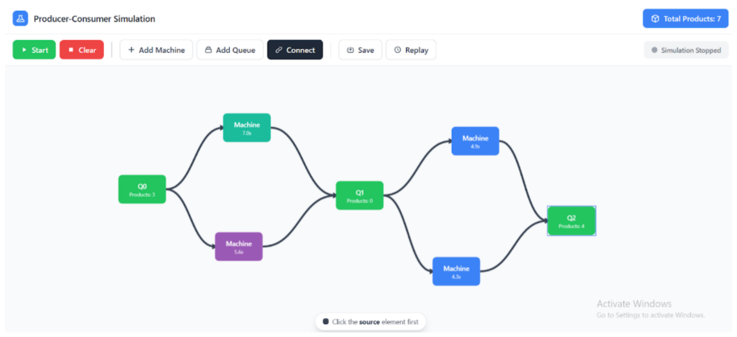
Producer-Consumer Simulation System

A Multithreaded Manufacturing Simulation

CSE 223: Programming 2 — Assignment 5

Alexandria University

Faculty of Engineering — Computer and Systems Engineering Department



Team Members

Name	ID
Jana Mohamed Rashed	23010359
Ahmed Sherif Abd El-Moniem Ali Abo El-Soud	23010199
Mohamed Radwan Mahmoud	23010745
Yousef Walid Abd El Wahab	23011017
Nour Sameh Samir	23010925

December 31, 2024

Contents

Chapter 1

Introduction

1.1 Problem Overview

The Producer-Consumer problem is one of the most fundamental synchronization problems in concurrent programming. It involves two types of processes that share a common, fixed-size buffer:

- **Producers**: Generate data items and place them into the buffer
- **Consumers**: Remove data items from the buffer and process them

The challenge lies in ensuring that producers do not add data when the buffer is full and consumers do not remove data when the buffer is empty, all while maintaining data integrity in a multithreaded environment.

1.2 Project Objectives

This project implements a visual manufacturing simulation that demonstrates the Producer-Consumer pattern with the following features:

1. **Dynamic Graph Building**: Users can create and connect queues (buffers) and machines (consumers/producers) in a visual interface
2. **Real-time Simulation**: Products flow through the system in real-time with visual feedback
3. **Thread-safe Operations**: Proper synchronization using Java's concurrency utilities
4. **Snapshot & Replay**: Save and restore simulation states for analysis
5. **Server-Sent Events (SSE)**: Real-time updates from backend to frontend

1.3 Technology Stack

1.3.1 Backend

- **Java 17**: Core programming language
- **Spring Boot 3.4.1**: Application framework
- **Spring WebFlux**: Reactive programming for SSE streaming
- **Lombok**: Boilerplate code reduction
- **Maven**: Build and dependency management

1.3.2 Frontend

- **Angular 19**: Frontend framework with standalone components
- **TypeScript**: Type-safe JavaScript
- **Tailwind CSS**: Utility-first CSS framework
- **RxJS**: Reactive Extensions for JavaScript

Chapter 2

System Architecture

2.1 High-Level Architecture

The system follows a client-server architecture with clear separation of concerns:

Frontend (Angular 19) Simulation Canvas | UI Components | State Management
↓ REST API + SSE ↓
Backend (Spring Boot) Controllers | Services | Models
↓
Simulation Engine Machine Runners | Queue Service | Event Broadcasting

Figure 2.1: High-Level System Architecture

2.2 Class Diagram

Figure 2.2 shows the complete class diagram of the backend system, illustrating the relationships between services and models. Figure 2.3 provides a detailed view of the domain model relationships, showing how `SimulationState` contains and manages `Queue`, `Machine`, `Connection`, and `Product` objects, as well as its relationship with `SimulationSnapshot` for state persistence.

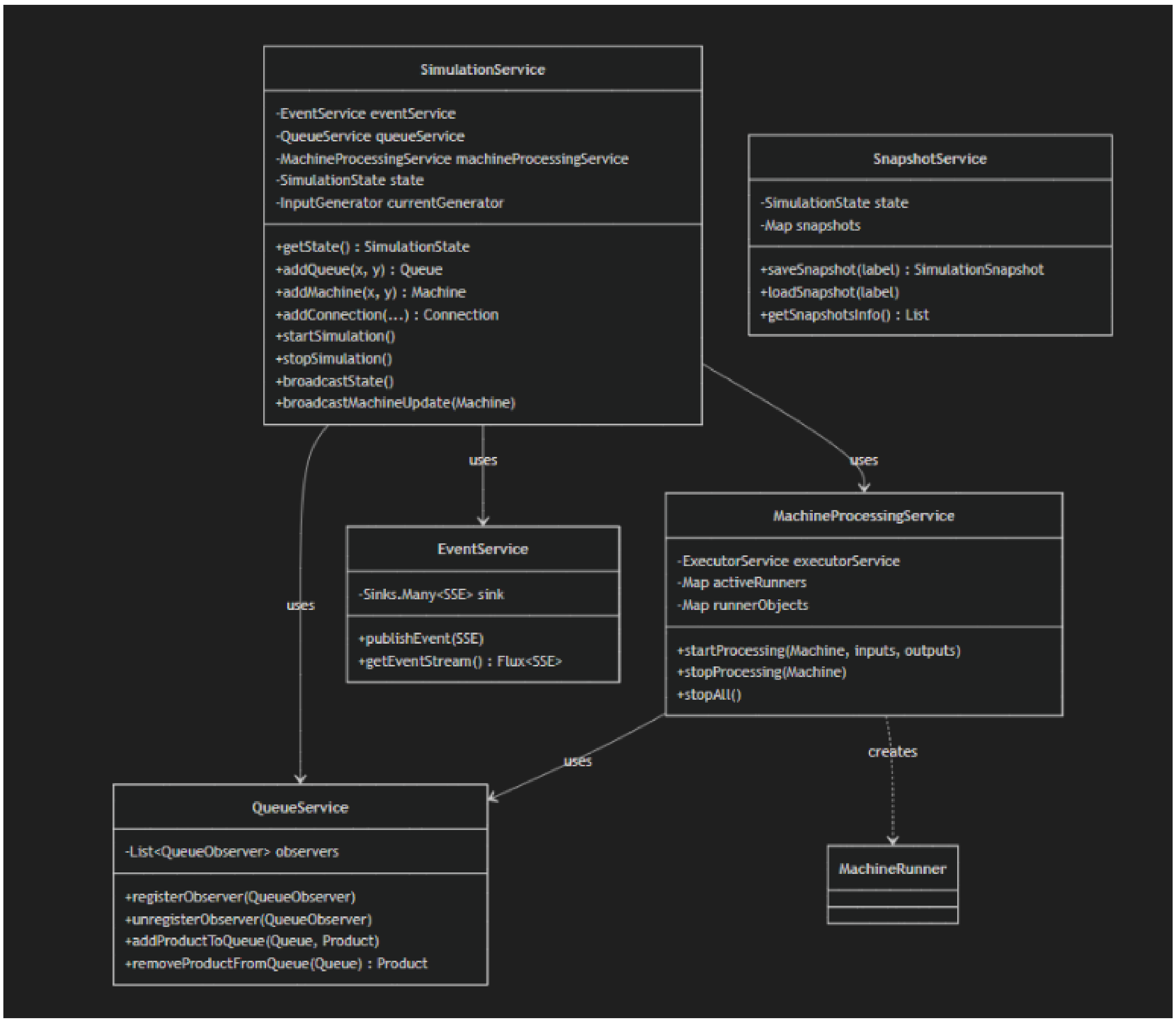

SimulationService
-EventService eventService
-QueueService queueService
-MachineProcessingService machineProcessingService
-SimulationState state
-InputGenerator currentGenerator
+getState() : SimulationState
+addQueue(x, y) : Queue
+addMachine(x, y) : Machine
+addConnection(...) : Connection
+startSimulation()
+stopSimulation()
+broadcastState()
+broadcastMachineUpdate(Machine)
SnapshotService
-SimulationState state
-Map snapshots
+saveSnapshot(label) : SimulationSnapshot
+loadSnapshot(label)
+getSnapshotsInfo() : List
uses
uses
uses
EventService
-Sinks.Many<SSE> sink
+publishEvent(SSE)
+getEventStream() : Flux<SSE>
MachineProcessingService
-ExecutorService executorService
-Map activeRunners
-Map runnerObjects
+startProcessing(Machine, inputs, outputs)
+stopProcessing(Machine)
+stopAll()
uses
creates
QueueService
-List<QueueObserver> observers
+registerObserver(QueueObserver)
+unregisterObserver(QueueObserver)
+addProductToQueue(Queue, Product)
+removeProductFromQueue(Queue) : Product
MachineRunner

Figure 2.2: Backend Class Diagram

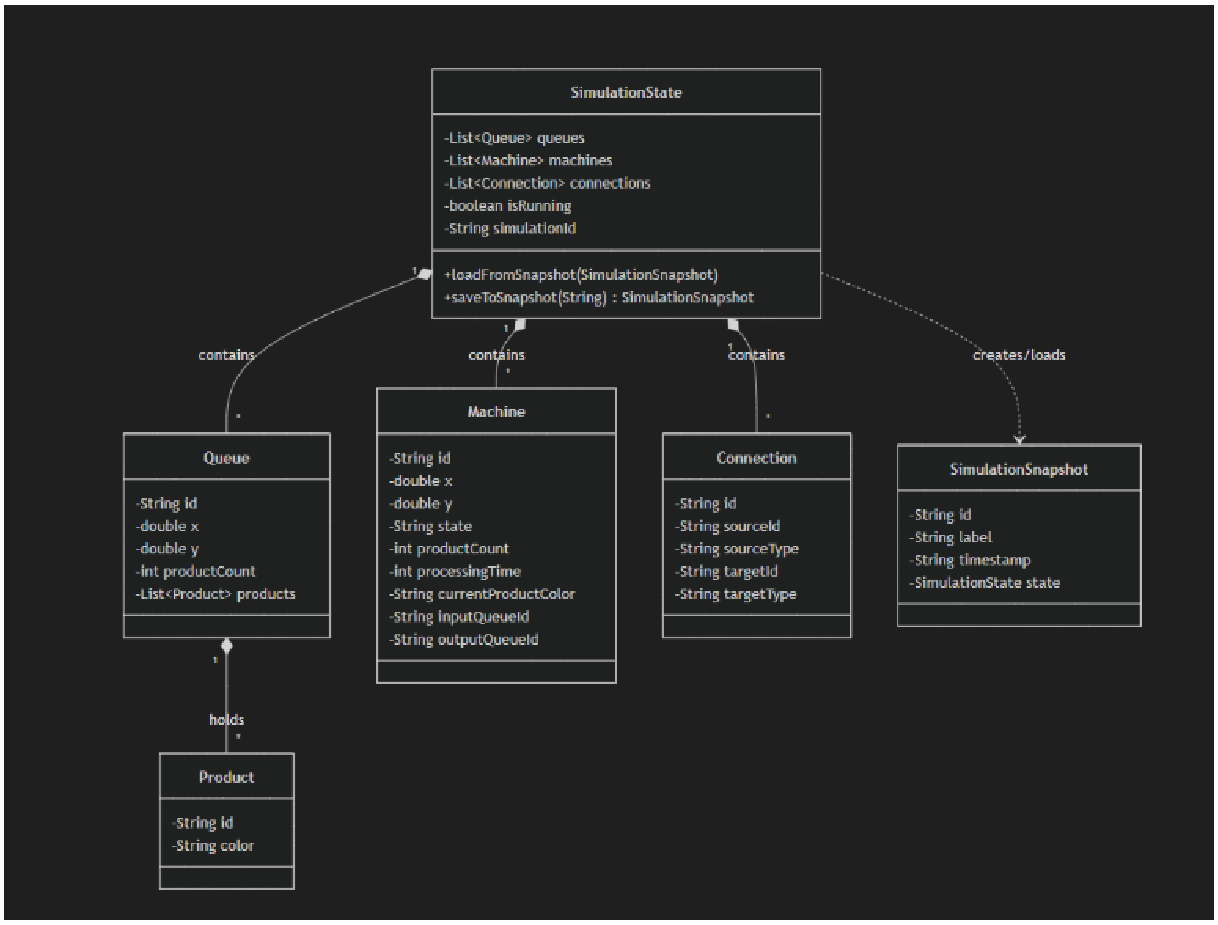


Figure 2.3: Simulation State Class Diagram - Domain Model Relationships

2.3 Backend Package Structure

The backend is organized into the following packages:

- `com.producesconsumer.backend.config`: Application configuration (CORS, thread pools)
- `com.producesconsumer.backend.controller`: REST API endpoints
- `com.producesconsumer.backend.dto`: Data Transfer Objects
- `com.producesconsumer.backend.model`: Domain models (Queue, Machine, Product, etc.)
- `com.producesconsumer.backend.observer`: Observer pattern implementation
- `com.producesconsumer.backend.service`: Business logic services

2.4 Frontend Component Structure

The Angular frontend is organized as follows:

- `src/app/app.ts`: Main application component
- `src/app/components/simulation-canvas`: Visual canvas for elements
- `src/app/components/toolbar`: Control toolbar
- `src/app/components/snapshot-list`: Snapshot management UI
- `src/app/services/simulation.service.ts`: Core service for API communication
- `src/app/models/simulation.model.ts`: TypeScript interfaces

Chapter 3

Design Patterns

This chapter details the design patterns used in the implementation, their purpose, and how they are applied in our codebase.

3.1 Observer Pattern

3.1.1 Purpose

The Observer pattern establishes a one-to-many dependency between objects so that when one object (the subject) changes state, all its dependents (observers) are notified and updated automatically.

3.1.2 Application in Our System

The Observer pattern is used to notify components when products are added to or removed from queues. This enables:

- **Real-time UI updates**: The frontend receives instant notifications via SSE
- **Machine thread wakeup**: Machines waiting for products are notified when new products arrive (Guarded Suspension)

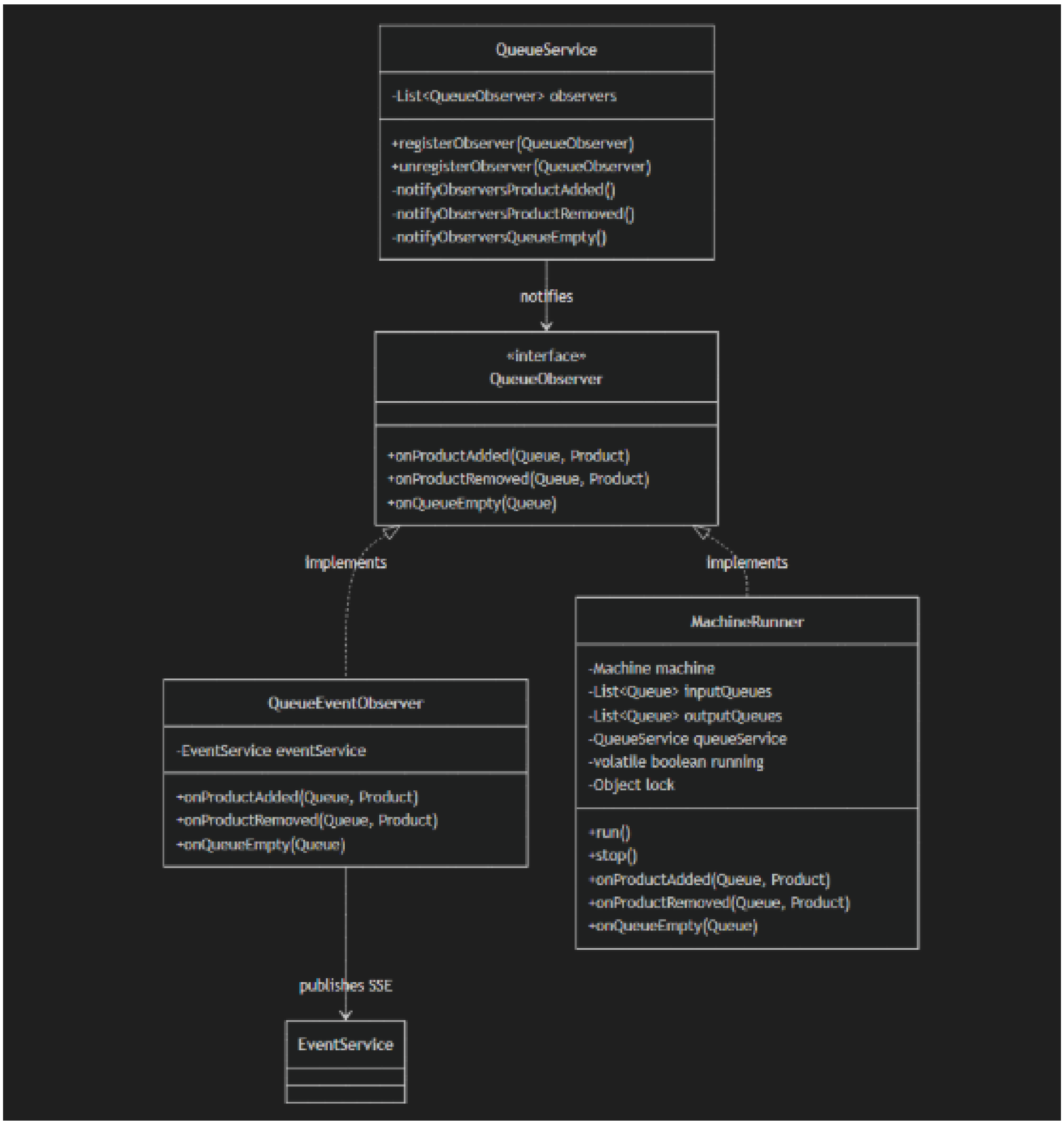

QueueService
-List<QueueObserver> observers
+registerObserver(QueueObserver)
+unregisterObserver(QueueObserver)
-notifyObserversProductAdded()
-notifyObserversProductRemoved()
-notifyObserversQueueEmpty()
notifies
«interface»
QueueObserver
+onProductAdded(Queue, Product)
+onProductRemoved(Queue, Product)
+onQueueEmpty(Queue)
Implements
Implements
MachineRunner
-Machine machine
-List<Queue> inputQueues
-List<Queue> outputQueues
-QueueService queueService
-volatile boolean running
-Object lock
+run()
+stop()
+onProductAdded(Queue, Product)
+onProductRemoved(Queue, Product)
+onQueueEmpty(Queue)
QueueEventObserver
-EventService eventService
+onProductAdded(Queue, Product)
+onProductRemoved(Queue, Product)
+onQueueEmpty(Queue)
publishes SSE
EventService

Figure 3.1: Observer Pattern Class Diagram

3.1.3 Code Implementation

QueueObserver Interface

```java
package com.producesconsumer.backend.observer;

import com.producesconsumer.backend.model.Product;
import com.producesconsumer.backend.model.Queue;

public interface QueueObserver {
    void onProductAdded(Queue queue, Product product);
    void onProductRemoved(Queue queue, Product product);
    void onQueueEmpty(Queue queue);
}
```

Listing 3.1: QueueObserver Interface

QueueService (Subject)

```java
@Service
@Slf4j
@RequiredArgsConstructor
public class QueueService {

    private final List<QueueObserver> observers =
        new CopyOnWriteArrayList<>();

    public void registerObserver(QueueObserver observer) {
        if (!observers.contains(observer)) {
            observers.add(observer);
            log.debug("Observer registered. Total: {}", observers.size());
        }
    }

    public void unregisterObserver(QueueObserver observer) {
        observers.remove(observer);
    }

    public synchronized void addProductToQueue(Queue queue, Product product) {
        queue.getProducts().add(product);
        queue.setProductCount(queue.getProducts().size());
        notifyObserversProductAdded(queue, product);
    }

    private void notifyObserversProductAdded(Queue queue, Product product) {
        for (QueueObserver observer : observers) {
            try {
                observer.onProductAdded(queue, product);
            } catch (Exception e) {
                log.error("Error notifying observer: {}", e.getMessage());
            }
        }
    }
}
```

Listing 3.2: QueueService - Observer Registration and Notification

QueueEventObserver (Concrete Observer)

```java
@Component
@Slf4j
```

```java
@RequiredArgsConstructor
public class QueueEventObserver implements QueueObserver {

    private final EventService eventService;

    @Override
    public void onProductAdded(Queue queue, Product product) {
        log.debug("Product added to queue {}: {}",
                  queue.getId(), product.getId());

        QueueEventPayload payload = new QueueEventPayload(
            "PRODUCT_ADDED",
            queue.getId(),
            product.getId(),
            product.getColor(),
            queue.getProductCount()
        );
        eventService.publishEvent(new SSE("QUEUE_EVENT", payload));
    }

    @Override
    public void onProductRemoved(Queue queue, Product product) {
        // Similar implementation...
    }

    @Override
    public void onQueueEmpty(Queue queue) {
        // Similar implementation...
    }
}
```

Listing 3.3: QueueEventObserver - Publishes SSE Events

3.2 Producer-Consumer Pattern with Guarded Suspension

3.2.1 Purpose

The Producer-Consumer pattern coordinates producers and consumers accessing a shared buffer. Guarded Suspension is a synchronization pattern where a thread waits (suspends) until a condition becomes true before proceeding.

3.2.2 Application in Our System

The `MachineRunner` class implements both consumer (taking from input queues) and producer (adding to output queues) behavior. When no products are available in input queues, the machine thread suspends using `wait()` and is awakened when new products arrive via the Observer pattern.

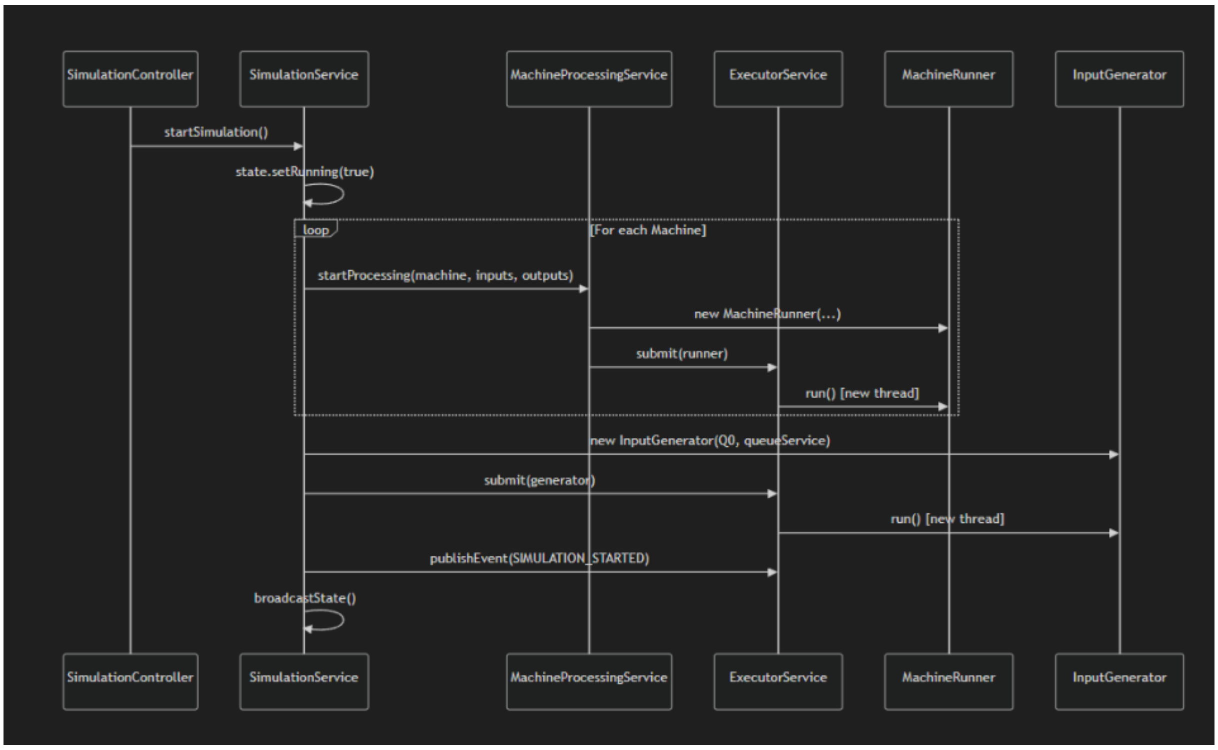


Figure 3.2: Sequence Diagram - Simulation Start

3.2.3 Code Implementation

```java
@Slf4j
public class MachineRunner implements Runnable, QueueObserver {
    private final Machine machine;
    private final List<Queue> inputQueues;
    private final List<Queue> outputQueues;
    private final QueueService queueService;
    private final SimulationService simulationService;

    private volatile boolean running = true;
    private final Random random = new Random();

    // Guarded Suspension lock object
    public final Object lock = new Object();

    @Override
    public void run() {
        log.info("Machine {} started", machine.getId());
        queueService.registerObserver(this);

        while (running) {
            try {
                // Consumer: Try to fetch a product
                Product productToProcess = fetchProductFromInput();

                if (productToProcess == null) {
                    machine.setState("idle");

                    // Guarded Suspension: wait for product
                    synchronized (lock) {
                        productToProcess = fetchProductFromInput();
                        if (productToProcess == null && running) {
                            lock.wait(); // Suspend thread
                            continue;
                        }
                    }
                }

                if (!running) break;

                // Processing
                machine.setState("processing");
                machine.setCurrentProductColor(productToProcess.getColor());
                simulationService.broadcastMachineUpdate(machine);

                Thread.sleep(machine.getProcessingTime());

                // Producer: Send to output queue
                if (!outputQueues.isEmpty()) {
                    Queue target = outputQueues.get(
                        random.nextInt(outputQueues.size())
                    );
                    queueService.addProductToQueue(target, productToProcess);
                }

                machine.setState("idle");

            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                running = false;
            }
        }
        queueService.unregisterObserver(this);
    }

    // Observer callback - wake up when product arrives
    @Override
    public void onProductAdded(Queue queue, Product product) {
        if (inputQueues.stream()
                .anyMatch(q -> q.getId().equals(queue.getId()))) {
            synchronized (lock) {
                lock.notify(); // Wake up waiting thread
            }
        }
    }
}
```

Listing 3.4: MachineRunner - Producer-Consumer with Guarded Suspension

3.3 Memento Pattern

3.3.1 Purpose

The Memento pattern captures and externalizes an object's internal state so that the object can be restored to this state later, without violating encapsulation.

3.3.2 Application in Our System

The Memento pattern is used for the Snapshot/Replay functionality:

- **Originator**: `SimulationState` - creates and restores snapshots
- **Memento**: `SimulationSnapshot` - stores the simulation state
- **Caretaker**: `SnapshotService` - manages saved snapshots

3.3.3 Code Implementation

```java
@Data
public class SimulationState {
    private List<Queue> queues;
    private List<Machine> machines;
    private List<Connection> connections;
    private boolean isRunning;

    public SimulationSnapshot saveToSnapshot(String label) {
        SimulationSnapshot snapshot = new SimulationSnapshot();
        snapshot.setLabel(label);

        // Create deep copy of current state
        SimulationState stateCopy = new SimulationState();
        stateCopy.setQueues(this.queues.stream()
            .map(this::deepCopyQueue)
            .collect(Collectors.toCollection(ArrayList::new)));
        stateCopy.setMachines(this.machines.stream()
            .map(this::deepCopyMachine)
            .collect(Collectors.toCollection(ArrayList::new)));
        stateCopy.setConnections(this.connections.stream()
            .map(this::deepCopyConnection)
            .collect(Collectors.toCollection(ArrayList::new)));
        stateCopy.setRunning(false);

        snapshot.setState(stateCopy);
        return snapshot;
    }

    public void loadFromSnapshot(SimulationSnapshot snapshot) {
        SimulationState snapshotState = snapshot.getState();

        // Deep copy from snapshot
        this.queues = snapshotState.getQueues().stream()
            .map(this::deepCopyQueue)
            .collect(Collectors.toCollection(ArrayList::new));
        // ... similar for machines and connections
        this.isRunning = false;
    }
}
```

Listing 3.5: SimulationState - Originator with Deep Copy

```java
@Service
@Slf4j
public class SnapshotService {
    private final Map<String, SimulationSnapshot> snapshots =
        new ConcurrentHashMap<>();

    @Autowired
    private SimulationService simulationService;
    private ObjectMapper objectMapper = new ObjectMapper();
    private final String snapshotsDir = "snapshots";

    public SimulationSnapshot saveSnapshot(String label) {
        SimulationSnapshot snapshot =
            simulationService.getState().saveToSnapshot(label);
        snapshots.put(snapshot.getLabel(), snapshot);

        // Persist to disk
        try {
            String json = objectMapper.writeValueAsString(snapshot);
            Files.createDirectories(Path.of(snapshotsDir));
            Files.writeString(
                Path.of(snapshotsDir, snapshot.getLabel() + ".json"),
                json
            );
        } catch (IOException e) {
            log.error("Failed to save snapshot: {}", e.getMessage());
        }
        return snapshot;
    }

    public SimulationSnapshot loadSnapshot(String label) {
        if (snapshots.containsKey(label)) {
            SimulationSnapshot snapshot = snapshots.get(label);
            simulationService.getState().loadFromSnapshot(snapshot);
            return snapshot;
        }
        // Load from disk if not in cache...
    }
}
```

Listing 3.6: SnapshotService - Caretaker with Persistence

3.4 Singleton Pattern

3.4.1 Purpose

The Singleton pattern ensures a class has only one instance and provides a global point of access to it.

3.4.2 Application in Our System

Spring's dependency injection naturally implements the Singleton pattern for all services annotated with `@Service`. Each service is instantiated once and shared across the application:

```java
@Service  // Singleton by default
@RequiredArgsConstructor
public class SimulationService {
    private final EventService eventService;      // Injected singleton
    private final QueueService queueService;      // Injected singleton
    private final SimulationState state = new SimulationState();
    // ...
}
```

Listing 3.7: Singleton Services via Spring

3.5 Dependency Injection Pattern

3.5.1 Purpose

Dependency Injection (DI) is a design pattern where dependencies are provided to a class rather than the class creating them. This promotes loose coupling and easier testing.

3.5.2 Application in Our System

All services use constructor injection via Lombok's `@RequiredArgsConstructor`:

```java
@Service
@RequiredArgsConstructor  // Generates constructor for final fields
public class MachineProcessingService {
    private final ExecutorService executorService;  // Injected
    private final QueueService queueService;        // Injected

    @Lazy  // Break circular dependency
    private SimulationService simulationService;
    // ...
}
```

Listing 3.8: Dependency Injection in Services

Chapter 4

Core Implementation Details

4.1 Domain Models

4.1.1 Queue Model

```java
@Data
public class Queue {
    private String id;
    private double x;
    private double y;
    private int productCount;
    private List<Product> products;

    public Queue() {
        this.products = new CopyOnWriteArrayList<>();
        this.productCount = 0;
    }
}
```

Listing 4.1: Queue Model

4.1.2 Machine Model

```java
@Data
public class Machine {
    private String id;
    private double x;
    private double y;
    private String state;           // "idle" or "processing"
    private int productCount;
    private int processingTime;     // in milliseconds
    private String currentProductColor;
    private String inputQueueId;
    private String outputQueueId;
}
```

Listing 4.2: Machine Model

4.2 REST API Controller

```java
@RestController
@RequestMapping("/api/simulation")
@RequiredArgsConstructor
public class SimulationController {

    private final SimulationService simulationService;
    private final SnapshotService snapshotService;
    private final EventService eventService;

    // SSE Endpoint for real-time updates
    @GetMapping(path = "/events",
                produces = MediaType.TEXT_EVENT_STREAM_VALUE)
    public Flux<ServerSentEvent<SSE>> streamEvents() {
        return eventService.getEventStream()
            .map(event -> ServerSentEvent.<SSE>builder()
                .data(event).build());
    }

    @GetMapping("/state")
    public ApiResponse<SimulationState> getState() {
        return ApiResponse.success(simulationService.getState());
    }

    @PostMapping("/queues")
    public ApiResponse<Queue> addQueue(@RequestBody PositionRequest req)
    {
        return ApiResponse.success(
            simulationService.addQueue(req.getX(), req.getY()));
    }

    @PostMapping("/machines")
    public ApiResponse<Machine> addMachine(@RequestBody PositionRequest
   req) {
        return ApiResponse.success(
            simulationService.addMachine(req.getX(), req.getY()));
    }

    @PostMapping("/connections")
    public ApiResponse<Connection> addConnection(
            @RequestBody ConnectionRequest request) {
        return ApiResponse.success(simulationService.addConnection(
            request.getSourceId(), request.getSourceType(),
            request.getTargetId(), request.getTargetType()));
    }

    @PostMapping("/start")
    public ApiResponse<Void> startSimulation() {
        simulationService.startSimulation();
        return ApiResponse.success(null);
    }

    @PostMapping("/stop")
    public ApiResponse<Void> stopSimulation() {
        simulationService.stopSimulation();
        return ApiResponse.success(null);
    }
}
```

Listing 4.3: SimulationController

4.3 Event Service (SSE)

```java
@Service
@Slf4j
public class EventService {

    private final Sinks.Many<SSE> eventSink =
        Sinks.many().multicast().onBackpressureBuffer();

    public Flux<SSE> getEventStream() {
        return eventSink.asFlux();
    }

    public void publishEvent(SSE event) {
        Sinks.EmitResult result = eventSink.tryEmitNext(event);
        log.debug("SSE Event: {} - Result: {}", event.getType(), result)
    ;
        if (result.isFailure()) {
            log.error("Failed to publish SSE event: {}", event.getType()
    );
        }
    }
}
```

Listing 4.4: EventService - Reactive SSE Publishing

4.4 Input Generator

```java
@Slf4j
public class InputGenerator implements Runnable {
    private final Queue targetQueue;
    private final QueueService queueService;
    private volatile boolean running = true;
    private final Random random = new Random();

    @Override
    public void run() {
        while (running && !Thread.currentThread().isInterrupted()) {
            try {
                // Generate products every 2-2.5 seconds
                int delay = 2000 + random.nextInt(500);
                Thread.sleep(delay);

                if (!running) break;

```

```java
                    Product product = new Product();
                    product.setId("PROD-" + System.nanoTime());
                    product.setColor(generateRandomHexColor());

                    queueService.addProductToQueue(targetQueue, product);
                    log.info("Generated Product {} into Queue {}",
                        product.getId(), targetQueue.getId());

                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    break;
                }
            }
        }

    private String generateRandomHexColor() {
        int nextInt = random.nextInt(0xffffff + 1);
        return String.format("#%06x", nextInt);
    }
}
```

Listing 4.5: InputGenerator - Product Generation

Chapter 5

Frontend Implementation

5.1 Angular Service with Signals

Angular 19 introduces Signals for reactive state management. Our `SimulationService` uses signals for state management:

```
@Injectable({ providedIn: 'root' })
export class SimulationService {
    private readonly API_BASE = 'http://localhost:8080/api/simulation';

    // Reactive state using signals
    private _state = signal<SimulationState>({
        queues: [],
        machines: [],
        connections: [],
        isRunning: false,
    });

    private _isConnected = signal<boolean>(false);
    private _isReplaying = signal<boolean>(false);

    // Public readonly signals
    readonly state = this._state.asReadonly();
    readonly isConnected = this._isConnected.asReadonly();

    // Computed values
    readonly queues = computed(() => this._state().queues);
    readonly machines = computed(() => this._state().machines);
    readonly isRunning = computed(() => this._state().isRunning);

    // SSE connection
    private eventSource: EventSource | null = null;

    connectSSE(): void {
        this.eventSource = new EventSource(`${this.API_BASE}/events`);

        this.eventSource.onmessage = (event) => {
            const sseEvent: SSEEvent = JSON.parse(event.data);
            this.handleSSEEvent(sseEvent);
        };
    }

    private handleSSEEvent(event: SSEEvent): void {
        switch (event.type) {
            case 'STATE_UPDATE':
                this._state.set(event.data as SimulationState);
                break;
            case 'QUEUE_EVENT':
                const queueEvent = event.data as QueueEventPayload;
                this._state.update((s) => ({
                    ...s,
                    queues: s.queues.map((q) =>
                        q.id === queueEvent.queueId
                            ? { ...q, productCount: queueEvent.
    newQueueSize }
                            : q
                    ),
                }));
                break;
            case 'SIMULATION_STARTED':
                this._state.update((s) => ({ ...s, isRunning: true }));
                break;
        }
    }
}
```

Listing 5.1: SimulationService - Signal-based State Management

5.2 Main Application Component

```
@Component({
    selector: 'app-root',
    standalone: true,
    imports: [SimulationCanvasComponent, SnapshotListComponent],
    templateUrl: './app.html',
})
export class App implements OnInit, OnDestroy {
    placementMode = signal<PlacementMode>('none');
    connectionSource = signal<SelectedElement | null>(null);

    // Computed total products
    totalProducts = computed(() => {
        const state = this.simulationService.state();
        const queueProducts = state.queues.reduce(
            (sum, q) => sum + q.productCount, 0);
        const machineProducts = state.machines.reduce(
            (sum, m) => sum + m.productCount, 0);
        return queueProducts + machineProducts;
    });

    constructor(private simulationService: SimulationService) {}

    ngOnInit(): void {
        this.simulationService.connectSSE();
        this.simulationService.loadSnapshots();
    }

    onStart(): void {
        this.simulationService.startSimulation().subscribe();
    }

    onCanvasClick(position: { x: number; y: number }): void {
        const mode = this.placementMode();
        if (mode === 'queue') {
            this.simulationService.addQueue(position.x, position.y)
                .subscribe();
        } else if (mode === 'machine') {
            this.simulationService.addMachine(position.x, position.y)
                .subscribe();
        }
    }
}
```

Listing 5.2: App Component

5.3 TypeScript Models

```
export interface Product {
    id: string;
    color: string;
}

export interface Queue {
    id: string;
    x: number;
    y: number;
    productCount: number;
    products: Product[];
}

export interface Machine {
    id: string;
    x: number;
    y: number;
    state: 'idle' | 'processing';
    productCount: number;
    processingTime: number;
    currentProductColor?: string;
}

export interface Connection {
    id: string;
    sourceId: string;
    sourceType: 'queue' | 'machine';
    targetId: string;
    targetType: 'queue' | 'machine';
}

export interface SimulationState {
    queues: Queue[];
    machines: Machine[];
    connections: Connection[];
    isRunning: boolean;
}
```

Listing 5.3: Simulation Models

Chapter 6

User Interface

6.1 Main Interface

The application features a clean, modern interface with the following components:

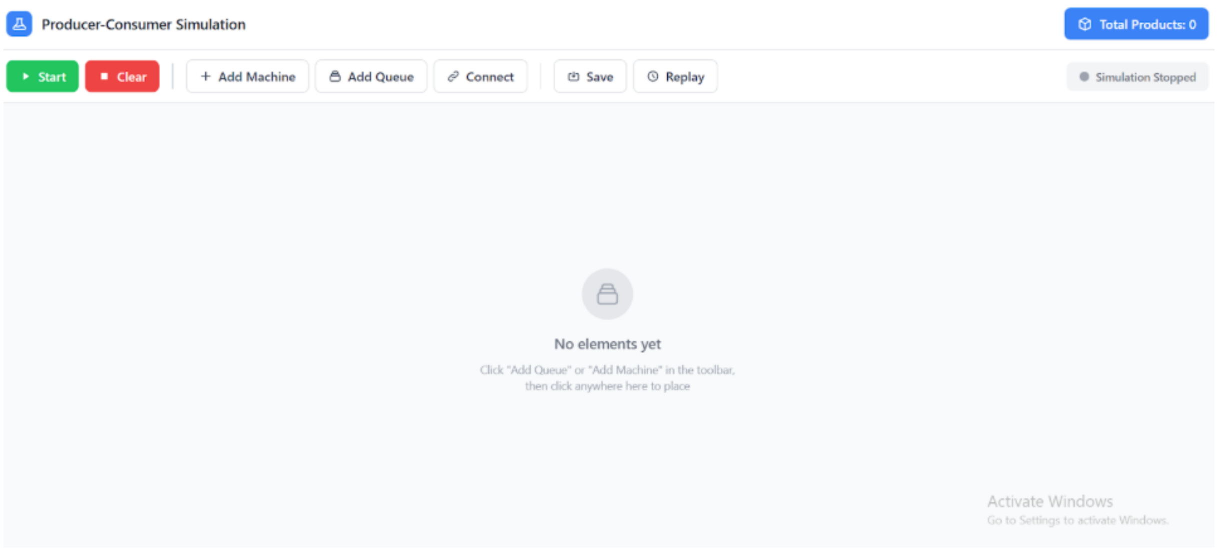

Producer-Consumer Simulation
Total Products: 0
Start
Clear
Add Machine
Add Queue
Connect
Save
Replay
Simulation Stopped
No elements yet
Click "Add Queue" or "Add Machine" in the toolbar, then click anywhere here to place
Activate Windows
Go to Settings to activate Windows.

Figure 6.1: Empty Canvas - Initial State

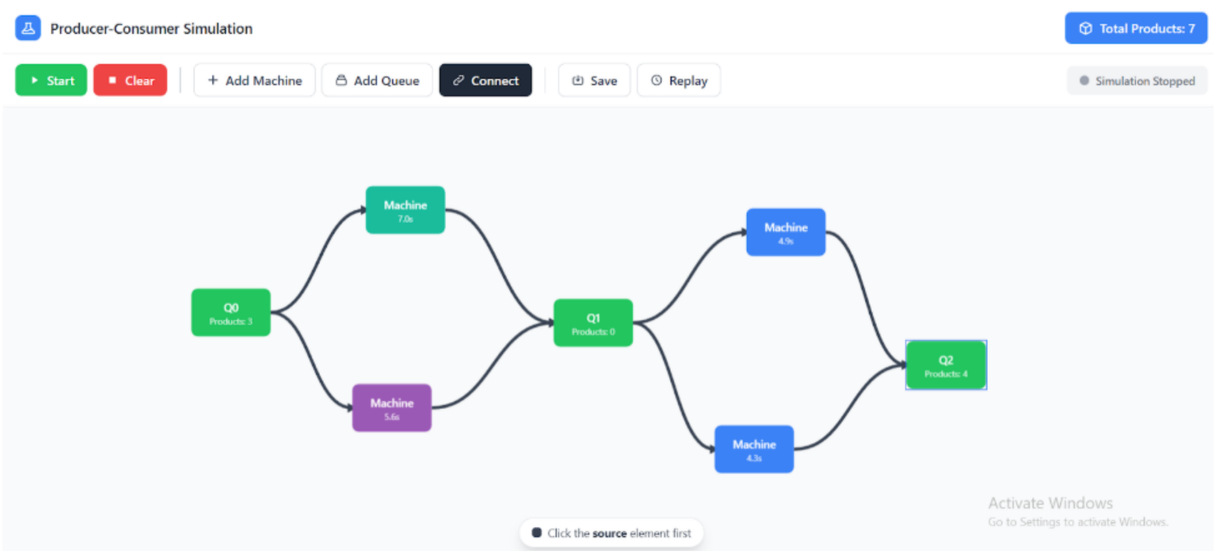


Figure 6.2: Active Simulation with Connected Elements

6.2 UI Components

6.2.1 Header

- Application logo and title
- Real-time product counter (total products in system)

6.2.2 Toolbar

- **Start/Pause**: Control simulation execution
- **Restart**: Reset counts and restart
- **Clear**: Remove all elements
- **Add Machine**: Place new machine on canvas
- **Add Queue**: Place new queue on canvas
- **Connect**: Create connections between elements
- **Save**: Create snapshot of current state
- **Replay**: Load and replay saved snapshots

6.2.3 Canvas

- Visual representation of queues (green boxes showing product count)
- Visual representation of machines (purple boxes showing processing time)
- Connections between elements (curved lines)
- Drag-and-drop repositioning of elements

6.2.4 Status Indicator

- Green pulsing dot: Simulation running
- Gray dot: Simulation stopped
- Purple dot: Replay mode active

Chapter 7

Thread Synchronization

7.1 Concurrency Challenges

The simulation involves multiple concurrent threads:

- Main application thread
- Input generator thread (produces products)
- Machine runner threads (one per machine)
- SSE event streaming

7.2 Synchronization Mechanisms

7.2.1 CopyOnWriteArrayList

Used for the observer list and queue products to ensure thread-safe iteration:

```java
// In QueueService
private final List<QueueObserver> observers =
    new CopyOnWriteArrayList<>();

// In Queue model
public Queue() {
    this.products = new CopyOnWriteArrayList<>();
}
```

Listing 7.1: Thread-safe Collections

7.2.2 ConcurrentHashMap

Used for managing active machine runners:

```java
private final Map<String, Future<?>> activeRunners =
    new ConcurrentHashMap<>();
private final Map<String, MachineRunner> runnerObjects =
    new ConcurrentHashMap<>();
```

7.2.3 Synchronized Methods

Queue operations are synchronized to prevent race conditions:

```java
public synchronized void addProductToQueue(Queue queue, Product product)
    {
    queue.getProducts().add(product);
    queue.setProductCount(queue.getProducts().size());
    notifyObserversProductAdded(queue, product);
}

public synchronized Product removeProductFromQueue(Queue queue) {
    if (queue.getProducts().isEmpty()) return null;
    Product product = queue.getProducts().remove(0);  // FIFO
    queue.setProductCount(queue.getProducts().size());
    notifyObserversProductRemoved(queue, product);
    return product;
}
```

7.2.4 Volatile Variables

Used for thread-safe boolean flags:

```java
private volatile boolean running = true;
```

7.2.5 Wait/Notify (Guarded Suspension)

Used for efficient thread suspension when queues are empty:

```java
public final Object lock = new Object();

// In run() method
synchronized (lock) {
    if (productToProcess == null && running) {
        lock.wait();  // Suspend until notified
    }
}

// In onProductAdded() observer callback
synchronized (lock) {
    lock.notify();  // Wake up waiting thread
}
```

7.3 Thread Pool Management

A cached thread pool is used for machine runners:

```java
@Configuration
public class AppConfig {
    @Bean
    public ExecutorService machineExecutorService() {
        return Executors.newCachedThreadPool();
    }
}
```

Listing 7.2: Thread Pool Configuration

Chapter 8

Conclusion

8.1 Summary

This project successfully implements a Producer-Consumer simulation system that demonstrates:

1. **Design Pattern Application**: Practical use of Observer, Memento, Producer-Consumer, Singleton, and Dependency Injection patterns
2. **Thread Synchronization**: Proper use of Java concurrency utilities including synchronized blocks, ConcurrentHashMap, CopyOnWriteArrayList, wait/notify mechanism, and volatile variables
3. **Modern Web Architecture**: Full-stack implementation with Spring Boot backend and Angular frontend communicating via REST APIs and Server-Sent Events
4. **Real-time Updates**: Live simulation visualization using SSE for instant UI updates
5. **State Management**: Snapshot and replay functionality using the Memento pattern

8.2 Key Learning Outcomes

- Understanding of classic concurrent programming problems
- Practical application of design patterns in a real-world scenario
- Experience with reactive programming using Spring WebFlux and Angular Signals
- Implementation of real-time communication using Server-Sent Events
- Thread-safe programming practices in Java