Nombre:

Armando González

Moisés Rodríguez

ID:

1014-4198

1014-5201

Profesor:

Freddy Peña

Materia:

Programación Paralela y Concurrente

Titulo:

Documentación de avance del proyecto final

Descripción del diseño del sistema

El sistema de simulación de tráfico consta de cuatro clases principales: Vehicle, TrafficLight, Street y TrafficController. A continuación se describe el propósito y funcionamiento de cada una:

Clase Vehicle:

- Representa un vehículo en el sistema.
- Atributos:
  - id: Identificador único del vehículo.
  - isEmergency: Indica si el vehículo es de emergencia.
  - direction: Dirección hacia la cual se dirige el vehículo.
  - inIntersection: Indica si el vehículo está en la intersección.
- Métodos:
  - getId, setId, isEmergency, setEmergency, getDirection, setDirection, isInIntersection, setInIntersection: Métodos getters y setters.
  - compareTo: Método para comparar vehículos por su id.

```
public class Vehicle implements Comparable<Vehicle> {
    String id;
    private boolean isEmergency;
    private String direction;
    private boolean inIntersection;

    public Vehicle(String id, boolean isEmergency, String direction) {
        this.id = id;
        this.isEmergency = isEmergency;
        this.direction = direction;
        this.inIntersection = false;
    }


    //…


    @Override
    public int compareTo(Vehicle other) {
        return this.id.compareTo(other.id);
    }
}
```

Clase TrafficLight:

- Representa un semáforo en el sistema.
- Atributos:
  - id: Identificador único del semáforo.

    - green: Estado del semáforo (verde o rojo) gestionado por un AtomicBoolean.
- Métodos:
    - getId, setId: Métodos getters y setters.
    - isGreen: Devuelve el estado actual del semáforo.
    - changeLight: Cambia el estado del semáforo.

```
public class TrafficLight {
    private String id;
    private AtomicBoolean green;

    public TrafficLight(String id) {
        this.id = id;
        this.green = new AtomicBoolean(false);
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public boolean isGreen() {
        return green.get();
    }

    public void changeLight() {
        green.set(!green.get());
    }
}
```

Clase Street:

- Representa una calle en el sistema.
- Atributos:
    - id: Identificador único de la calle.
    - vehicles: Cola de prioridad de vehículos en la calle, implementada con PriorityBlockingQueue.
- Métodos:
    - getId, setId: Métodos getters y setters.
    - getVehicles, setVehicles: Métodos getters y setters para la cola de vehículos.
    - addVehicle: Añade un vehículo a la cola.
    - getNextVehicle: Obtiene y elimina el siguiente vehículo en la cola.
    - hasEmergencyVehicle: Comprueba si hay un vehículo de emergencia en la calle.

```java
public class Street {
    private String id;
    private PriorityBlockingQueue<Vehicle> vehicles;

    public Street(String id) {
        this.id = id;
        this.vehicles = new PriorityBlockingQueue<>(10);
        //this.trafficLight = trafficLight;
    }

    //…

    public boolean hasEmergencyVehicle() {
        for (Vehicle vehicle : vehicles) {
            if (vehicle.isEmergency()) {
                return true;
            }
        }
        return false;
    }
}
```

Clase TrafficController:

- Controla el flujo de tráfico en la intersección.
- Atributos:
  - streets: Lista de calles gestionadas por el controlador.
  - trafficLights: Lista de semáforos gestionados por el controlador.
  - executor: Ejecuta tareas de control a intervalos regulares.
  - exec: Gestiona las tareas de procesamiento de calles.
  - lock: Controla el acceso a la intersección.
  - intersectionOccupied: Indica si la intersección está ocupada.
- Métodos:
  - startControl: Inicia el control del tráfico.
  - manageIntersection: Gestiona el flujo de tráfico en la intersección.
  - processStreet: Procesa los vehículos en una calle.
  - processEmergencyVehicle: Procesa los vehículos de emergencia en una calle.
  - tryToCrossIntersection: Gestiona el cruce de la intersección por un vehículo.
  - crossIntersection: Simula el cruce de la intersección por un vehículo.
  - stopControl: Detiene el control del tráfico.

```
public class TrafficController {
    private List<Street> streets;
    private List<TrafficLight> trafficLights;
    private ScheduledExecutorService executor;
    ExecutorService exec;
    ReentrantLock lock;
    private volatile boolean intersectionOccupied = false;

    public TrafficController(List<TrafficLight> trafficLights, List<Street>
streets) {
        this.streets= streets;
        this.trafficLights = trafficLights;
        this.executor = Executors.newScheduledThreadPool(1);
        this.exec = Executors.newFixedThreadPool(streets.size());
        this.lock = new ReentrantLock();
    }

    //El algoritmo se explica mas abajo
}
```

Explicación de los algoritmos de control

El control del tráfico se gestiona mediante el TrafficController, que utiliza dos servicios de ejecución (executor y exec) y un bloqueo (lock) para coordinar el flujo de vehículos en una intersección.

Inicio del Control (startControl):

- executor programa la tarea manageIntersection para ejecutarse a intervalos regulares.

```
public void startControl() {
    executor.scheduleAtFixedRate(this::manageIntersection, 0, 1,
TimeUnit.SECONDS);
}
```

Gestión de la Intersección (manageIntersection):

- Para cada calle en streets, se envía una tarea a exec para procesar los vehículos en la calle mediante processStreet.

```
public void manageIntersection() {
    for (Street street : streets) {
        exec.submit(() -> processStreet(street));
    }
}
```

Procesamiento de una Calle (processStreet):

- Se verifica si hay un vehículo de emergencia en la calle con hasEmergencyVehicle. Si es así, se llama a processEmergencyVehicle.
- Si no hay vehículos de emergencia, se obtiene el siguiente vehículo en la cola con getNextVehicle y se intenta cruzar la intersección con tryToCrossIntersection.

```
private void processStreet(Street street) {
    while (true) {
        Vehicle vehicle;
        if (street.hasEmergencyVehicle()) {
            processEmergencyVehicle(street);
        } else {
            vehicle = street.getNextVehicle();
            if (vehicle != null) {
                tryToCrossIntersection(vehicle);
            }
        }

        // Esperar un momento antes de volver a verificar la calle para
nuevos vehículos
        try {
            Thread.sleep(100); // Puedes ajustar el tiempo de espera
según sea necesario
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return;
        }
    }
}
```

Procesamiento de Vehículos de Emergencia (processEmergencyVehicle):

- Se procesa cada vehículo en la calle hasta que el vehículo de emergencia haya cruzado la intersección.

```
private void processEmergencyVehicle(Street street) {
    System.out.println("Emergency vehicle detected on street " +
street.getId());
    Vehicle vehicle;
    while ((vehicle = street.getNextVehicle()) != null) {
        System.out.println("Vehicle " + vehicle.getId() + " crossing
before emergency vehicle");
        crossIntersection(vehicle);
        if (vehicle.isEmergency()) {
            System.out.println("Emergency vehicle " + vehicle.getId() +
" has crossed the intersection");
            break; // Detener el procesamiento después del cruce del
vehículo de emergencia
        }
    }
}
```

Intento de Cruce de la Intersección (tryToCrossIntersection):

- Se adquiere un bloqueo (lock) para asegurar que solo un vehículo cruce la intersección a la vez.
- Si la intersección está ocupada, el vehículo espera hasta que esté libre.
- Se simula una parada en una señal de "Stop".
- Se llama a crossIntersection para simular el cruce de la intersección.
- Se libera el bloqueo y se marca la intersección como libre.

```
private void tryToCrossIntersection(Vehicle vehicle) {
    try {
        lock.lock();
        while (intersectionOccupied) {
            System.out.println("Vehicle " + vehicle.getId() + " is
waiting to cross the intersection");
            lock.unlock();
            Thread.sleep(100); // Esperar un poco antes de intentar
nuevamente
            lock.lock();
        }
        intersectionOccupied = true;
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    } finally {
        lock.unlock();
    }

    // Simular la detención en una señal de "Stop"
    System.out.println("Vehicle " + vehicle.getId() + " is stopping at
the stop sign");
    try {
        Thread.sleep(500); // Simular la detención en la señal de
"Stop" (500 ms)
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }

    crossIntersection(vehicle);

    try {
        lock.lock();
        intersectionOccupied = false;
    } finally {
        lock.unlock();
    }
}
```

Cruce de la Intersección (crossIntersection):

- Se marca el vehículo como en la intersección.
- Se simula el tiempo que tarda en cruzar la intersección.
- Se marca el vehículo como fuera de la intersección.

```
private void crossIntersection(Vehicle vehicle) {
    vehicle.setInIntersection(true);
    System.out.println("Vehicle " + vehicle.getId() + " is crossing the
intersection");
    try {
        Thread.sleep(1000); // Simular el cruce de la intersección
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
    vehicle.setInIntersection(false);
}
```

Instrucciones para ejecutar la aplicación

Preparación del entorno:

- Asegúrese de tener JDK 8 o superior instalado.
- Configure su entorno de desarrollo preferido (por ejemplo, IntelliJ IDEA, Eclipse).
- Tener Java FX configurado.