CS 431 Programming Languages Lab Assignment 1 (Concurrent Programming)

160101039 Kapil Goyal

160101072 Sparsh Bansal

Problem 1: Merchandise sale for Alcheringa 2020

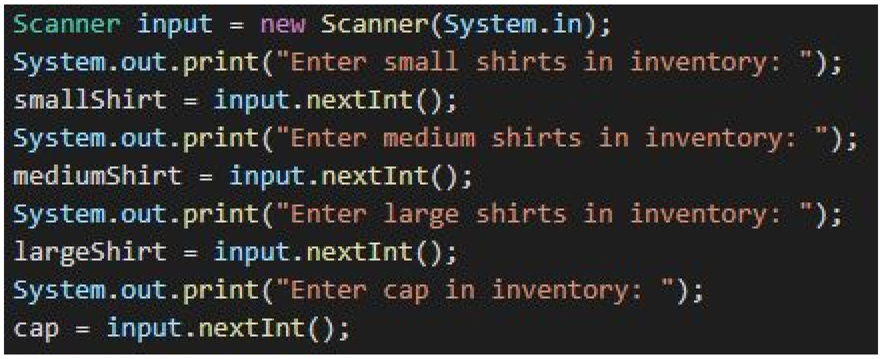
Classes used:

- **Order:** Stores the details such as orderID, quantity, status and type about an order.
- **Inventory:** Stores and updates the quantities of different commodities.
- **Main:** Takes input and calls different methods to process orders

Q. Specify the synchronization technique that you used in the program. Explain your implementation using code snippets.

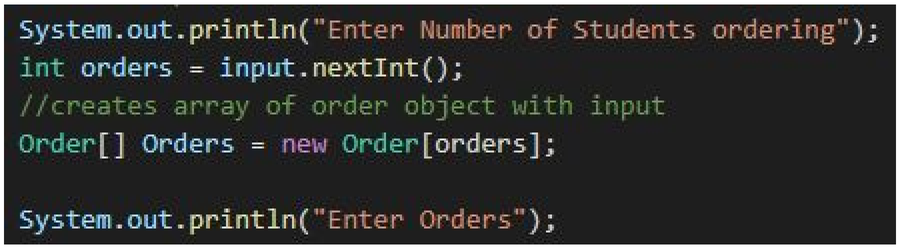
Implementation:

- First, we take input from the user for the number of items of each product in the inventory.



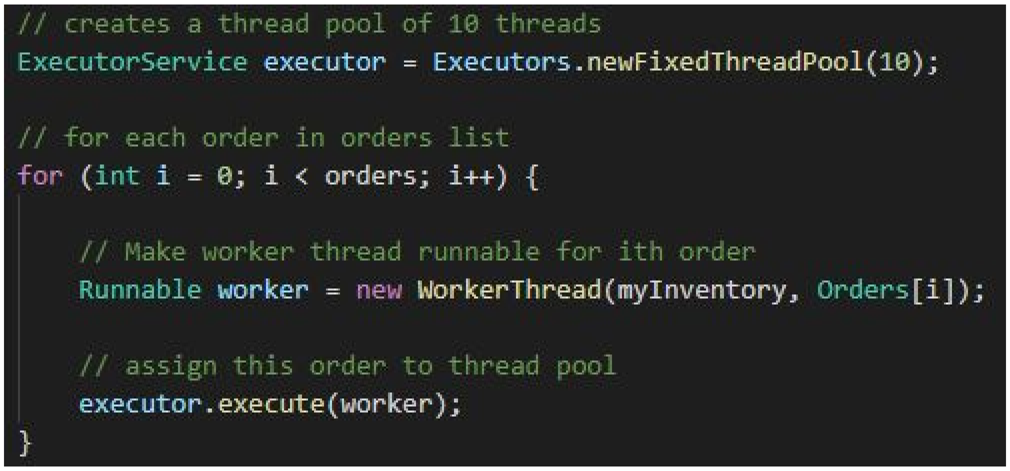
```
Scanner input = new Scanner(System.in);
System.out.print("Enter small shirts in inventory: ");
smallShirt = input.nextInt();
System.out.print("Enter medium shirts in inventory: ");
mediumShirt = input.nextInt();
System.out.print("Enter large shirts in inventory: ");
largeShirt = input.nextInt();
System.out.print("Enter cap in inventory: ");
cap = input.nextInt();
```

- Then the user is asked for the number of orders and the details of each order and input are taken.



```
System.out.println("Enter Number of Students ordering");
int orders = input.nextInt();
//creates array of order object with input
Order[] Orders = new Order[orders];

System.out.println("Enter Orders");
```

- Once we get all the orders, we process a batch of orders concurrently using a thread pool.
- To process orders in a batch concurrently, we have made a thread pool of 10 threads. This thread pool will assign orders to these threads which will run concurrently. Once an order is completed by a thread, the thread will pick up another order and start executing that order.



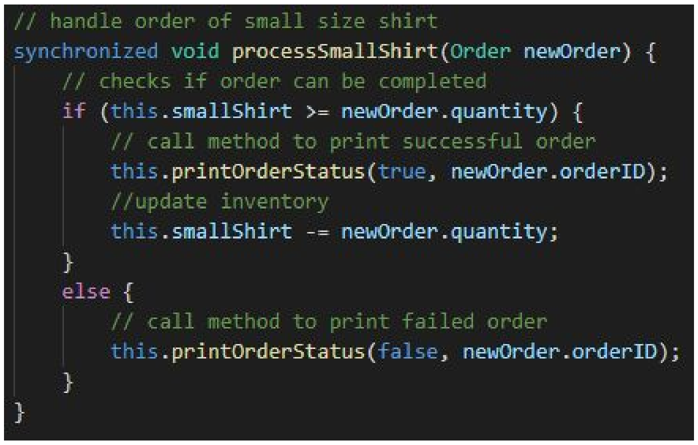
```
// creates a thread pool of 10 threads
ExecutorService executor = Executors.newFixedThreadPool(10);

// for each order in orders list
for (int i = 0; i < orders; i++) {

    // Make worker thread runnable for ith order
    Runnable worker = new WorkerThread(myInventory, Orders[i]);

    // assign this order to thread pool
    executor.execute(worker);
}
```

- Now, each thread will process its order while maintaining proper synchronization and print its result such that output for two orders is not interleaved.

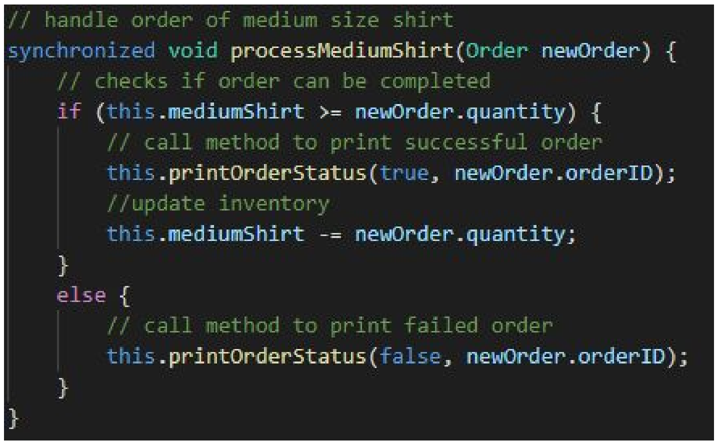
Synchronization:

For synchronization, we are using synchronized methods. Synchronized methods provide an easy solution for preventing thread interferences and memory inconsistencies. It is not possible for two invocations of synchronized methods to interleave. If one thread is executing a synchronized method, other thread will have to wait to access the same method. Synchronization is built around an internal entity known as the intrinsic lock or monitor lock. When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

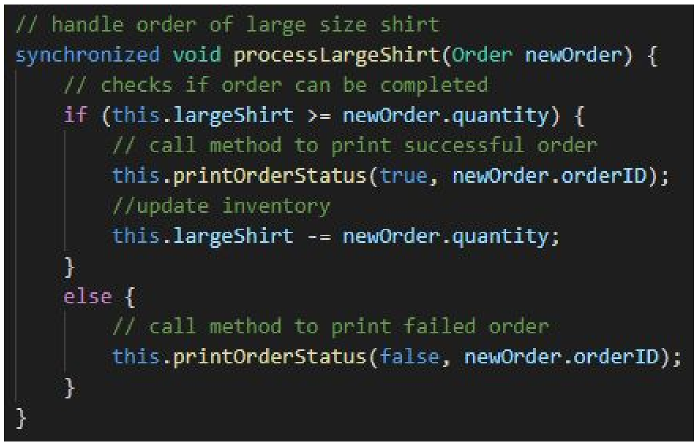
- Since we can have four different types of orders (i.e, Cap, Small Shirt, Medium Shirt, Large Shirt), we can process these four items of inventory in parallel. Hence we will use four different synchronized methods for each of the different items so that multiple orders of the item do not interleave.



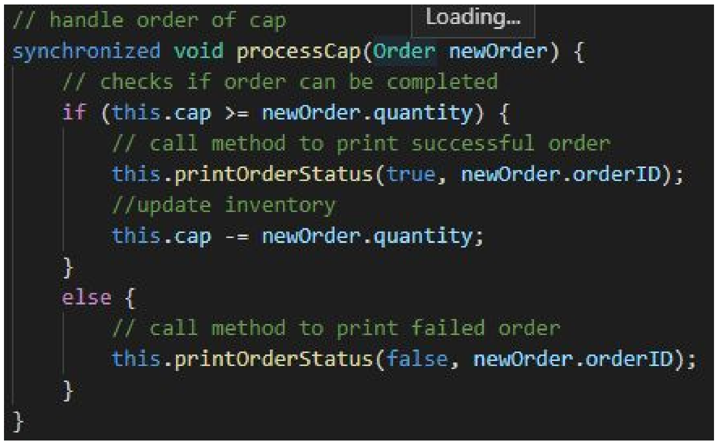
```java
// handle order of small size shirt
synchronized void processSmallShirt(Order newOrder) {
    // checks if order can be completed
    if (this.smallShirt >= newOrder.quantity) {
        // call method to print successful order
        this.printOrderStatus(true, newOrder.orderID);
        //update inventory
        this.smallShirt -= newOrder.quantity;
    }
    else {
        // call method to print failed order
        this.printOrderStatus(false, newOrder.orderID);
    }
}
```



```java
// handle order of medium size shirt
synchronized void processMediumShirt(Order newOrder) {
    // checks if order can be completed
    if (this.mediumShirt >= newOrder.quantity) {
        // call method to print successful order
        this.printOrderStatus(true, newOrder.orderID);
        //update inventory
        this.mediumShirt -= newOrder.quantity;
    }
    else {
        // call method to print failed order
        this.printOrderStatus(false, newOrder.orderID);
    }
}
```

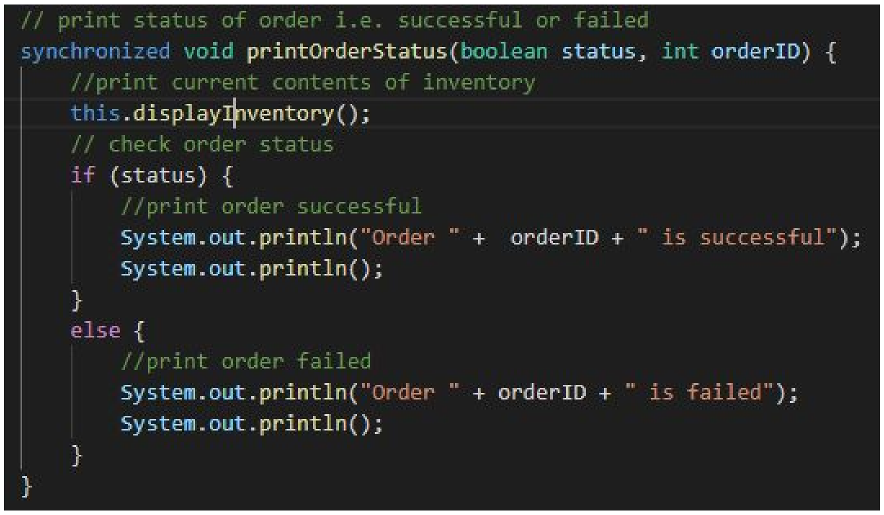


```java
// handle order of large size shirt
synchronized void processLargeShirt(Order newOrder) {
    // checks if order can be completed
    if (this.largeShirt >= newOrder.quantity) {
        // call method to print successful order
        this.printOrderStatus(true, newOrder.orderID);
        //update inventory
        this.largeShirt -= newOrder.quantity;
    }
    else {
        // call method to print failed order
        this.printOrderStatus(false, newOrder.orderID);
    }
}
```



```java
// handle order of cap
synchronized void processCap(Order newOrder) {
    // checks if order can be completed
    if (this.cap >= newOrder.quantity) {
        // call method to print successful order
        this.printOrderStatus(true, newOrder.orderID);
        //update inventory
        this.cap -= newOrder.quantity;
    }
    else {
        // call method to print failed order
        this.printOrderStatus(false, newOrder.orderID);
    }
}
```

- We also need to ensure that if an order is processed, terminal output for two orders does not show inconsistency. So we use another synchronized method to print order status.



```java
// print status of order i.e. successful or failed
synchronized void printOrderStatus(boolean status, int orderID) {
    //print current contents of inventory
    this.displayInventory();
    // check order status
    if (status) {
        //print order successful
        System.out.println("Order " +  orderID + " is successful");
        System.out.println();
    }
    else {
        //print order failed
        System.out.println("Order " + orderID + " is failed");
        System.out.println();
    }
}
```

Problem 2: Cold Drink Manufacturing

Classes used:

- **Bottle:** Stores the type & state and updates state of a bottle.
- **Packaging:** This class simulates the packaging unit. It has 3 states (i.e Empty, Packaging and Waiting). Empty state means that the packaging unit is empty and is waiting for a bottle. Packaging means that the unit is packaging a bottle. Waiting means that it has finished packaging and is attempting to put the bottle in the sealing buffer. It has a method that acts as a controller to process and switching of states.
- **Sealing:** This class simulates the sealing unit. Like Packaging class it also has 3 states (i.e. Empty, Sealing and Waiting) and a controller function that controls switching of the states.
- **Buffer:** This class stores a queue of bottles. This class is used to simulate each kind of buffer that is mentioned in the question (i.e. unfinished tray, packing buffer, sealing buffer and godown)
- **UnfinishedTray:** This class is used to simulate the behaviour of the unfinished tray as mentioned the question.
- **Main:** This class takes input, creates threads for packaging & sealing and simulate the whole system.

Q.1 Discuss the importance of concurrent programming here.

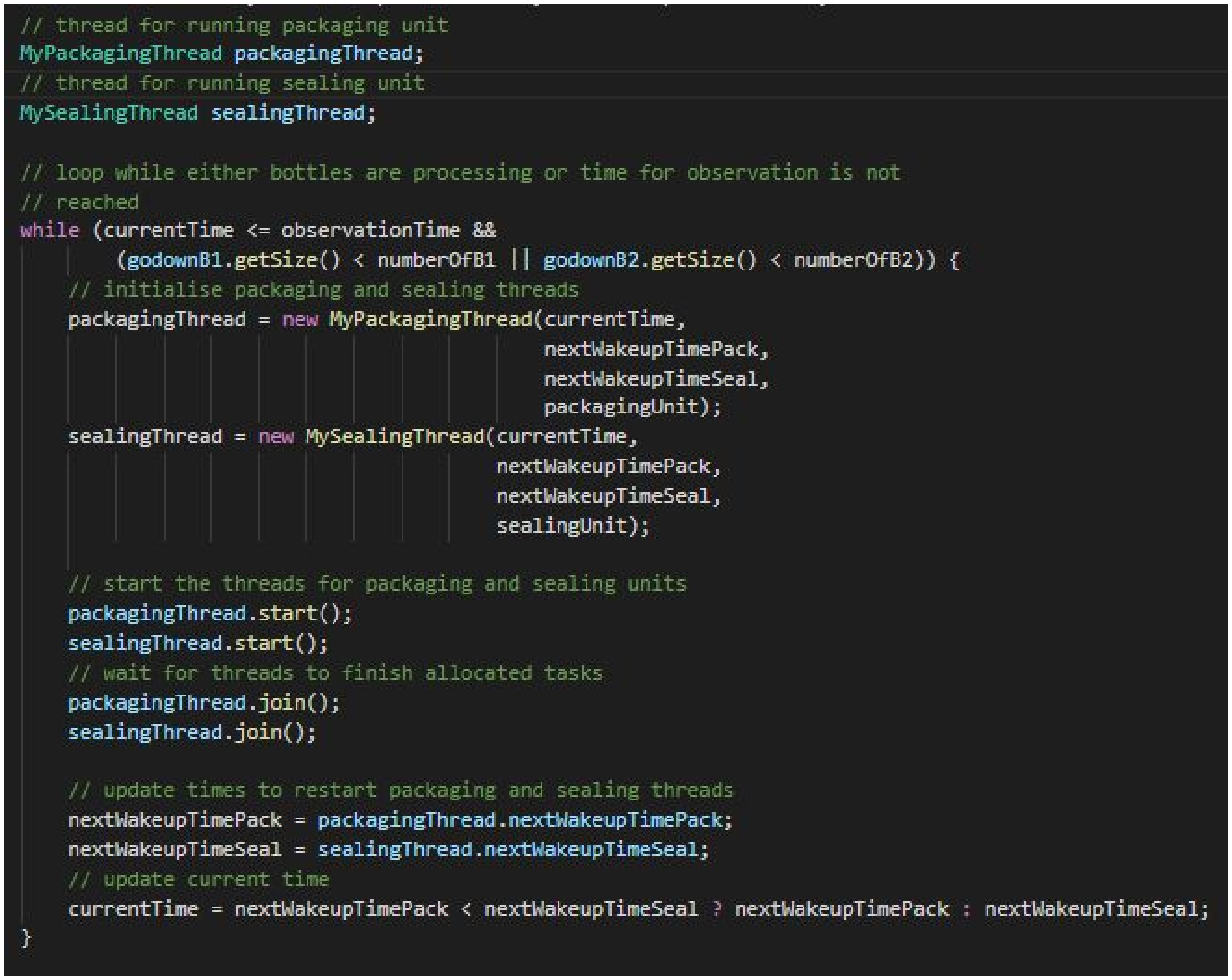
According to the problem, a factory needs to be simulated which consists of two subunits i.e. packaging and sealing. The packaging and sealing units work independently and simultaneously. The order in which processing can occur between the above two units is random. Using concurrent programming, the execution time of the application improves substantially as the packaging and sealing units can be simulated at the same time. The suspension, waiting and resuming of a unit (due to full buffers) can be done independently.

Q.2 Using code snippets describe how you used the concurrency and synchronization in your program.

Concurrency

- To do the two operations concurrently, two threads are running at a time which handles the packaging and sealing of bottles respectively. These two threads represent the packaging and the sealing unit. We are associating a variable named **nextWakeUpTime** to each unit which is used to simulate the state switching in the units. When **currentTime** becomes equal to the **nextWakeUpTime** for any unit, that unit starts processing and switch its state (if possible). The threads will keep running until

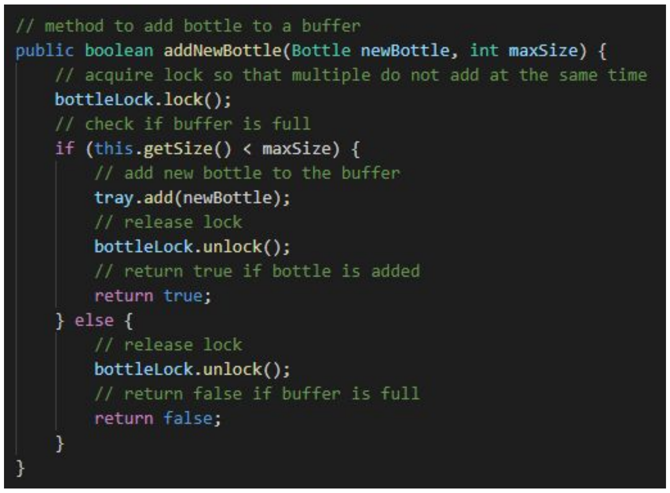
currentTime does not become equal to observation time and at least one buffer tray is not empty. Once the threads stop, bottles' status' are computed and output is shown in the terminal.



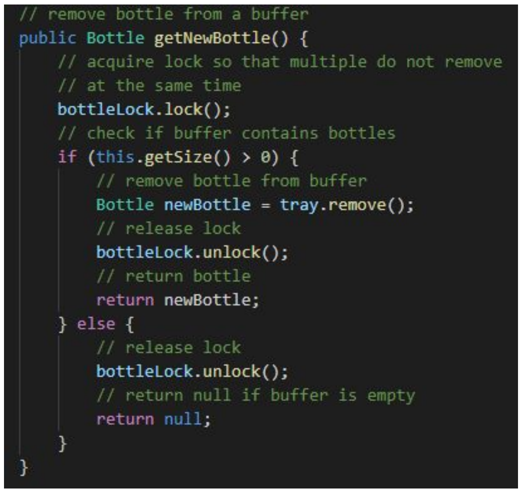
```
// thread for running packaging unit
MyPackagingThread packagingThread;
// thread for running sealing unit
MySealingThread sealingThread;

// loop while either bottles are processing or time for observation is not
// reached
while (currentTime <= observationTime &&
        (godownB1.getSize() < numberOfB1 || godownB2.getSize() < numberOfB2)) {
    // initialise packaging and sealing threads
    packagingThread = new MyPackagingThread(currentTime,
                                            nextWakeupTimePack,
                                            nextWakeupTimeSeal,
                                            packagingUnit);
    sealingThread = new MySealingThread(currentTime,
                                        nextWakeupTimePack,
                                        nextWakeupTimeSeal,
                                        sealingUnit);

    // start the threads for packaging and sealing units
    packagingThread.start();
    sealingThread.start();
    // wait for threads to finish allocated tasks
    packagingThread.join();
    sealingThread.join();

    // update times to restart packaging and sealing threads
    nextWakeupTimePack = packagingThread.nextWakeupTimePack;
    nextWakeupTimeSeal = sealingThread.nextWakeupTimeSeal;
    // update current time
    currentTime = nextWakeupTimePack < nextWakeupTimeSeal ? nextWakeupTimePack : nextWakeupTimeSeal;
}
```

Synchronization:

Since we are using **Buffer** class to implement each of the buffer (i.e. unfinished tray, packing buffer, sealing buffer and godown), we just need to maintain synchronization for any object of this class and all buffers will be synchronized. In this class, we are using an object-level lock to synchronize addition and removal of bottle from the buffer as shown in the snippets. If any unit tries to add or remove a bottle from a common buffer (say B1 packaging buffer), it will hold its lock until the bottle gets added or removed. The other unit won't be able to add or remove until the previous unit does not finish its task. Hence, this way buffers are synchronized.



```java
// method to add bottle to a buffer
public boolean addNewBottle(Bottle newBottle, int maxSize) {
    // acquire lock so that multiple do not add at the same time
    bottleLock.lock();
    // check if buffer is full
    if (this.getSize() < maxSize) {
        // add new bottle to the buffer
        tray.add(newBottle);
        // release lock
        bottleLock.unlock();
        // return true if bottle is added
        return true;
    } else {
        // release lock
        bottleLock.unlock();
        // return false if buffer is full
        return false;
    }
}
```



```java
// remove bottle from a buffer
public Bottle getNewBottle() {
    // acquire lock so that multiple do not remove
    // at the same time
    bottleLock.lock();
    // check if buffer contains bottles
    if (this.getSize() > 0) {
        // remove bottle from buffer
        Bottle newBottle = tray.remove();
        // release lock
        bottleLock.unlock();
        // return bottle
        return newBottle;
    } else {
        // release lock
        bottleLock.unlock();
        // return null if buffer is empty
        return null;
    }
}
```

Q3. How would the program be affected if synchronization is not used? Discuss.

If synchronisation is not used while implementing concurrent programming in the problem, certain race conditions can occur which lead to consistency in the data structures (e.g. sealing buffer etc) shared by different units. Let us take the case when there is only one bottle in the unfinished tray and both the units concurrently access the unfinished tray. They see that they can process a bottle individually and both of them start processing the bottle. But only one unit should start processing the bottle. Hence to avoid such race conditions and data inconsistency, there should be synchronization management for the buffers and each unit should be able to exclusively update buffers.

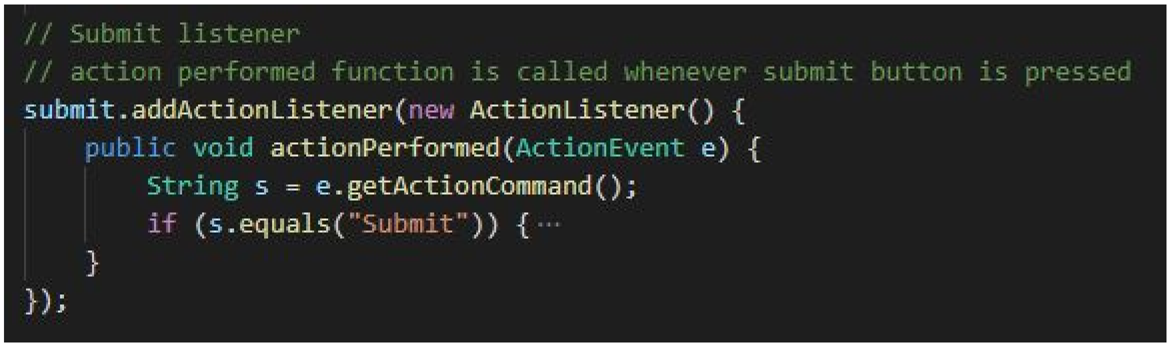
Problem 3: Automatic Traffic Light System

Classes used:

- **Vehicle:** Stores information like source, destination, id, status and remaining time about a vehicle.
- **VehicleQueue**: Creates and updates the queue of vehicles for each traffic lane.
- **TrafficLights**: Handles the change in traffic lights and the corresponding changes in UI and vehicles.
- **Main**: Handles the dynamic input procedures and creates UI elements.

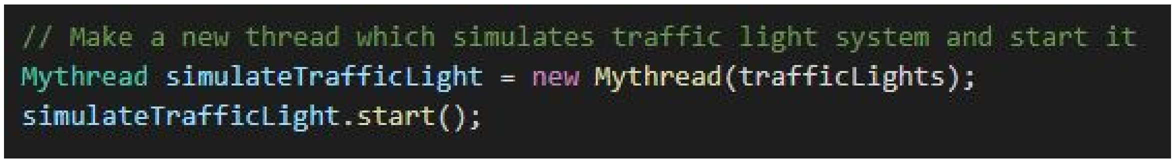
Q1. Do you think the concept of concurrency is applicable while implementing the program? If yes, why? Explain using code snippets

We have implemented our program in such a way that the user can give inputs dynamically i.e. during runtime as well. So one thread needs to keep checking whether there is any new input given by the user and one thread that keeps updating the UI after every second. That's why we need concurrency to simulate this behaviour and obtain real-time simulation with faster execution.

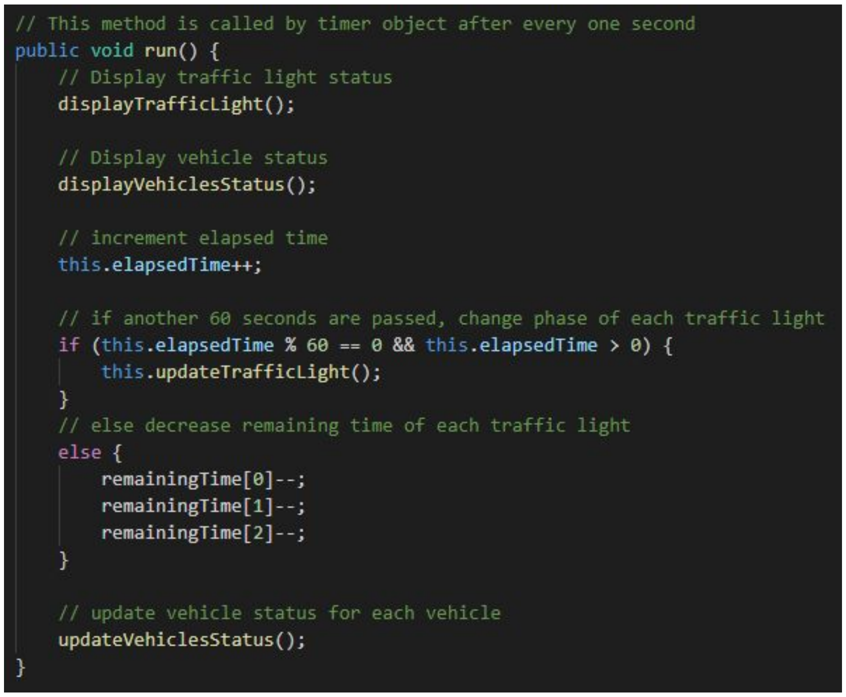


```java
// Submit listener
// action performed function is called whenever submit button is pressed
submit.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String s = e.getActionCommand();
        if (s.equals("Submit")) {···
    }
});
```

The first code block shows submit event listener which performs the action whenever the submit button is pressed and it works in one thread. The two code blocks that are shown below implements another thread which keeps updating the UI elements after every second. Thus using these two threads concurrently, we simulate the traffic light behaviour. There is implicit threading occurring in Swing GUIs. This threading (and concurrency) is also important because GUIs need to be responsive quickly (important for good user interaction)



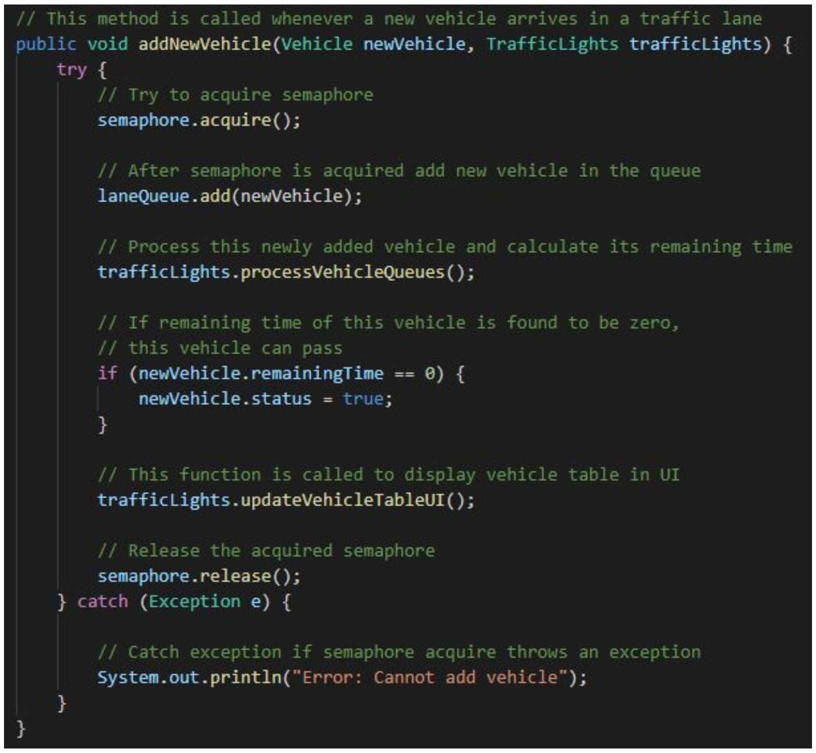
```java
// Make a new thread which simulates traffic light system and start it
Mythread simulateTrafficLight = new Mythread(trafficLights);
simulateTrafficLight.start();
```



```java
// This method is called by timer object after every one second
public void run() {
    // Display traffic light status
    displayTrafficLight();

    // Display vehicle status
    displayVehiclesStatus();

    // increment elapsed time
    this.elapsedTime++;

    // if another 60 seconds are passed, change phase of each traffic light
    if (this.elapsedTime % 60 == 0 && this.elapsedTime > 0) {
        this.updateTrafficLight();
    }
    // else decrease remaining time of each traffic light
    else {
        remainingTime[0]--;
        remainingTime[1]--;
        remainingTime[2]--;
    }

    // update vehicle status for each vehicle
    updateVehiclesStatus();
}
```

Q2. Did you use synchronization in the problem? If yes, explain with code snippets

Yes, we used synchronization in the problem. Since we are making two threads one of which simulates traffic light and the other one takes input, we need to make sure that those two threads do not access the vehicle queues (for T1/T2/T3) simultaneously. Otherwise, there would be data inconsistency and simulation will produce wrong results. Here we are using an object-level semaphore that is initialized with the number of permits equal to 1. So each traffic lane queue can only be held by either of the two threads. Whenever a new vehicle is added by the main thread in the vehicle queue, it acquires semaphore for that queue and releases it when vehicle is added.



```java
// This method is called whenever a new vehicle arrives in a traffic lane
public void addNewVehicle(Vehicle newVehicle, TrafficLights trafficLights) {
    try {
        // Try to acquire semaphore
        semaphore.acquire();

        // After semaphore is acquired add new vehicle in the queue
        laneQueue.add(newVehicle);

        // Process this newly added vehicle and calculate its remaining time
        trafficLights.processVehicleQueues();

        // If remaining time of this vehicle is found to be zero,
        // this vehicle can pass
        if (newVehicle.remainingTime == 0) {
            newVehicle.status = true;
        }

        // This function is called to display vehicle table in UI
        trafficLights.updateVehicleTableUI();

        // Release the acquired semaphore
        semaphore.release();
    } catch (Exception e) {

        // Catch exception if semaphore acquire throws an exception
        System.out.println("Error: Cannot add vehicle");
    }
}
```

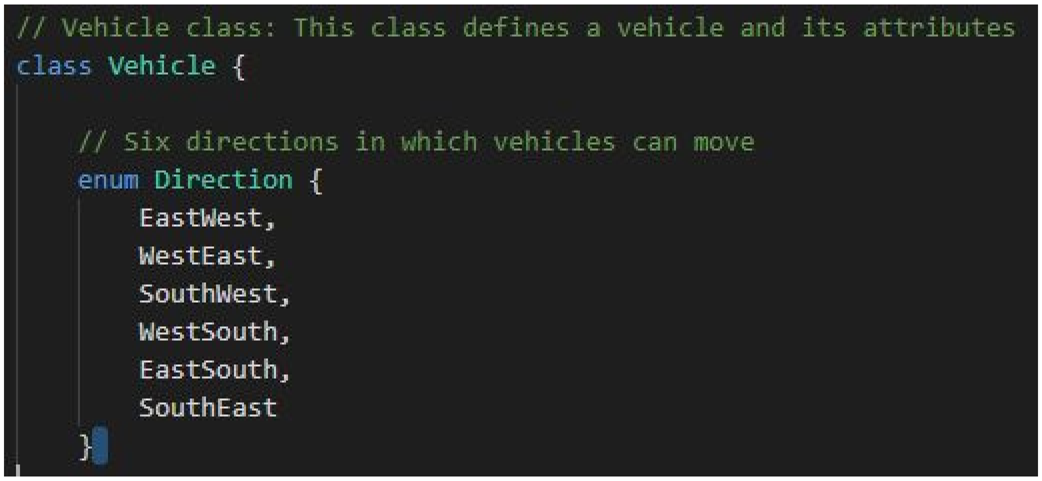
Q3. If instead of T-junction, it would have been a four-way junction (i.e. another way towards old SAC), will there be any change in the problem implementation? If yes, write the pseudocode

Assumptions:

- If a vehicle is moving from its current lane to adjacent lane (from S to W or W to N or N to E or E to S), it can pass freely.
- There is a common queue for vehicles travelling in forward direction or vehicles travelling in the right direction.

- The time for a vehicle to pass is the same i.e. 6 seconds.
- There is a common traffic light to pass from a source to all possible destinations(except adjacent lane).

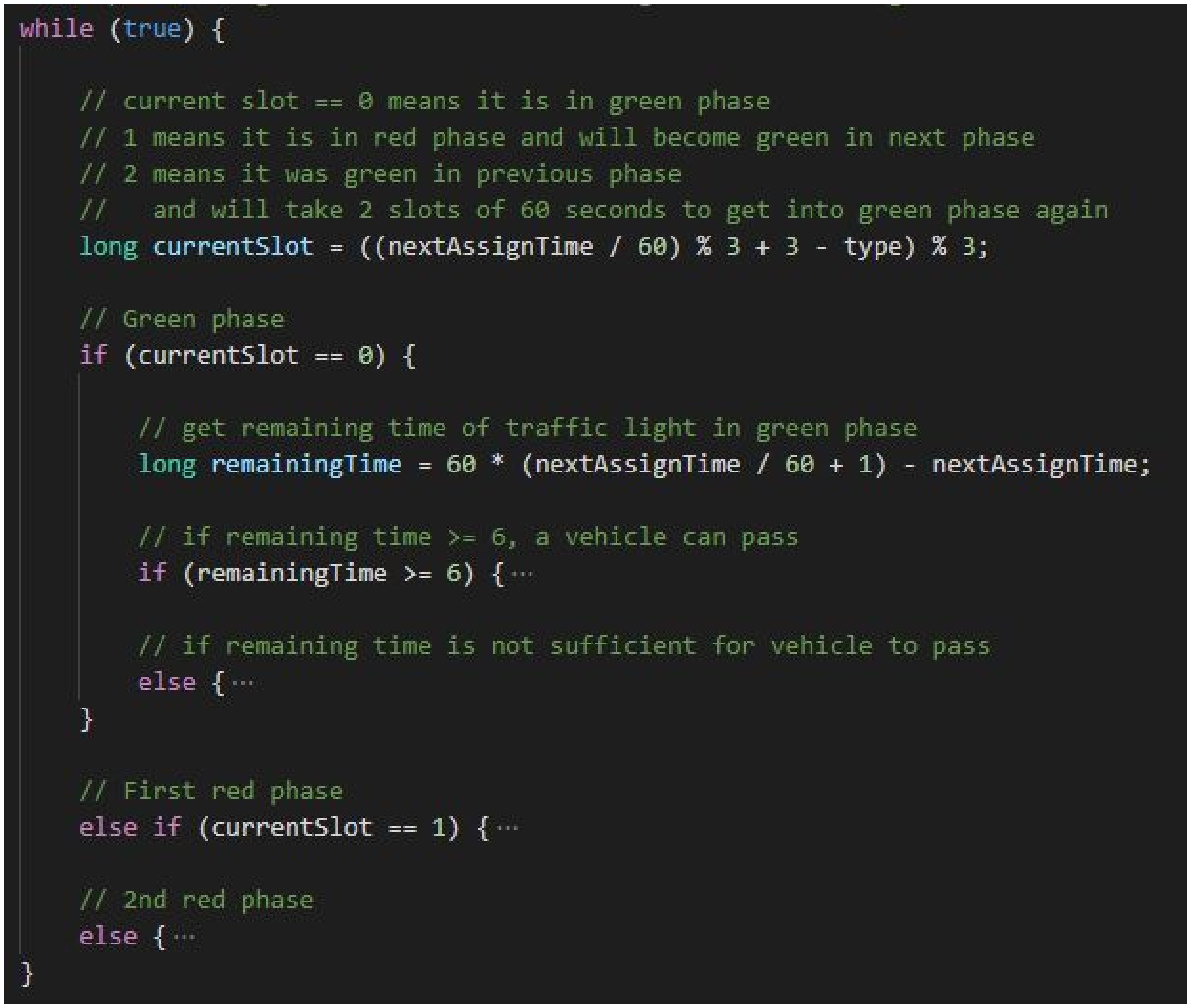
Changes In code: Our main idea to calculate the remaining time for each traffic light and vehicle will remain the same. However, since a new traffic light and lane is added, we need to store some extra information to implement this new change. The formula to calculate the remaining time for a vehicle was based on the mod 3 calculation, but now we will use mod 4 calculation. We are presenting here the code snippets where new changes will be added and the pseudo-code for those changes.



```
// Vehicle class: This class defines a vehicle and its attributes
class Vehicle {

    // Six directions in which vehicles can move
    enum Direction {
        EastWest,
        WestEast,
        SouthWest,
        WestSouth,
        EastSouth,
        SouthEast
    }
```

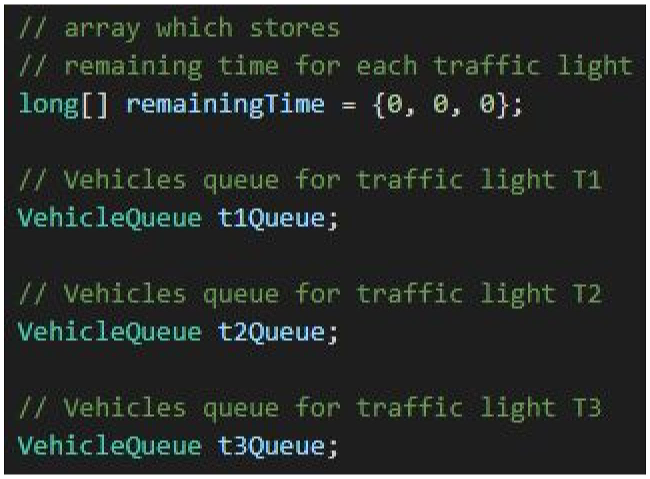
In the vehicle class, the vehicles now have 12 possible source-destination combinations.
Pseudocode:

```
enum Direction {
    EastWest,
    WestEast,
    SouthWest,
    WestSouth,
    EastSouth,
    SouthEast,
    NorthWest,
    NorthEast,
    SouthNorth,
    WestNorth,
    EastNorth,
    NorthSouth
  }
```


```
while (true) {

    // current slot == 0 means it is in green phase
    // 1 means it is in red phase and will become green in next phase
    // 2 means it was green in previous phase
    //   and will take 2 slots of 60 seconds to get into green phase again
    long currentSlot = ((nextAssignTime / 60) % 3 + 3 - type) % 3;

    // Green phase
    if (currentSlot == 0) {

        // get remaining time of traffic light in green phase
        long remainingTime = 60 * (nextAssignTime / 60 + 1) - nextAssignTime;

        // if remaining time >= 6, a vehicle can pass
        if (remainingTime >= 6) {···

        // if remaining time is not sufficient for vehicle to pass
        else {···
    }

    // First red phase
    else if (currentSlot == 1) {···

    // 2nd red phase
    else {···
}
```

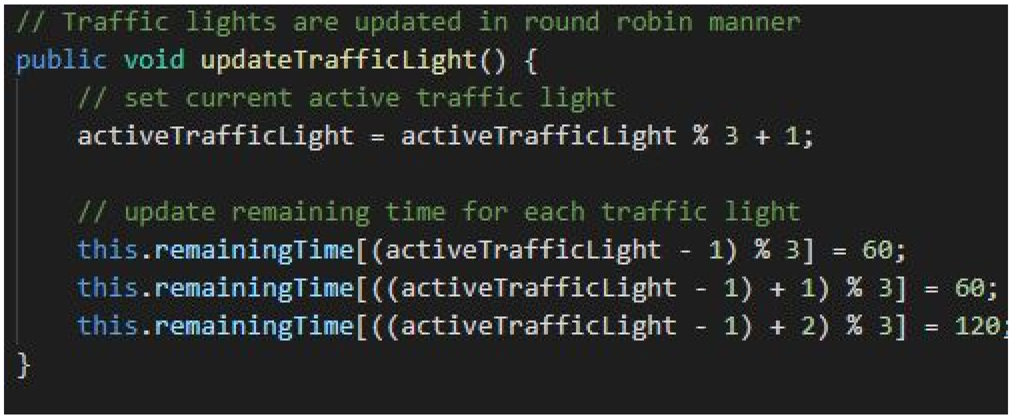
Pseudo Code:

```
currentSlot = ((nextAssignTime / 60) % 4 + 4 - type ) % 4;
If currentSlot is 0
 Then remainingTime = remainingTime = 60 * (nextAssignTime / 60 + 1) - nextAssignTime;
Else if currentSlot is 1
 Then nextAssignTime = 60 * (nextAssignTime / 60 + 3);
Else if currentSlot is 2
 Then nextAssignTime = 60 * (nextAssignTime / 60 + 2);
Else
 Then nextAssignTime = 60 * (nextAssignTime / 60 + 1);
```



```
// array which stores
// remaining time for each traffic light
long[] remainingTime = {0, 0, 0};

// Vehicles queue for traffic light T1
VehicleQueue t1Queue;

// Vehicles queue for traffic light T2
VehicleQueue t2Queue;

// Vehicles queue for traffic light T3
VehicleQueue t3Queue;
```

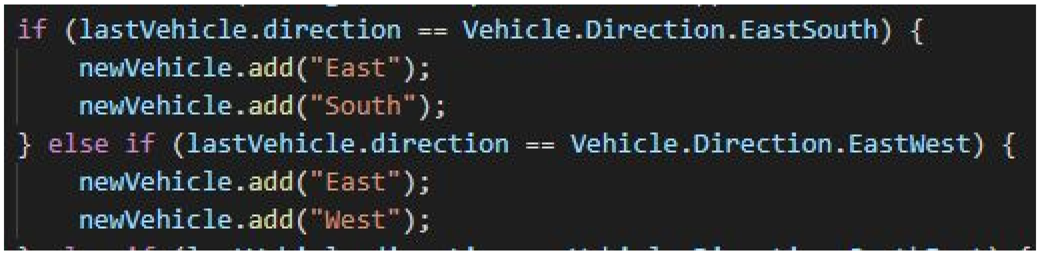
Pseudo Code: (Additional code)
long[] remainingTime = {0, 0, 0, 0};
VehicleQueue t4Queue;



```
// Traffic lights are updated in round robin manner
public void updateTrafficLight() {
    // set current active traffic light
    activeTrafficLight = activeTrafficLight % 3 + 1;

    // update remaining time for each traffic light
    this.remainingTime[(activeTrafficLight - 1) % 3] = 60;
    this.remainingTime[((activeTrafficLight - 1) + 1) % 3] = 60;
    this.remainingTime[((activeTrafficLight - 1) + 2) % 3] = 120;
}
```

Pseudo Code:

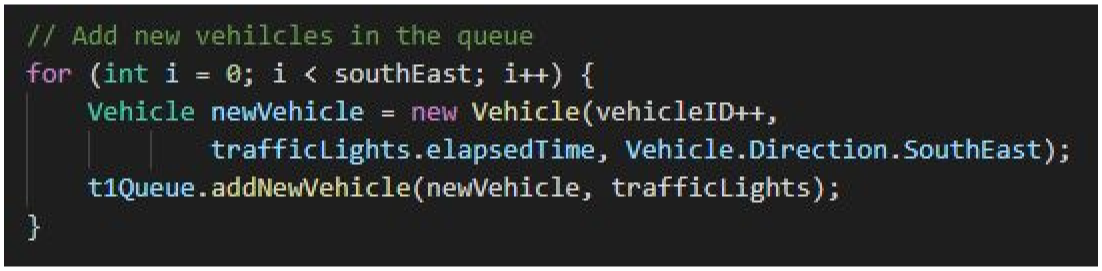
```
 activeTrafficLight = activeTrafficLight % 4 + 1;
       // update remaining time for each traffic light
  this.remainingTime[(activeTrafficLight - 1) % 4] = 60;
  this.remainingTime[((activeTrafficLight - 1) + 1) % 4] = 60;
  this.remainingTime[((activeTrafficLight - 1) + 2) % 4] = 120;
  this.remainingTime[((activeTrafficLight - 1) + 3) % 4] = 180;
```



```
if (lastVehicle.direction == Vehicle.Direction.EastSouth) {
    newVehicle.add("East");
    newVehicle.add("South");
} else if (lastVehicle.direction == Vehicle.Direction.EastWest) {
    newVehicle.add("East");
    newVehicle.add("West");
```

Pseudo Code: Conditions for new possible directions will be added similar to the one written below.

```
Else if(lastVehicle.direction == Vehicle.Direction.NorthEast){
        newVehicle.add("North");
        newVehicle.add("East");
}
```



```
// Add new vehilcles in the queue
for (int i = 0; i < southEast; i++) {
    Vehicle newVehicle = new Vehicle(vehicleID++,
            trafficLights.elapsedTime, Vehicle.Direction.SouthEast);
    t1Queue.addNewVehicle(newVehicle, trafficLights);
}
```

Pseudo Code:

```
for (int i = 0; i < NorthEast / EastNorth / NorthSouth / SouthNorth / WestNorth / NorthWest;
i++) {
        Vehicle newVehicle = new Vehicle(vehicleID++,
            trafficLights.elapsedTime, Vehicle.Direction.SouthEast);
        t1Queue.addNewVehicle(newVehicle, trafficLights);
}
```

These are the major changes that we need to do, to introduce a new traffic light and a new traffic lane. UI part will work in a similar fashion. For vehicles, since we are adding a direction attribute, source and destination will be displayed according to that attribute.