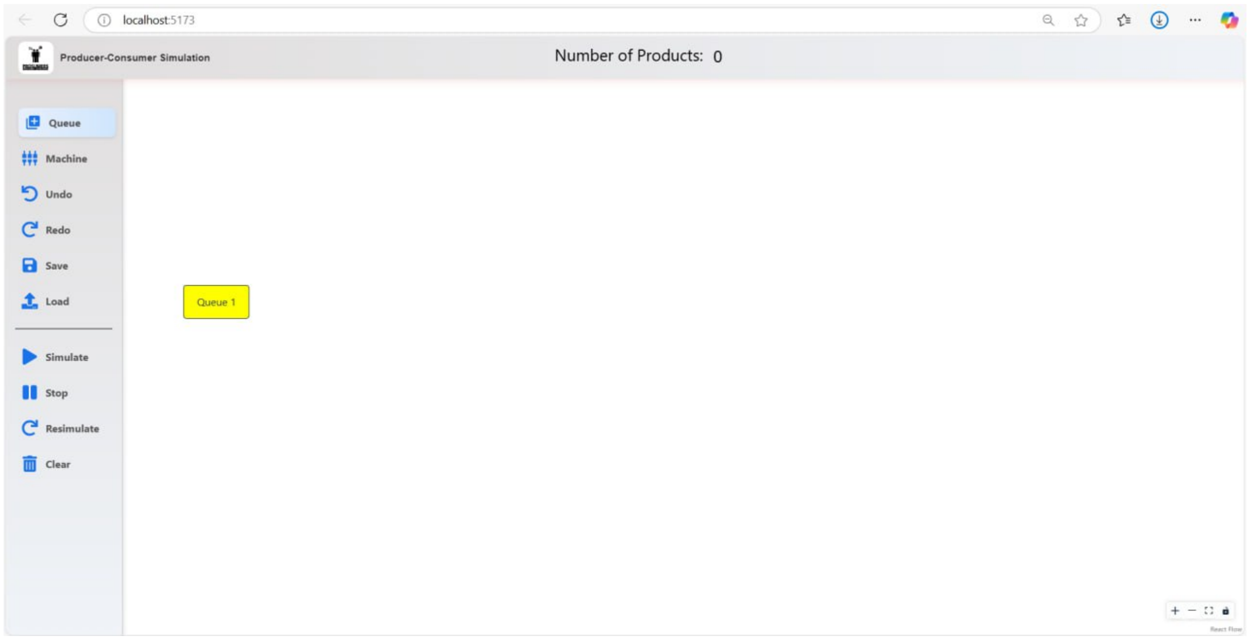
**Saving and Loading**: Use the "Save" button to save the current state of your simulation, including the arrangement of queues and machines, and the number of products as a json file.

**Click "Resimulate"** to restart the simulation from the beginning.

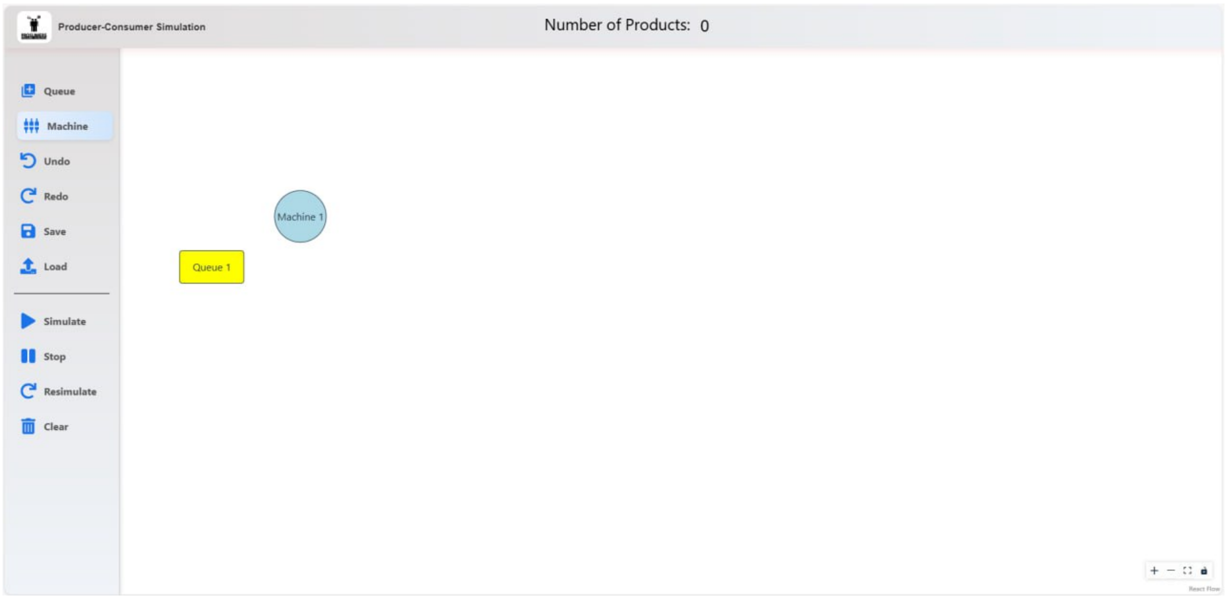
**Clearing the Simulation:**Click "Clear" to delete everything from the simulation area and start fresh.

## Snapshots of UI:

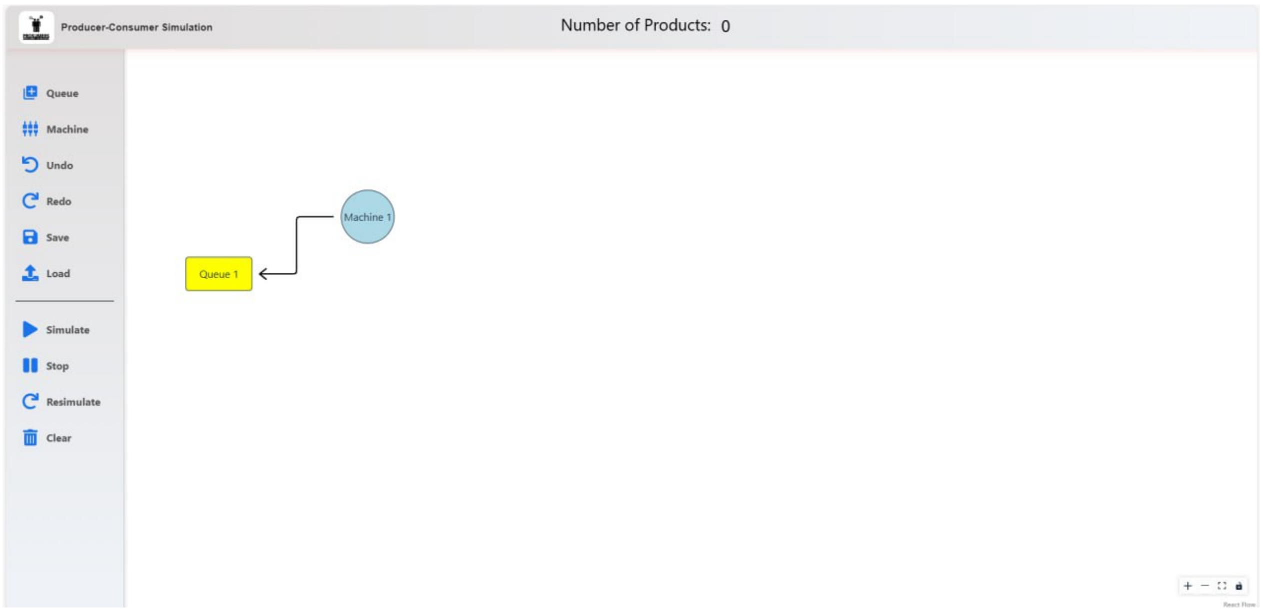
### Adding Queues



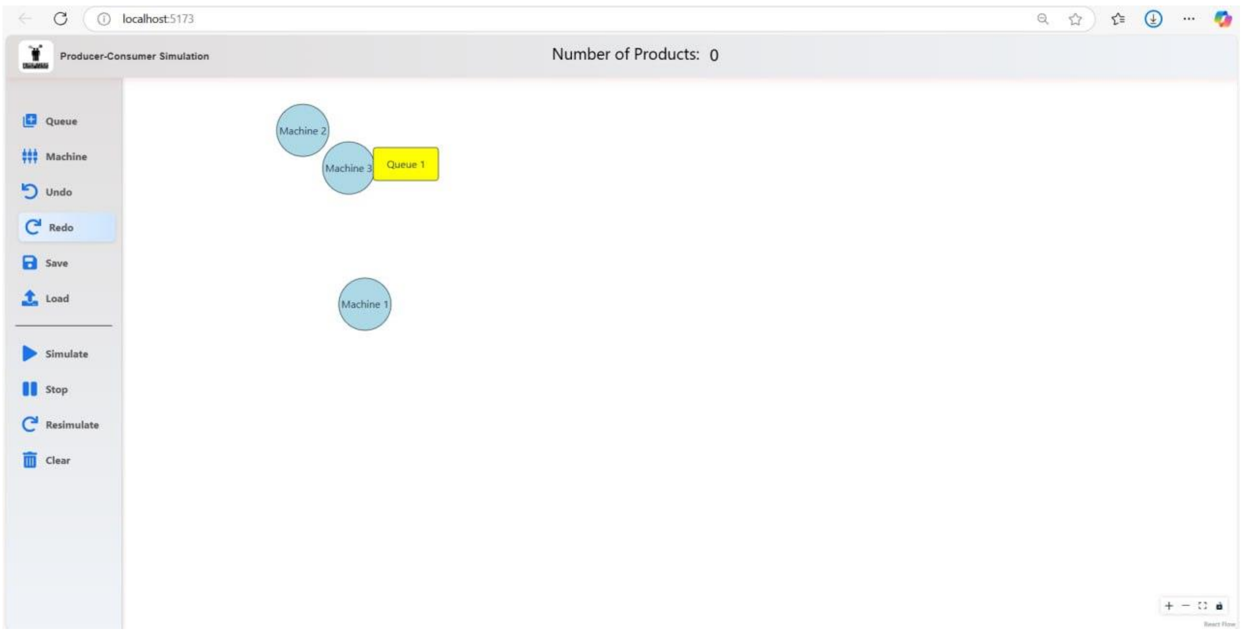
## Adding Machines

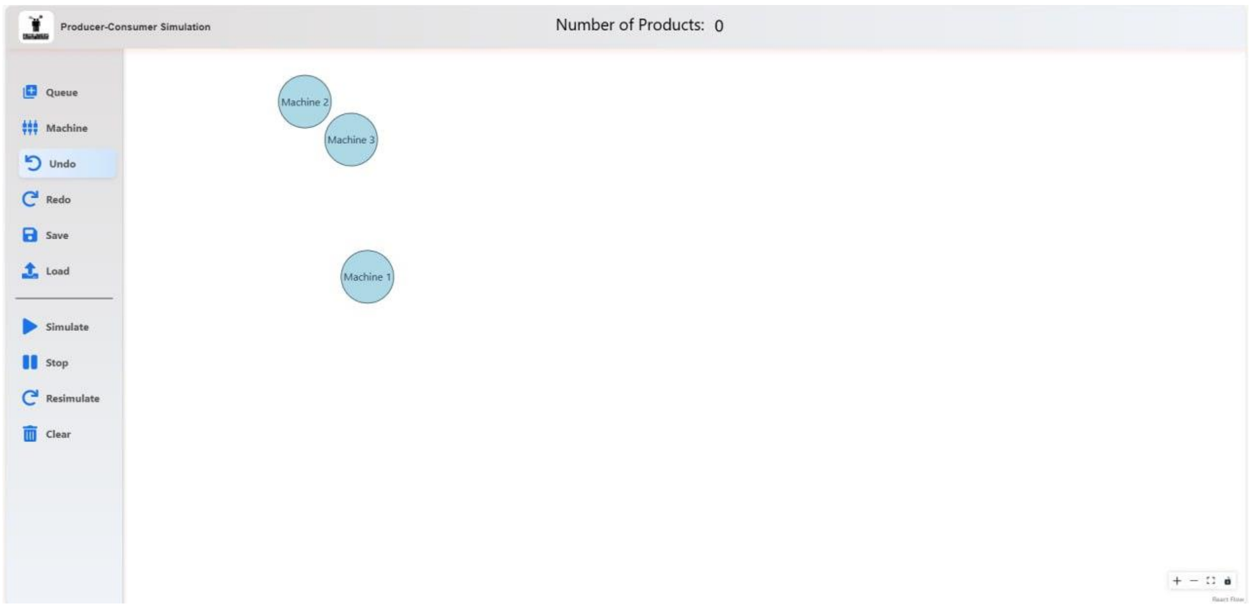


## Link between them

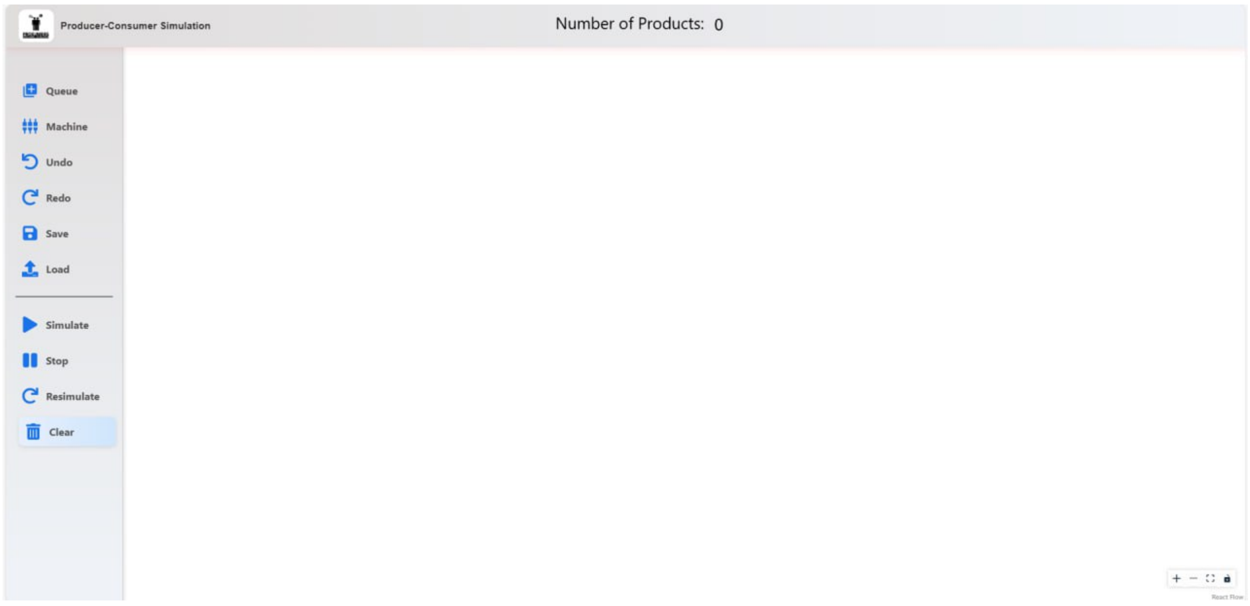


## Undo and Redo





## Clear


Producer-Consumer Simulation
Number of Products: 0
Queue
Machine
Undo
Redo
Save
Load
Simulate
Stop
Resimulate
Clear