PRODUCER CONSUMER ASSIGNMENT

TEAMMATES NAMES

MOHAMED MOHAMED IBRAHIM-21011211
MUHAMMAD HASSAN MUHAMMAD-21011115
NADA ALI HASSAN AHMED -19016781
OMAR HANI BISHR-21010891
SARA MUHAMMAD MAHMOUD- 18010770

Section 1 (Downloading & Running the program)

1.1 GitHub Repositories

- Link of GITHUB:
  nada-ali1711/producer_consumer (github.com)

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 the IDE to be run.
  2. Use the normal Run button in your IDE.

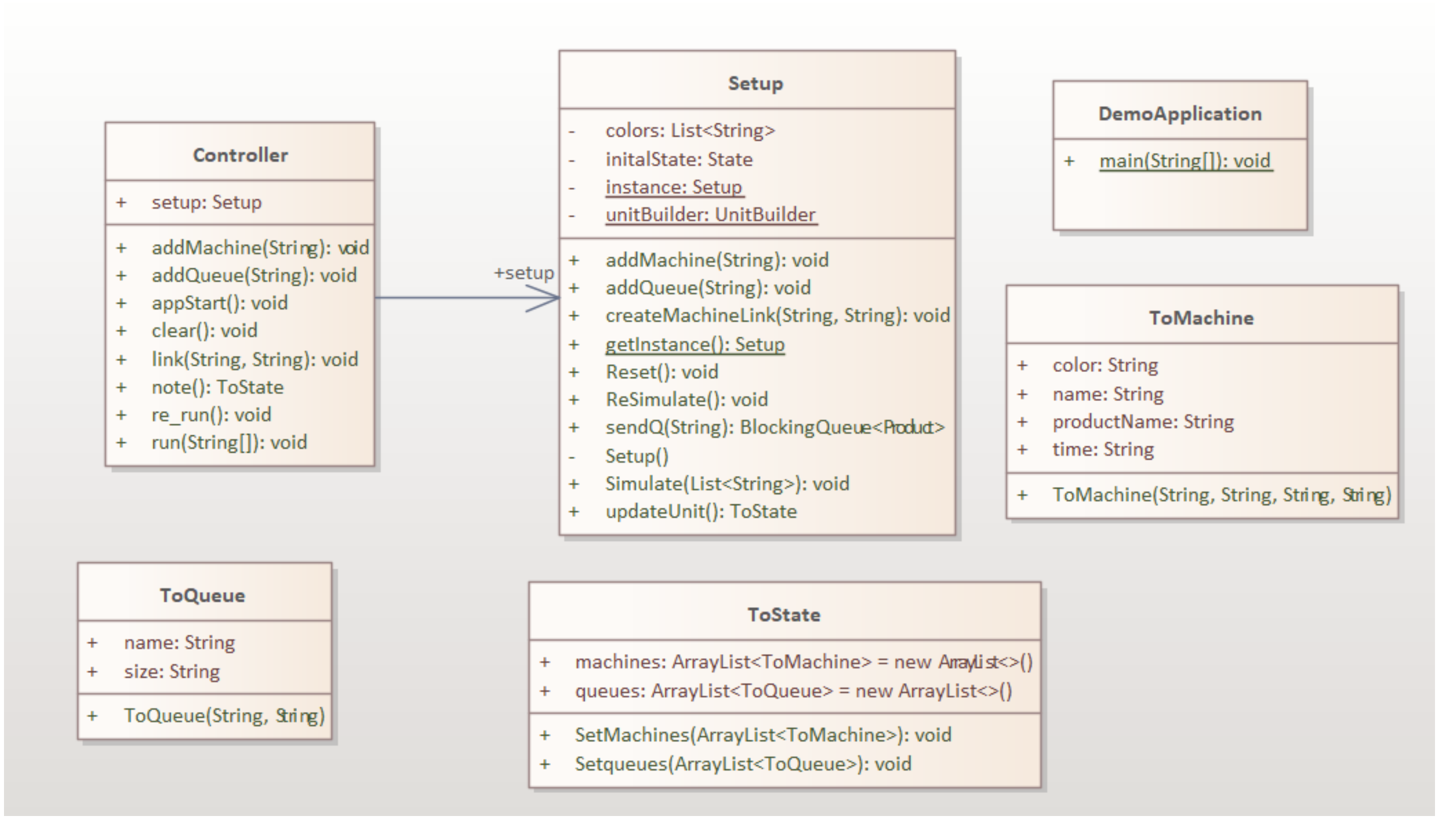
1.4 Instruction to Run Front-End Server

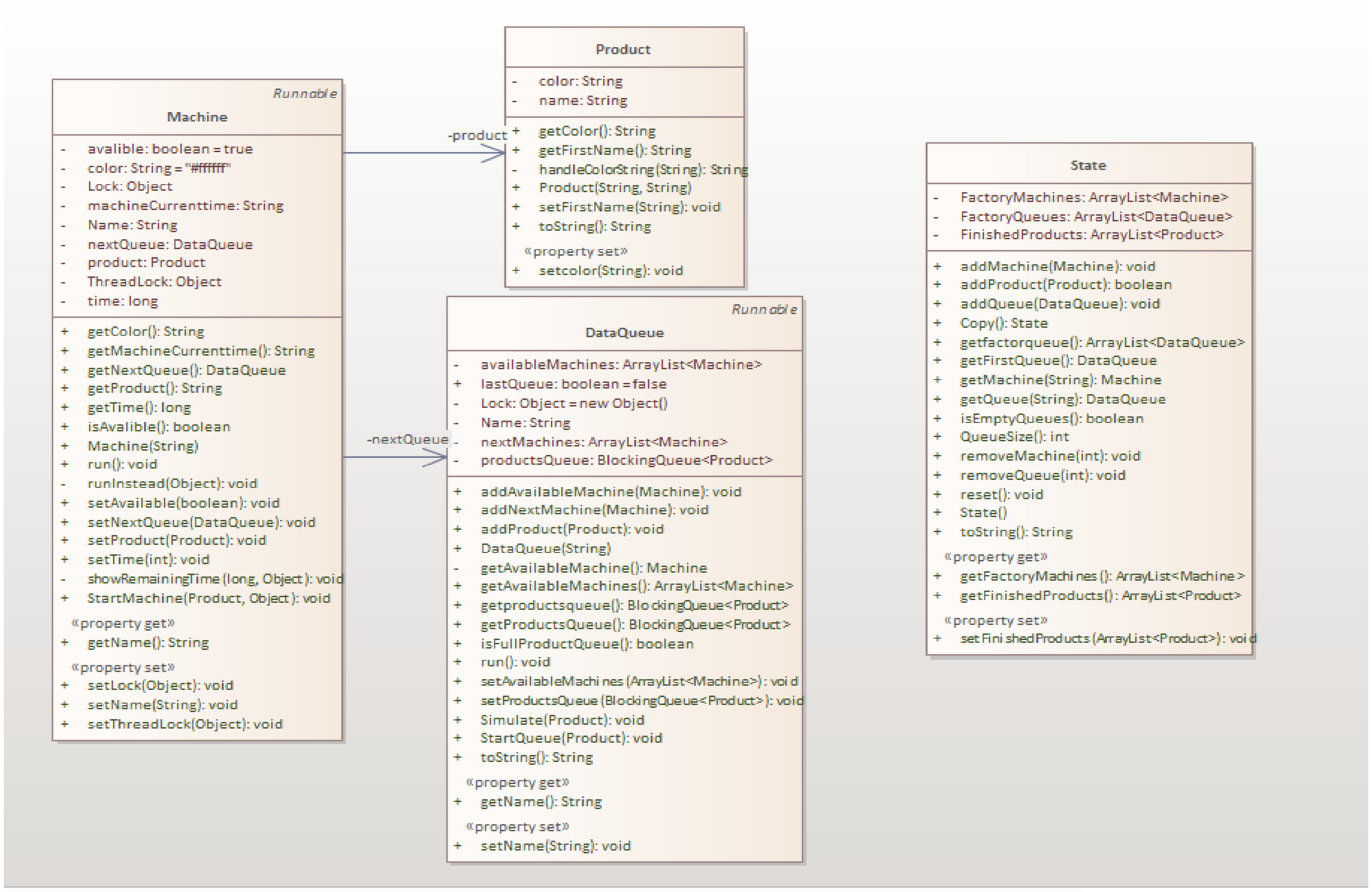
- **Running Front-End codes**
  1. Download and install Node.js from the official website
  2. Open your Command Prompt.

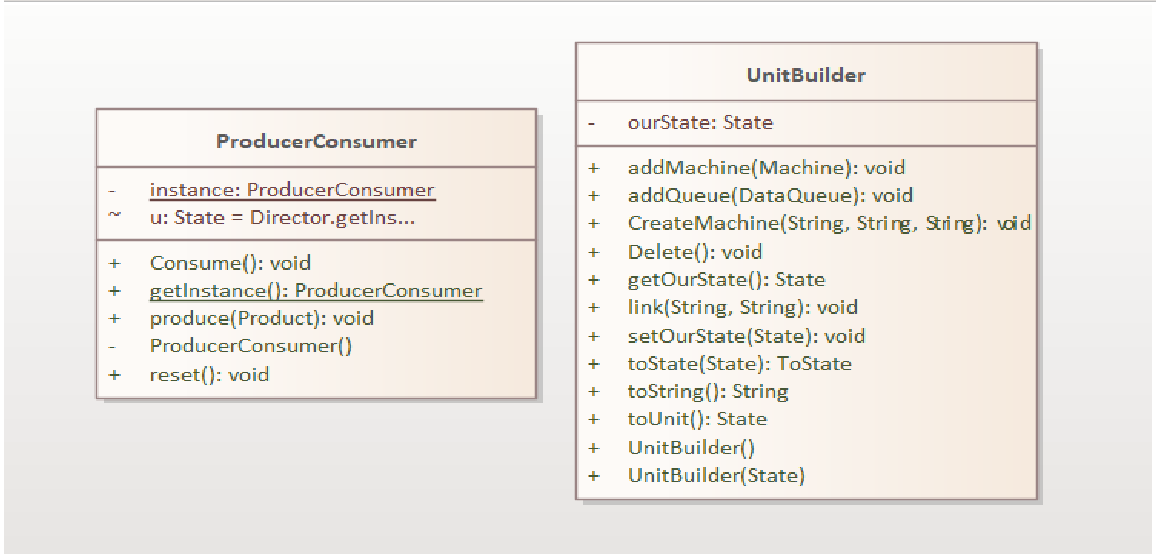
3. In your Command Prompt “npm install -g @vue/cli”.
4. Open Front-End files to IDE to run it.
5. Running Vue server from command prompt or terminal by using “npm run serve”.

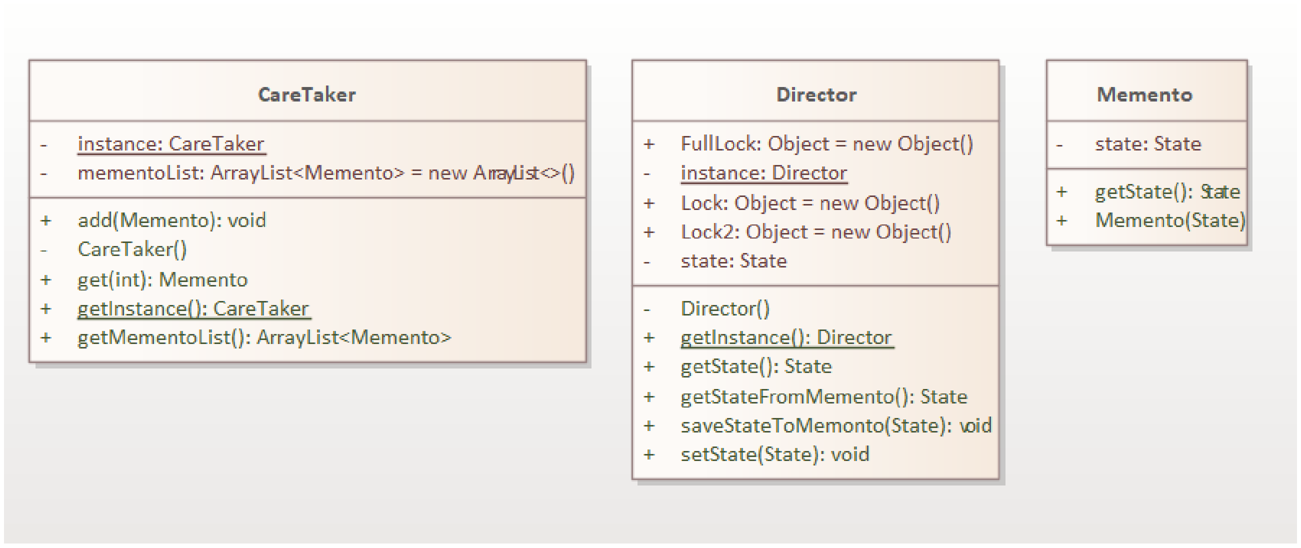
Section 2 (UML Class diagram)

2.1 Class Diagram snippets




Product
- color: String
- name: String
+ getColor(): String
+ getFirstName(): String
- handleColorString(String): String
+ Product(String, String)
+ setFirstName(String): void
+ toString(): String
«property set»
+ setcolor(String): void
Runnable
Machine
- avalible: boolean = true
- color: String = "#ffffff"
- Lock: Object
- machineCurrenttime: String
- Name: String
- nextQueue: DataQueue
- product: Product
- ThreadLock: Object
- time: long
+ getColor(): String
+ getMachineCurrenttime(): String
+ getNextQueue(): DataQueue
+ getProduct(): String
+ getTime(): long
+ isAvalible(): boolean
+ Machine(String)
+ run(): void
- runInstead(Object): void
+ setAvailable(boolean): void
+ setNextQueue(DataQueue): void
+ setProduct(Product): void
+ setTime(int): void
- showRemainingTime(long, Object): void
+ StartMachine(Product, Object): void
«property get»
+ getName(): String
«property set»
+ setLock(Object): void
+ setName(String): void
+ setThreadLock(Object): void
-product
-nextQueue
Runnable
DataQueue
- availableMachines: ArrayList<Machine>
+ lastQueue: boolean = false
- Lock: Object = new Object()
- Name: String
- nextMachines: ArrayList<Machine>
- productsQueue: BlockingQueue<Product>
+ addAvailableMachine(Machine): void
+ addNextMachine(Machine): void
+ addProduct(Product): void
+ DataQueue(String)
- getAvailableMachine(): Machine
+ getAvailableMachines(): ArrayList<Machine>
+ getproductsqueue(): BlockingQueue<Product>
+ getProductsQueue(): BlockingQueue<Product>
+ isFullProductQueue(): boolean
+ run(): void
+ setAvailableMachines(ArrayList<Machine>): void
+ setProductsQueue(BlockingQueue<Product>): void
+ Simulate(Product): void
+ StartQueue(Product): void
+ toString(): String
«property get»
+ getName(): String
«property set»
+ setName(String): void
State
- FactoryMachines: ArrayList<Machine>
- FactoryQueues: ArrayList<DataQueue>
- FinishedProducts: ArrayList<Product>
+ addMachine(Machine): void
+ addProduct(Product): boolean
+ addQueue(DataQueue): void
+ Copy(): State
+ getfactorqueue(): ArrayList<DataQueue>
+ getFirstQueue(): DataQueue
+ getMachine(String): Machine
+ getQueue(String): DataQueue
+ isEmptyQueues(): boolean
+ QueueSize(): int
+ removeMachine(int): void
+ removeQueue(int): void
+ reset(): void
+ State()
+ toString(): String
«property get»
+ getFactoryMachines(): ArrayList<Machine>
+ getFinishedProducts(): ArrayList<Product>
«property set»
+ setFinishedProducts(ArrayList<Product>): void


ProducerConsumer
- instance: ProducerConsumer
~ u: State = Director.getIns...
+ Consume(): void
+ getInstance(): ProducerConsumer
+ produce(Product): void
- ProducerConsumer()
+ reset(): void
UnitBuilder
- ourState: State
+ addMachine(Machine): void
+ addQueue(DataQueue): void
+ CreateMachine(String, String, String): void
+ Delete(): void
+ getOurState(): State
+ link(String, String): void
+ setOurState(State): void
+ toState(State): ToState
+ toString(): String
+ toUnit(): State
+ UnitBuilder()
+ UnitBuilder(State)


CareTaker
- instance: CareTaker
- mementoList: ArrayList<Memento> = new ArrayList<>()
+ add(Memento): void
- CareTaker()
+ get(int): Memento
+ getInstance(): CareTaker
+ getMementoList(): ArrayList<Memento>
Director
+ FullLock: Object = new Object()
- instance: Director
+ Lock: Object = new Object()
+ Lock2: Object = new Object()
- state: State
- Director()
+ getInstance(): Director
+ getState(): State
+ getStateFromMemento(): State
+ saveStateToMemonto(State): void
+ setState(State): void
Memento
- state: State
+ getState(): State
+ Memento(State)

Section 3 (Design Patterns)

3.1 Snapshot Pattern:

In our project, the Snapshot pattern is employed to capture and restore the state of objects. This pattern allows us to take snapshots of object states and later restore them, which can be useful for scenarios where we need to save and restore the state of a system or its components.

3.2- Facade Pattern:

The Facade pattern is utilized to provide a simplified interface to a complex subsystem in our project. By using a facade, we create a unified interface that hides the complexities of the subsystem from the client code, thus making it easier to use and understand.

3.3- Singleton Pattern:

The Singleton pattern is used to ensure that certain classes have only a single instance in our project. This pattern provides a global point of access to that instance, which can be beneficial for managing resources that should be shared across the application.

3.4 Producer-Consumer Pattern:

The Producer-Consumer pattern is applied to coordinate the work of multiple producers and consumers that share a common resource in our project. This pattern helps in managing the concurrent interactions between producers and consumers, .ensuring that the resource is accessed in a synchronized manner

3.5 Guarded Suspension Pattern:

The Guarded Suspension pattern is used to manage interactions between producers and consumers based on specific conditions in our project. This pattern allows a consumer to wait for a condition to be met before consuming a product produced by a producer, ensuring that the consumption happens under the right circumstances.

3.6 Observer Pattern:

Additionally, the Observer pattern is employed in our project to establish a one-to-many dependency between objects. This pattern allows multiple observers (or listeners) to be notified and updated automatically when the state of a subject (or publisher) changes. It is useful for implementing distributed event handling systems, where changes in one part of the system need to be reflected in other parts.

Section 4 (UI & User Guide)

4.1 User Guide

- I Recommend for you to have a look on 5-min demo video on this link
- https://drive.google.com/file/d/1epcko_lISkC5qO4mziwNRHBVXdKPDuMV/view?usp=sharing

Section 5 Decisions

- we didn't use the socket method, instead, we continuously called the backend from the frontend to get the updates from the backend.

- setup was the middle layer between the controller and the code implementation
- products will start on Q0 and not any other Q
- user cannot join 2 machines or 2 queues with each other
- to start another simulation, the back server must be restarted, and webpage refreshed
- it shouldn’t make self-loop.