```
package signalGreen;

import java.util.Map;
import repast.simphony.space.gis.Geography;
import repast.simphony.space.graph.Network;
import repast.simphony.space.graph.RepastEdge;

/**
 * CarVehicle simulates vehicles of type Car.
 * Cars are initialised with different graphics, higher speed
 * than other vehicles. Cars can be either slow or fast.
 *
 * @author Yoann
 */
public class CarVehicle extends Vehicle {
	private String carIcon;

	/**
	 * Cars have different types of graphics
	 * depending on their maxVelocity.
	 * Faster cars look like sport cars.
	 *
	 * @param network
	 * @param geography
	 * @param roads
	 * @param maxVelocity
	 */
	public CarVehicle(Network<Junction> network, Geography geography,
			Map<RepastEdge<Junction>, Road> roads, int maxVelocity)
{
		super(network, geography, roads, maxVelocity);
		// set icon
		if (maxVelocity >= Constants.FAST) {
			this.carIcon = Constants.ICON_FAST_CAR;
		}
		else {
			this.carIcon = Constants.ICON_SLOW_CAR;
		}
	}

	/**
	 * @return string the graphics relative name
	 */
	public String getCarIcon() {
		return this.carIcon;
	}

}
```

```
package signalGreen;

/**
 * Constant class holds all arbitrary calibration parameters
 * derived from manual testing of Signal Green.
 *
 * @author Signal Green Team*
 */
public final class Constants {

	public static final String NETWORK = "road network";
	public static final String ID = "signalGreen";

	// put all GIS maps in the following folder
	public static final String MAPS_FOLDER = "data/maps/";

	// user-defined parameter
	public static final String NUM_VEHICLES = "numVehicle";

	// distance between lanes in meters
	public static final double DIST_LANE = 1.0;
	// distance of Lights in meters from Junction for display purposes
	public static final double DIST_LIGHTS = (DIST_LANE * 2); // should be e
q. 2 * DIST_LANE
	// minimum distance between vehicles driving in meters
	public static final double DIST_VEHICLES = 1.8;
	// minimum distance between vehicles stopped in meters
	public static final double DIST_VEHICLES_STOPPED = 1.0;

	// arbitrary value for time to make simulation faster, in reality t = 1
tick.
	// Used to compute velocity and displacement.
	public static final int t = 4;
	// arbitary value used to adjust GIS projection meters
	// because vehicle graphics are bigger than real scale
	public final static int CONV_RATIO_METERS = 70;

	// default speed limit for roads
	public static final int DEFAULT_SPEEDLIMIT = 80;
	// maximum velocity of cars when initialised
	public final static int[] speed = {100, 120, 140, 80};
	// boundaries what is fast and slow, in km/h
	public static final int VERY_SLOW = 80;
	public static final int SLOW = 100;
	public static final int FAST = 140;
	public static final int VERY_FAST = 160;

	// arbitrary value for vehicle acceleration
	public static final double ACCELERATION = 1.6; // m/s
	// acceleration factor: trucks have smaller accel. than cars
	public static final double CAR_SLOW_ACC = 1.0;
	public static final double CAR_FAST_ACC = 1.2;
	public static final double TRUCK_ACC = 0.8;
	public static final int TRUCK_DEFAULT_MAX_VELOCITY = 80;

	// car graphics
	public static String ICON_SLOW_CAR = "car_simple.png";
	public static String ICON_FAST_CAR = "car_fast.png";

	// traffic light signals
	public static enum Signal {
		GREEN, AMBER, RED
	}

	//
	public static enum RoadType {
		SINGLE_LANE, TWO_LANES
	}
```



```
	// roads have two lanes per side
	public static enum Lane {
		INNER, OUTER
	}

}
```

```
package signalGreen;

/**
 *
 * Auxiliary agent just for setting auxiliary
 * coordinates to be used as placeholders on
 * the GIS projection. Typically used to generate road lanes.
 *
 * @author Yoann
 */
public class CoordinateAgent extends GisAgent {

        public CoordinateAgent() {
                super();
        }

}
```

```
package signalGreen;

import com.vividsolutions.jts.geom.Coordinate;

import repast.simphony.space.gis.Geography;
import repast.simphony.space.graph.Network;

/**
 * A generic GIS agent.
 * Every GIS agent should extend this class.
 */
public abstract class GisAgent {

	public static int UniqueID = 0;
	private int ID;

	// Repast projections
	private Network<Junction> network;
	private Geography geography;

	// position on GIS projection
	private Coordinate coordinate;
	// field used in GIS display for debug purposes
	protected String debug;
	// gets the base url to display agent's icons
	private String baseURL;

	/**
	 * Default constructor.
	 * This constructor should not be used.
	 */
	public GisAgent() {
		this.ID = UniqueID++;
		network = null;
		geography = null;
	}

	/**
	 * Constructs a generic GIS agent with its unique ID
	 * and references to the road network and GIS geography.
	 *
	 * @param network
	 * @param geography
	 */
	public GisAgent(Network<Junction> network, Geography geography) {
		this.ID = UniqueID++;
		this.network = network;
		this.geography = geography;
		this.baseURL = System.getProperty("user.dir");
	}

	/**
	 * Get the GIS geography
	 * @return geograpy
	 */
	public Geography getGeography() {
		return geography;
	}

	/**
	 * Get the road network
	 * @return network
	 */
	public Network<Junction> getNetwork() {
		return network;
	}

	/**
	 * Every GIS agent has a unique ID.
```



```
	 * @return unique ID
	 */
	public int getID() {
		return ID;
	}

	/**
	 * Returns the url where the raphics are located.
	 * Used by sld stylesheets for display purposes.
	 *
	 * @return url
	 */
	public String getBaseURL() {
		return baseURL;
	}

    /**
     * Get the coordinates on GIS projection.
     * @return coordinate
     */
    public Coordinate getCoords() {
		return coordinate;
    }

    public void setCoords(Coordinate c) {
		this.coordinate = c;
    }

	public String getDebug() {
		return debug;
	}
}
```

```
package signalGreen;

import repast.simphony.space.gis.Geography;
import repast.simphony.space.graph.Network;

/**
 *
 * Give Way sign junction for traffic management policies.
 * In proximity of give way intersections,
 * vehicles check for each road segment which
 * vehicle is closest to that intersection,
 * to know who has precedence.
 *
 * @author Yoann
 *
 */
public class GiveWaySign extends Junction {

	/**
	 * Constructs an instance of Give Way junction.
	 * @param network
	 * @param geography
	 */
	public GiveWaySign(Network<Junction> network, Geography geography) {
		super(network, geography);
	}

}
```

```
package signalGreen;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Queue;

import repast.simphony.space.gis.Geography;
import repast.simphony.space.graph.Network;
import repast.simphony.space.graph.RepastEdge;
import signalGreen.Constants.Lane;

/**
 * Generic class for junctions of the Traffic Simulator.<br />
 *
 * A Junction object is a node on a Graph. It can have a lane to
 * any number of other Junction objects. Each lane can have
 * the direction specified and these lanes represent Edges
 * on the Graph.
 *
 * TrafficLights is a subclass of Junction, and has the special
 * behavior of scheduling traffic light management.
 *
 * @author Waqar
 *
 */
public class Junction extends GisAgent {

	private List<Road> roads;
	//List of Junctions it has a lane between
	private List<Junction> junctions;
	// Map holds a queue of Vehicles for each Junction.
	// This way we know for each incoming road segment to the current junctio
n
	// which vehicles are approaching.
	public Map<Junction, PriorityBlockingDeque<Vehicle>> vehicles;

	/**
	 * Generic Junction constructor.
	 *
	 * @param network
	 * @param geography
	 */
	public Junction(Network<Junction> network, Geography geography) {
		super(network, geography);
		this.junctions = new ArrayList<Junction>();
		this.roads = new ArrayList<Road>();
		this.vehicles = new HashMap<Junction, PriorityBlockingDeque<Vehi
cle>>();
	}

	/**
	 * @return List of junctions that this junction has a lane between.
	 */
	public List<Junction> getJunctions() {
		return junctions;
	}

	/**
	 * Tells the Junction about its adjacent Junctions. The given Junction
	 * is added to the List of Junctions that it now has a lane between.
	 *
	 * @param j is the other Junction the lane will be between.
	 */
	public void addJunction(Junction j) {
		this.junctions.add(j);
	}
```



```
	/**
	 * Remove a lane between another Junction. This given Junction
	 * is removed from the List of Junctions that it has a lane between.
	 * It represents an Edge being removed on the Graph and is therefore
	 * updated on the Network Object.
	 *
	 * @param junc is the other Junction the lane is between.
	 * @param out is a boolean flag for the lane direction being outward.
	 */
	public void removeLane(Junction junc, boolean out) {
		this.junctions.remove(junc);

		RepastEdge<Junction> edge;

		if (out) {
			edge = getNetwork().getEdge(this, junc);
		} else {
			edge = getNetwork().getEdge(junc, this);
		}

		if (edge != null)
			getNetwork().removeEdge(edge);
	}

	/**
	 * Remove all lanes joining toward this Junction, the Graph
	 * will therefore have no Edges to this Node.
	 */
	public void removeAllLanes() {

		RepastEdge<Junction> edgeIn;
		RepastEdge<Junction> edgeOut;

		for (Junction junc : junctions) {
			edgeIn = getNetwork().getEdge(this, junc);
			edgeOut = getNetwork().getEdge(junc, this);
			getNetwork().removeEdge(edgeIn);
			getNetwork().removeEdge(edgeOut);
		}

		this.junctions.clear();
	}

	/**
	 * Two junctions with same coordinates are
	 * equivalent.
	 * @return true if junctions have the same coordinates
	 */
@Override
public boolean equals(Object obj) {
    if (!(obj instanceof Junction)) {
		return false;
    }
    Junction j = (Junction) obj;
    return this.getCoords().equals(j.getCoords());
}

    public List<Road> getRoads() {
		return this.roads;
    }


    /**
     * Returns a list of vehicles that are running on a road segment
     * from j to this junction. Assumes j is in the this.junctions list.
     *
     * @param j the junction
     * @return queue of vehicles
```

```
	 */
	public PriorityBlockingDeque<Vehicle> getVehiclesQueue(Junction j) {
		return this.vehicles.get(j);
	}

	/**
	 * Used by the context builder class only.
	 *
	 * @return all queues for initialisation purposes
	 */
	public Map<Junction, PriorityBlockingDeque<Vehicle>> getVehiclesMap() {
		return this.vehicles;
	}

	/**
	 * Every vehicle entering a new road segment should call this method.
	 * Every junction holds a queue of vehicles running on a particular road
	 * segment going towards this junction from junction j.
	 *
	 * @param j junction at the other side of the current road segment
	 * @param v vehicle entering a road segment
	 */
	public void enqueueVehicle(Junction j, Vehicle v) {
		PriorityBlockingDeque<Vehicle> q = this.vehicles.get(j);
		q.add(v);
	}

	/**
	 * Once a vehicle has overtaken another vehicle
	 * it must tell the next junction its new position.
	 * This is done by re-adding the vehicle in the
	 * priority queue with its updated weight, the distance
	 * to the next junction.
	 *
	 * @param j previous junction
	 * @param v vehicle
	 */
	public void reorderVehicle(Junction j, Vehicle v) {
		PriorityBlockingDeque<Vehicle> q = this.vehicles.get(j);
		q.remove(v);
		q.add(v);
	}

	/**
	 * Every vehicle leaving a road segment should call this method.
	 * Vehicles are removed from queue.
	 *
	 * @see signalGreen.Junction#enqueueVehicle(Junction j, Vehicle v)
	 * @param j junction at the other side of the current road segment
	 * @param v vehicle leaving a road segment
	 * @return true if success
	 */
	public boolean dequeueVehicle(Junction j, Vehicle v) {
		PriorityBlockingDeque<Vehicle> q = this.vehicles.get(j);
		return q.remove(v);
	}

	/**
	 * Returns the closest vehicle to the current junction
	 * from junction j. It does not take into account lanes.
	 *
	 * @param j the junction
	 * @return v closest vehicle from j, if any
	 */
	public Vehicle peekVehicle(Junction j) {
		Vehicle v = null;
		PriorityBlockingDeque<Vehicle> q = this.vehicles.get(j);
		v = q.element();
		return v;
```



```
	}

	/**
	 * Debug the vehicles queue for a particular junction.
	 *
	 * @param j the junction
	 */
	public void printVehiclesQueue(Junction j) {
		PriorityBlockingDeque<Vehicle> q = this.vehicles.get(j);
		System.out.println(q.toString());
		System.out.println("Peek vehicle: " + this.peekVehicle(j).toString()
);
	}

	/**
	 * Returns an array with size of 2 of vehicles
	 * that are ahead/behind of a given vehicle as follows:<br />
	 * v[0] => Vehicle on Lane.OUTER<br />
	 * v[1] => Vehicle on Lane.INNER
	 *
	 * @param j junction
	 * @param vehicle
	 * @param checkAhead check for vehicle ahead if true
	 * @return array of vehicles
	 */
	public Vehicle[] getNextVehicles(Junction j, Vehicle vehicle, boolean ch
eckAhead) {
		Vehicle[] v = new Vehicle[2];
		v[0] = null; // outer lane
		v[1] = null; // inner lane
		Vehicle tmp = null;
		boolean found = false;
		boolean foundOuter = false;
		boolean foundInner = false;
		PriorityBlockingDeque<Vehicle> q = this.vehicles.get(j);

		Iterator<Vehicle> it = null;
		if (checkAhead == true) {
			it = q.descendingIterator();
		}
		else {
			it = q.iterator();
		}

		while (it.hasNext()) {
			tmp = it.next();

			if (found == true) {
				if (foundOuter == false && tmp.getLane() == Lane
.OUTER) {
					v[0] = tmp;
					foundOuter = true;

				}
				if (foundInner == false && tmp.getLane() == Lane
.INNER) {
					v[1] = tmp;
					foundInner = true;

				}
			}

			if (tmp.equals(vehicle)) {
				found = true;
			}
		}
		return v;
	}
```

```
        /**
         * How many vehicles there are on this road segment.
         *
         * @param j the origin junction
         * @return number of vehicles
         */
        public int getNumVehicles(Junction j) {
                PriorityBlockingDeque<Vehicle> q = this.vehicles.get(j);
                return q.size();
        }
}
```

```
package signalGreen;

/**
 * Auxiliary agent used to display lanes on
 * the GIS projection.
 *
 * @author Yoann
 *
 */
public class LaneAgent extends GisAgent {

        /**
         * Uses defauls constructor.
         * Lane is used for display purposes only, so it
         * doesn't need references to geography or network.
         *
         */
        public LaneAgent() {
                super();
        }

}
```

```
package signalGreen;

import signalGreen.Constants.Signal;

/**
 * Light object for Traffic Light Junctions.
 *
 * @author Waqar
 *
 */
public class Light extends GisAgent {

	private Signal signal;

	/**
	 * Initialize light with user-defined condition.
	 *
	 * @param signal to start with.
	 */
	public Light(Signal signal) {

		this.signal = signal;
	}

	/**
	 * Get the current signal.
	 *
	 * @return the signal
	 */
	public Signal getSignal() {
		return signal;
	}

	/**
	 * Set the current signal.
	 *
	 * @param signal the signal to set
	 */
	public void setSignal(Signal signal) {
		this.signal = signal;
	}

	/**
	 * Switches signal from GREEN to RED or AMBER.
	 */
	public void toggleSignal() {
		if (this.signal == Signal.GREEN) {
			this.signal = Signal.RED;
		} else {
			this.signal = Signal.GREEN;
		}
	}

	/**
	 * Used by the GIS display to know which color
	 * has the traffic light at any moment.
	 *
	 * @return integer representation of the current signal
	 */
	public int getColor() {
		// DO NOT CHANGE VALUES
		if (this.signal == Signal.GREEN) return 0;
		if (this.signal == Signal.AMBER) return 5;
		if (this.signal == Signal.RED) return 10;
		return 15;
	}

}
```

```
package signalGreen;

import java.util.*;
import java.util.concurrent.BlockingDeque;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

/**
 * Copyright (c) 2007, Aviad Ben Dov
 *
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without modificati
on,
 * are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice, thi
s list
 * of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright notice,
this
 * list of conditions and the following disclaimer in the documentation and/or o
ther
 * materials provided with the distribution.
 * 3. Neither the name of Infomancers, Ltd. nor the names of its contributors ma
y be
 * used to endorse or promote products derived from this software without specif
ic
 * prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 */


/**
 * An optionally-bounded {@linkplain BlockingDeque blocking deque} based on
 * a navigable set.
 * <p/>
 * <p> The optional capacity bound constructor argument serves as a
 * way to prevent excessive expansion. The capacity, if unspecified,
 * is equal to {@link Integer#MAX_VALUE}.
 * <p/>
 * <p>This class and its iterator implement all of the
 * <em>optional</em> methods of the {@link Collection} and {@link
 * Iterator} interfaces.
 * <p/>
 * This code is loosely based on the {@linkplain java.util.concurrent.LinkedBloc
kingDeque linked blocking deque} code.
 *
 * @author Aviad Ben Dov
 * @param <E> the type of elements held in this collection
 * @since 0.3
 */
public class PriorityBlockingDeque<E>
        extends AbstractQueue<E>
        implements BlockingDeque<E>, java.io.Serializable {
```



```
    /*
     * Implemented as a navigable set protected by a
     * single lock and using conditions to manage blocking.
     */

    private final int capacity;

    private final LinkedList<E> list;
    /**
     * Main lock guarding all access
     */
    private final ReentrantLock lock = new ReentrantLock();
    /**
     * Condition for waiting takes
     */
    private final Condition notEmpty = lock.newCondition();
    /**
     * Condition for waiting puts
     */
    private final Condition notFull = lock.newCondition();
    private Comparator<E> comparator;

    /**
     * Creates a <tt>PriorityBlockingDeque</tt> with a capacity of
     * {@link Integer#MAX_VALUE}.
     */
    public PriorityBlockingDeque() {
        this(null, Integer.MAX_VALUE);
    }

    /**
     * Creates a <tt>PriorityBlockingDeque</tt> with the given (fixed) capacity.
     *
     * @param capacity the capacity of this deque
     * @throws IllegalArgumentException if <tt>capacity</tt> is less than 1
     */
    public PriorityBlockingDeque(int capacity) {
        this(null, capacity);
    }

    public PriorityBlockingDeque(Comparator<E> comparator, int capacity) {
        if (capacity <= 0) throw new IllegalArgumentException();
        this.capacity = capacity;
        this.list = new LinkedList<E>();
        this.comparator = comparator;
    }

    // Basic adding and removing operations, called only while holding lock

    /**
     * Adds e or returns false if full.
     *
     * @param e The element to add.
     * @return Whether adding was successful.
     */
    private boolean innerAdd(E e) {
        if (list.size() >= capacity)
            return false;

        int insertionPoint = Collections.binarySearch(list, e, comparator);
        if (insertionPoint < 0) {
            // this means the key didn't exist, so the insertion point is negati
ve minus 1.
            insertionPoint = -insertionPoint - 1;
        }

        list.add(insertionPoint, e);
//        Collections.sort(list, comparator);
        notEmpty.signal();
```

```
        return true;
    }

    /**
     * Removes and returns first element, or null if empty.
     *
     * @return The removed element.
     */
    private E innerRemoveFirst() {
        E f = list.pollFirst();
        if (f == null)
            return null;


        notFull.signal();
        return f;
    }

    /**
     * Removes and returns last element, or null if empty.
     *
     * @return The removed element.
     */
    private E innerRemoveLast() {
        E l = list.pollLast();
        if (l == null)
            return null;

        notFull.signal();
        return l;
    }

    // BlockingDeque methods

    /**
     * @throws IllegalStateException {@inheritDoc}
     * @throws NullPointerException  {@inheritDoc}
     */
    public void addFirst(E e) {
        if (!offerFirst(e))
            throw new IllegalStateException("Deque full");
    }

    /**
     * @throws IllegalStateException {@inheritDoc}
     * @throws NullPointerException  {@inheritDoc}
     */
    public void addLast(E e) {
        if (!offerLast(e))
            throw new IllegalStateException("Deque full");
    }

    /**
     * @throws NullPointerException {@inheritDoc}
     */
    public boolean offerFirst(E e) {
        if (e == null) throw new NullPointerException();
        lock.lock();
        try {
            return innerAdd(e);
        } finally {
            lock.unlock();
        }
    }

    /**
     * @throws NullPointerException {@inheritDoc}
```



```
    public boolean offerLast(E e) {
        if (e == null) throw new NullPointerException();
        lock.lock();
        try {
            return innerAdd(e);
        } finally {
            lock.unlock();
        }
    }

    /**
     * @throws NullPointerException {@inheritDoc}
     * @throws InterruptedException {@inheritDoc}
     */
    public void putFirst(E e) throws InterruptedException {
        if (e == null) throw new NullPointerException();
        lock.lock();
        try {
            while (!innerAdd(e))
                notFull.await();
        } finally {
            lock.unlock();
        }
    }

    /**
     * @throws NullPointerException {@inheritDoc}
     * @throws InterruptedException {@inheritDoc}
     */
    public void putLast(E e) throws InterruptedException {
        if (e == null) throw new NullPointerException();
        lock.lock();
        try {
            while (!innerAdd(e))
                notFull.await();
        } finally {
            lock.unlock();
        }
    }

    /**
     * @throws NullPointerException {@inheritDoc}
     * @throws InterruptedException {@inheritDoc}
     */
    public boolean offerFirst(E e, long timeout, TimeUnit unit)
            throws InterruptedException {
        if (e == null) throw new NullPointerException();
        long nanos = unit.toNanos(timeout);
        lock.lockInterruptibly();
        try {
            for (; ;) {
                if (innerAdd(e))
                    return true;
                if (nanos <= 0)
                    return false;
                nanos = notFull.awaitNanos(nanos);
            }
        } finally {
            lock.unlock();
        }
    }

    /**
     * @throws NullPointerException {@inheritDoc}
     * @throws InterruptedException {@inheritDoc}
     */
    public boolean offerLast(E e, long timeout, TimeUnit unit)
            throws InterruptedException {
        if (e == null) throw new NullPointerException();
```

```
        long nanos = unit.toNanos(timeout);
        lock.lockInterruptibly();
        try {
            for (; ;) {
                if (innerAdd(e))
                    return true;
                if (nanos <= 0)
                    return false;
                nanos = notFull.awaitNanos(nanos);
            }
        } finally {
            lock.unlock();
        }
    }

    /**
     * @throws NoSuchElementException {@inheritDoc}
     */
    public E removeFirst() {
        E x = pollFirst();
        if (x == null) throw new NoSuchElementException();
        return x;
    }

    /**
     * @throws NoSuchElementException {@inheritDoc}
     */
    public E removeLast() {
        E x = pollLast();
        if (x == null) throw new NoSuchElementException();
        return x;
    }

    public E pollFirst() {
        lock.lock();
        try {
            return innerRemoveFirst();
        } finally {
            lock.unlock();
        }
    }

    public E pollLast() {
        lock.lock();
        try {
            return innerRemoveLast();
        } finally {
            lock.unlock();
        }
    }

    public E takeFirst() throws InterruptedException {
        lock.lock();
        try {
            E x;
            while ((x = innerRemoveFirst()) == null)
                notEmpty.await();
            return x;
        } finally {
            lock.unlock();
        }
    }

    public E takeLast() throws InterruptedException {
        lock.lock();
        try {
            E x;
            while ((x = innerRemoveLast()) == null)
                notEmpty.await();
```



```
            return x;
        } finally {
            lock.unlock();
        }
    }

    public E pollFirst(long timeout, TimeUnit unit)
            throws InterruptedException {
        long nanos = unit.toNanos(timeout);
        lock.lockInterruptibly();
        try {
            for (; ;) {
                E x = innerRemoveFirst();
                if (x != null)
                    return x;
                if (nanos <= 0)
                    return null;
                nanos = notEmpty.awaitNanos(nanos);
            }
        } finally {
            lock.unlock();
        }
    }

    public E pollLast(long timeout, TimeUnit unit)
            throws InterruptedException {
        long nanos = unit.toNanos(timeout);
        lock.lockInterruptibly();
        try {
            for (; ;) {
                E x = innerRemoveLast();
                if (x != null)
                    return x;
                if (nanos <= 0)
                    return null;
                nanos = notEmpty.awaitNanos(nanos);
            }
        } finally {
            lock.unlock();
        }
    }

    /**
     * @throws NoSuchElementException {@inheritDoc}
     */
    public E getFirst() {
        E x = peekFirst();
        if (x == null) throw new NoSuchElementException();
        return x;
    }

    /**
     * @throws NoSuchElementException {@inheritDoc}
     */
    public E getLast() {
        E x = peekLast();
        if (x == null) throw new NoSuchElementException();
        return x;
    }

    public E peekFirst() {
        lock.lock();
        try {
            return list.size() == 0 ? null : list.peekFirst();
        } finally {
            lock.unlock();
        }
    }
```

```
    public E peekLast() {
        lock.lock();
        try {
            return list.size() == 0 ? null : list.peekLast();
        } finally {
            lock.unlock();
        }
    }

    public boolean removeFirstOccurrence(Object o) {
        if (o == null) return false;
        lock.lock();
        try {
            for (Iterator<E> it = list.iterator(); it.hasNext();) {
                E e = it.next();
                if (o.equals(e)) {
                    it.remove();
                    return true;
                }
            }
            return false;
        } finally {
            lock.unlock();
        }
    }

    public boolean removeLastOccurrence(Object o) {
        if (o == null) return false;
        lock.lock();
        try {
            for (Iterator<E> it = list.descendingIterator(); it.hasNext();) {
                E e = it.next();
                if (o.equals(e)) {
                    it.remove();
                    return true;
                }
            }
            return false;
        } finally {
            lock.unlock();
        }
    }

    // BlockingQueue methods

    /**
     * Inserts the specified element to the deque unless it would
     * violate capacity restrictions.  When using a capacity-restricted deque,
     * it is generally preferable to use method {@link #offer(Object) offer}.
     * <p/>
     * <p>This method is equivalent to {@link #addLast}.
     *
     * @throws IllegalStateException if the element cannot be added at this
     *                               time due to capacity restrictions
     * @throws NullPointerException  if the specified element is null
     */
    @Override
    public boolean add(E e) {
        addLast(e);
        return true;
    }

    /**
     * @throws NullPointerException if the specified element is null
     */
    public boolean offer(E e) {
        return offerLast(e);
    }
```



```
    /**
     * @throws NullPointerException {@inheritDoc}
     * @throws InterruptedException {@inheritDoc}
     */
    public void put(E e) throws InterruptedException {
        putLast(e);
    }

    /**
     * @throws NullPointerException {@inheritDoc}
     * @throws InterruptedException {@inheritDoc}
     */
    public boolean offer(E e, long timeout, TimeUnit unit)
            throws InterruptedException {
        return offerLast(e, timeout, unit);
    }

    /**
     * Retrieves and removes the head of the queue represented by this deque.
     * This method differs from {@link #poll poll} only in that it throws an
     * exception if this deque is empty.
     * <p/>
     * <p>This method is equivalent to {@link #removeFirst() removeFirst}.
     *
     * @return the head of the queue represented by this deque
     * @throws NoSuchElementException if this deque is empty
     */
    @Override
    public E remove() {
        return removeFirst();
    }

    public E poll() {
        return pollFirst();
    }

    public E take() throws InterruptedException {
        return takeFirst();
    }

    public E poll(long timeout, TimeUnit unit) throws InterruptedException {
        return pollFirst(timeout, unit);
    }

    /**
     * Retrieves, but does not remove, the head of the queue represented by
     * this deque.  This method differs from {@link #peek peek} only in that
     * it throws an exception if this deque is empty.
     * <p/>
     * <p>This method is equivalent to {@link #getFirst() getFirst}.
     *
     * @return the head of the queue represented by this deque
     * @throws NoSuchElementException if this deque is empty
     */
    @Override
    public E element() {
        return getFirst();
    }

    public E peek() {
        return peekFirst();
    }

    /**
     * Returns the number of additional elements that this deque can ideally
     * (in the absence of memory or resource constraints) accept without
     * blocking. This is always equal to the initial capacity of this deque
     * less the current <tt>size</tt> of this deque.
     * <p/>
```

```
     * <p>Note that you <em>cannot</em> always tell if an attempt to insert
     * an element will succeed by inspecting <tt>remainingCapacity</tt>
     * because it may be the case that another thread is about to
     * insert or remove an element.
     */
    public int remainingCapacity() {
        lock.lock();
        try {
            return capacity - list.size();
        } finally {
            lock.unlock();
        }
    }

    /**
     * @throws UnsupportedOperationException {@inheritDoc}
     * @throws ClassCastException            {@inheritDoc}
     * @throws NullPointerException          {@inheritDoc}
     * @throws IllegalArgumentException      {@inheritDoc}
     */
    public int drainTo(Collection<? super E> c) {
        if (c == null)
            throw new NullPointerException();
        if (c == this)
            throw new IllegalArgumentException();
        lock.lock();
        try {
            for (E e : list) {
                c.add(e);
            }
            int n = list.size();
            list.clear();
            notFull.signalAll();
            return n;
        } finally {
            lock.unlock();
        }
    }

    /**
     * @throws UnsupportedOperationException {@inheritDoc}
     * @throws ClassCastException            {@inheritDoc}
     * @throws NullPointerException          {@inheritDoc}
     * @throws IllegalArgumentException      {@inheritDoc}
     */
    public int drainTo(Collection<? super E> c, int maxElements) {
        if (c == null)
            throw new NullPointerException();
        if (c == this)
            throw new IllegalArgumentException();
        lock.lock();
        try {
            int n = 0;
            for (Iterator<E> it = list.iterator(); n < maxElements && it.hasNext
();) {
                E e = it.next();
                c.add(e);
                it.remove();
                ++n;
            }

            notFull.signalAll();
            return n;
        } finally {
            lock.unlock();
        }
    }

    // Stack methods
```



```
    /**
     * @throws IllegalStateException {@inheritDoc}
     * @throws NullPointerException  {@inheritDoc}
     */
    public void push(E e) {
        addFirst(e);
    }

    /**
     * @throws NoSuchElementException {@inheritDoc}
     */
    public E pop() {
        return removeFirst();
    }

    // Collection methods

    /**
     * Removes the first occurrence of the specified element from this deque.
     * If the deque does not contain the element, it is unchanged.
     * More formally, removes the first element <tt>e</tt> such that
     * <tt>o.equals(e)</tt> (if such an element exists).
     * Returns <tt>true</tt> if this deque contained the specified element
     * (or equivalently, if this deque changed as a result of the call).
     * <p/>
     * <p>This method is equivalent to
     * {@link #removeFirstOccurrence(Object) removeFirstOccurrence}.
     *
     * @param o element to be removed from this deque, if present
     * @return <tt>true</tt> if this deque changed as a result of the call
     */
    @Override
    public boolean remove(Object o) {
        return removeFirstOccurrence(o);
    }

    /**
     * Returns the number of elements in this deque.
     *
     * @return the number of elements in this deque
     */
    @Override
    public int size() {
        lock.lock();
        try {
            return list.size();
        } finally {
            lock.unlock();
        }
    }

    /**
     * Returns <tt>true</tt> if this deque contains the specified element.
     * More formally, returns <tt>true</tt> if and only if this deque contains
     * at least one element <tt>e</tt> such that <tt>o.equals(e)</tt>.
     *
     * @param o object to be checked for containment in this deque
     * @return <tt>true</tt> if this deque contains the specified element
     */
    @Override
    public boolean contains(Object o) {
        if (o == null) return false;
        lock.lock();
        try {
            return list.contains(o);
        } finally {
            lock.unlock();
        }
```

```
    }

    /**
     * Returns an array containing all of the elements in this deque, in
     * proper sequence (from first to last element).
     * <p/>
     * <p>The returned array will be "safe" in that no references to it are
     * maintained by this deque.  (In other words, this method must allocate
     * a new array).  The caller is thus free to modify the returned array.
     * <p/>
     * <p>This method acts as bridge between array-based and collection-based
     * APIs.
     *
     * @return an array containing all of the elements in this deque
     */
    @Override
    public Object[] toArray() {
        lock.lock();
        try {
            return list.toArray();
        } finally {
            lock.unlock();
        }
    }

    /**
     * Returns an array containing all of the elements in this deque, in
     * proper sequence; the runtime type of the returned array is that of
     * the specified array.  If the deque fits in the specified array, it
     * is returned therein.  Otherwise, a new array is allocated with the
     * runtime type of the specified array and the size of this deque.
     * <p/>
     * <p>If this deque fits in the specified array with room to spare
     * (i.e., the array has more elements than this deque), the element in
     * the array immediately following the end of the deque is set to
     * <tt>null</tt>.
     * <p/>
     * <p>Like the {@link #toArray()} method, this method acts as bridge between
     * array-based and collection-based APIs.  Further, this method allows
     * precise control over the runtime type of the output array, and may,
     * under certain circumstances, be used to save allocation costs.
     * <p/>
     * <p>Suppose <tt>x</tt> is a deque known to contain only strings.
     * The following code can be used to dump the deque into a newly
     * allocated array of <tt>String</tt>:
     * <p/>
     * <pre>
     *     String[] y = x.toArray(new String[0]);</pre>
     * <p/>
     * Note that <tt>toArray(new Object[0])</tt> is identical in function to
     * <tt>toArray()</tt>.
     *
     * @param a the array into which the elements of the deque are to
     *          be stored, if it is big enough; otherwise, a new array of the
     *          same runtime type is allocated for this purpose
     * @return an array containing all of the elements in this deque
     * @throws ArrayStoreException  if the runtime type of the specified array
     *                              is not a supertype of the runtime type of ev
ery element in
     *                              this deque
     * @throws NullPointerException if the specified array is null
     */
    @Override
    public <T> T[] toArray(T[] a) {
        lock.lock();
        try {
            return list.toArray(a);
        } finally {
            lock.unlock();
```



```
        }
    }

    @Override
    public String toString() {
        lock.lock();
        try {
            return super.toString();
        } finally {
            lock.unlock();
        }
    }

    /**
     * Atomically removes all of the elements from this deque.
     * The deque will be empty after this call returns.
     */
    @Override
    public void clear() {
        lock.lock();
        try {
            list.clear();
            notFull.signalAll();
        } finally {
            lock.unlock();
        }
    }

    @Override
    public Iterator<E> iterator() {
        return list.iterator();
    }

    public Iterator<E> descendingIterator() {
        return list.descendingIterator();
    }
}
```

```
package signalGreen;

import java.util.ArrayList;

import com.vividsolutions.jts.geom.Coordinate;

import repast.simphony.space.graph.RepastEdge;

/**
 * Road agent is used to display the road on the GIS display.
 *
 * @author Yoann
 *
 */
public class Road extends GisAgent {

	public static int UniqueID = 0;
	private int ID;
	private String name;
//	private RepastEdge<Junction> inEdge = null;
//	private RepastEdge<Junction> outEdge = null;
	private ArrayList<Junction> junctions;
	private ArrayList<Coordinate> coordinates; // A list of coordinates betw
een the two junctions
	private double length = 0;
	private int speedLimit = 30; //mph in built up areas in uk

	/**
	 * Constructs a road with no speed limit.
	 * Defaults to 80 Km/h
	 * @param name of street, ex. "Madison Ave"
	 */
	public Road(String name) {
		super();
		this.name = name;
		this.junctions = new ArrayList<Junction>();
		this.coordinates = new ArrayList<Coordinate>();
		this.speedLimit = Constants.DEFAULT_SPEEDLIMIT;
	}


	/**
	 * preferred contructor for roads.
	 * Speed limit is usually determined from
	 * the GIS attributes of the shapefile.
	 *
	 * @param name
	 * @param speedLimit
	 */
	public Road(String name, int speedLimit){
		this.name = name;
		this.junctions = new ArrayList<Junction>();
		this.coordinates = new ArrayList<Coordinate>();
		this.speedLimit = speedLimit;
	}

	@Override
	public boolean equals(Object obj) {
		if (!(obj instanceof Road))
			return false;
		Road b = (Road) obj;
		return this.ID == b.ID;
	}

	public String getName() {
		return this.name;
	}

	public void setName(String name) {
```



```
		this.name = name;
	}

	@Override
	public String toString() {
		return "Road: ID: " + this.getID() + (this.getName() == null ? ""
: ", Name: " + this.getName() +
				", Length: " + this.getLength());
	}

	/**
	 * Roads need to know which junctions they are connected to.
	 *
	 * @param j
	 */
	public void addJunction(Junction j) {
		if (this.junctions.size() == 2) {
			System.err.println("Road Error: only two Junctions allowed.");
		}
		this.junctions.add(j);
	}

	public ArrayList<Junction> getJunctions() {
		if (this.junctions.size() != 2) {
			System.err.println("Road Error: road must have two Junctions.");
		}
		return this.junctions;
	}

	/**
	 * Length is determined while reading
	 * spatial data in the shapefile.
	 * @param len the length in meters
	 */
	public void setLength(double len) {
		this.length = len;
	}

	public double getLength() {
		return this.length;
	}

	public int getSpeedLimit() {
		return speedLimit;
	}

	public void setSpeedLimit(int speedLimit) {
		this.speedLimit = speedLimit;
	}

}
```

```
package signalGreen;

import java.io.File;
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Random;

import org.geotools.data.shapefile.ShapefileDataStore;
import org.geotools.data.simple.SimpleFeatureIterator;
import org.opengis.feature.simple.SimpleFeature;

import com.vividsolutions.jts.geom.Coordinate;
import com.vividsolutions.jts.geom.Geometry;
import com.vividsolutions.jts.geom.GeometryFactory;
import com.vividsolutions.jts.geom.LineString;
import com.vividsolutions.jts.geom.MultiLineString;
import com.vividsolutions.jts.geom.Point;

import repast.simphony.context.Context;
import repast.simphony.context.space.gis.GeographyFactoryFinder;
import repast.simphony.context.space.graph.NetworkBuilder;
import repast.simphony.dataLoader.ContextBuilder;
import repast.simphony.engine.environment.RunEnvironment;
import repast.simphony.parameter.Parameters;
import repast.simphony.space.gis.Geography;
import repast.simphony.space.gis.GeographyParameters;
import repast.simphony.space.graph.Network;
import repast.simphony.space.graph.RepastEdge;

/**
 * This is custom context builder implementation which is responsible
 * to perform the initialization of the Traffic Simulator.
 *
 * @see repast.simphony.dataLoader.ContextBuilder
 * @author Waqar, Yoann
 *
 */
public class SignalGreenBuilder implements ContextBuilder<Object> {

	// List to store Junctions
	private List<Junction> junctions;
	private Network<Junction> network;
	private Geography geography;

	// user-defined parameters
	private int vehCount;
	private boolean usesTrafficLights;
	private String inputShapefile;

	// holds mapping between repast edges and roads, used to get the individ
ual coordinates
	// alond the road segment.
	private Map<RepastEdge<Junction>, Road> roads = new HashMap<RepastEdge<J
unction>, Road>();

	@SuppressWarnings({ "rawtypes", "unchecked" })
	@Override
	public Context build(Context context) {

		junctions = new ArrayList<Junction>();

		// User defined parameters
```



```
		final Parameters params = RunEnvironment.getInstance().getParame
ters();
		vehCount = ((Integer) params.getValue(Constants.NUM_VEHICLES)).i
ntValue();
		usesTrafficLights = ((boolean) params.getValue("usesTrafficLights"));
		inputShapefile = Constants.MAPS_FOLDER + ((String) params.getVal
ue("inputShapefile"));

		// GIS projection holds real position of vehicles
		createGISGeography(context);

		// Road network topology holds logical position of vehicles
		createRoadNetwork(context);

		// load user defined GIS shapefile to populate
		// both GIS and road network projections
		File f = new File(inputShapefile);
		if (!f.exists() && f.isDirectory()) {
			System.out.println("File Not Found!");
			return null;
		}
		loadShapefile(inputShapefile, context, geography, network);

		// set some default data for each junction in the topology
		// and appropriate position of traffic lights if needed.
		initJunctions(context);

		// Environment is all set up at this point.
		// Generate some vehicles using user parameters
		generateVehicles(context);

		return context;
	}

	/**
	 * Methods uses GeographyFactory to create
	 * the GIS geography projection, where all agents are displayed
	 *
	 * @param context
	 */
	@SuppressWarnings({ "rawtypes", "unchecked" })
	private void createGISGeography(Context context) {
		// To store GIS roads
		GeographyParameters geoParams = new GeographyParameters();
		geography = GeographyFactoryFinder.createGeographyFactory(null)
				.createGeography("Geography", context, geoParams)
;
	}

	/**
	 * Creates the road network projection which will hold
	 * the road network topology. Used by vehicles to select
	 * routes, and to know their direction of travel.
	 *
	 * @param context
	 */
	@SuppressWarnings({ "unchecked", "rawtypes" })
	private void createRoadNetwork(Context context) {
		NetworkBuilder<Object> roadBuilder = new NetworkBuilder<Object>(
"road network", context, true);
		roadBuilder.buildNetwork();
		network = (Network<Junction>) context.getProjection("road network")
;
	}

	/**
	 * Loads roads and junctions for a shapefile in both the
	 * GIS geograohy and road network. Finally, it
	 * adds them as agents to the Signal Green's context.
```

```
         *
         * @param filename relative path of shapefile
         * @param context
         * @param geography the GIS geography
         */
        @SuppressWarnings({ "unchecked", "unused" })
        private void loadShapefile(String filename, Context context, Geography g
eography, Network<Junction> network) {
                // used to create junctions on the gis projection
                GeometryFactory geomFac = new GeometryFactory();

                // read in shapefile
                URL url = null;
                try {
                        url = new File(filename).toURL();
                } catch (MalformedURLException e1) {
                        e1.printStackTrace();
                }

                List<SimpleFeature> features = new ArrayList<SimpleFeature>();

                // Try to load the shapefile
                SimpleFeatureIterator fiter = null;
                ShapefileDataStore store = null;
                store = new ShapefileDataStore(url);

                try {
                        fiter = store.getFeatureSource().getFeatures().features(
);
                        while(fiter.hasNext()){
                                features.add(fiter.next());
                        }
                } catch (IOException e) {
                        e.printStackTrace();
                }
                finally{
                        fiter.close();
                        store.dispose();
                }

                // tmp map to understand which junctions will be actually traffi
c lights
                // or stop signs. Integer will be > 2, ie. junction has at least
 3 roads.
                // We will also skip duplicate coordinates as we do not wand dup
licate junctions.
                Map<Coordinate, Integer> map = new HashMap<Coordinate, Integer>(
);

                // For each feature in the shapefile
                for (SimpleFeature feature : features) {
                        Geometry geom = (Geometry) feature.getDefaultGeometry();
                        Object agent = null;

                        // take into account MultiLineString shapes for Road obj
ects
                        if (geom instanceof MultiLineString){
                                MultiLineString line = (MultiLineString)feature.
getDefaultGeometry();
                                geom = (LineString) line.getGeometryN(0);
                                // get first and last, which are the junctions t
o create
                                Coordinate[] c = geom.getCoordinates();
                                Coordinate c1 = c[0]; // First coordinate
                Coordinate c2 = c[geom.getNumPoints() - 1]; // Last coordinate


                // 1. initial/end Coordinate already found
                // 2. initial/end Coordinate not found yet
```



```
                if (map.containsKey(c1)) {
                        map.put(c1, map.get(c1) + 1);
                }
                else {
                    map.put(c1, 0);
                }

                if (map.containsKey(c2)) {
                        map.put(c2, map.get(c2) + 1);
                }
                else {
                        map.put(c2, 0);
                }
                        }
                }

                // contains a list of Junctions, so that we do not create duplic
ate Junctions.
                // This happens whenever two roads meet in a Junction.
                Map<Coordinate, Junction> cache = new HashMap<Coordinate, Juncti
on>();

                // now create the junctions
                for (Map.Entry<Coordinate, Integer> entry : map.entrySet()) {

                        Junction j;
                    Coordinate c = entry.getKey();
                    Integer nRoads = entry.getValue();

                    if (nRoads > 1) {
                        // this junction needs special traffic management policy
                        // such as traffic lights or give way signs because it
                        // has more than two roads.
                        if (this.usesTrafficLights == true) {
                                j = new TrafficLight(this.network, this.geograph
y);
                        }
                        else { // implement here all policies
                                j = new GiveWaySign(this.network, this.geography
);
                        }
                    }
                    else {
                        // generic junction
                        j = new Junction(this.network, this.geography);
                    }
                    // put Junction in the GIS projection
                    j.setCoords(c);
                    cache.put(c, j);
            context.add(j);
            Point p = geomFac.createPoint(c);
            geography.move(j, p);
            junctions.add(j);
                }

                // Now we have all Junctions created, but we need to
                // create the network topology. We do this by iterating
                // through each road in the shapefile, and create a network
                // edge for each road.
                for (SimpleFeature feature : features) {
                        Geometry geom = (Geometry) feature.getDefaultGeometry();
                        Object agent = null;

                        // if shape is MultiLineString, create a Road object
                        if (geom instanceof MultiLineString){
                                MultiLineString line = (MultiLineString)feature.
getDefaultGeometry();
                                geom = (LineString) line.getGeometryN(0);
```

```
                                // Get attributes and assign them to the agent
                                // attributes depend on the shapefile attributes
.
                                String name = (String)feature.getAttribute("LNA
ME");
                                agent = new Road(name);

                                // road segment start and end coordinate
                                Coordinate[] c = geom.getCoordinates();
                                Coordinate c1 = c[0]; // First coordinate
                Coordinate c2 = c[geom.getNumPoints() - 1]; // Last coordinate

                addLanes(c1, c2, geography, context);

                Junction j1 = cache.get(c1);
                Junction j2 = cache.get(c2);

                // set road data
                double weight = Utils.distance(c1, c2, geography);
                ((Road) agent).setLength(weight);
                // weight is adjusted based on the type of road
                weight = weight / ((int) feature.getAttribute("THRULANES") + 1)
;


                        RepastEdge<Junction> re1 = network.addEdge(j1, j2, weigh
t);
                        RepastEdge<Junction> re2 = network.addEdge(j2, j1, weigh
t);


                j1.addJunction(j2);
                j2.addJunction(j1);

                // Road-RepastEdge mapping for lane management use
                this.roads.put(re1, (Road) agent);
                this.roads.put(re2, (Road) agent);

                                // put road in the GIS projection
                                // 1. show the road as it is in the GIS shapefil
e <-- many details shown
                                // context.add(agent);
                                // geography.move(agent, geom);


                                // 2. or display a simplified version of the map
                                // in this case need to uncomment previous block
 of code
                                Coordinate[] coords = new Coordinate[] { c1, c2
};
                                LineString ls = geomFac.createLineString(coords)
;
                                geom = (LineString)ls.getGeometryN(0);
                                context.add(ls);
                                geography.move(agent, geom);
                        }
                }
        }


        /**
         * Creates two lanes each side of a road.
         *
         * @param c1 start of lane
         * @param c2 end of lane
         * @param geography
         * @param context
         */
        @SuppressWarnings("unchecked")
        private void addLanes(Coordinate c1, Coordinate c2, Geography geography,
 Context context) {
```



```
                GeometryFactory geomFac = new GeometryFactory();
        double azimuth = Utils.getAzimuth(c1, c2, geography);

        // generate coordinates for creating lanes
        Coordinate dest1[] = Utils.createCoordsFromCoordAndAngle(c1, azimuth, Co
nstants.DIST_LANE, geography);
        Coordinate dest2[] = Utils.createCoordsFromCoordAndAngle(c2, azimuth, Co
nstants.DIST_LANE, geography);

                // create set of auxiliary lanes, two for each side of the road
        for (int i = 0; i < 4; i++) {
                // Left
                CoordinateAgent cLeft = new CoordinateAgent();
                context.add(cLeft);
                Point pLeft = geomFac.createPoint(dest1[i]);
                geography.move(cLeft, pLeft);

                // Right
                CoordinateAgent cRight = new CoordinateAgent();
                context.add(cRight);
                Point pRight = geomFac.createPoint(dest2[i]);
                geography.move(cRight, pRight);

                // Lane
                Coordinate[] coords = new Coordinate[] { dest1[i], dest2[i] };
                LineString ls = geomFac.createLineString(coords);
                Geometry geom = (LineString) ls.getGeometryN(0);
                context.add(ls);
                geography.move(new LaneAgent(), geom);
        }

        }

        /**
         * 1. Initialises queues for every junction. Each junction holds
         * a list of incoming vehicles for each in-edge road segment.
         * 2. Puts Lights of TrafficLights on GIS projection if needed.
         * To be called only after all junctions have been loaded from
         * the GIS shapefile.
         */
        @SuppressWarnings("unchecked")
        private void initJunctions(Context context) {
                Iterator<Junction> it = this.junctions.iterator();
                while (it.hasNext()) {
                        Junction j = it.next();
                        List<Junction> l = j.getJunctions();

                        // initialise vehicle queues
                        Iterator<Junction> itmap = l.iterator();
                        while (itmap.hasNext()) {
//                              j.vehicles.put(itmap.next(), new PriorityDeque<V
ehicle>());
                                j.vehicles.put(itmap.next(), new PriorityBlockin
gDeque<Vehicle>());
                        }

                        // position Lights of TrafficLights if needed
                        if (j instanceof TrafficLight) {
                                Map<Junction, Light> lights = ((TrafficLight) j)
.getLights();
                                for (Map.Entry<Junction, Light> e : lights.entry
Set()) {
                                        Light light = e.getValue();
                                        Coordinate coords = e.getKey().getCoords
();
                                        Coordinate currPos = geography.getGeomet
ry(j).getCoordinate();
                                        double angle = Utils.getAngle(currPos, c
oords, geography);
```

```
                                        context.add(light);
                                        GeometryFactory geomFac = new GeometryFa
ctory();
                                        Point p = geomFac.createPoint(currPos);
                                        geography.move(light, p);
                                        geography.moveByVector(light, Constants.
DIST_LIGHTS, angle);
                                }

                        }
                }
        }

        /**
         * Generates vehicles using user defined params.
         * Vehicles are added to the context at random Junctions.
         *
         * @param context
         */
        private void generateVehicles(Context context) {
                Random rand = new Random();
                Vehicle vehicles[] = new Vehicle[vehCount];
                for (int i = 0; i < vehCount; i++) {
                        // assign random speed to vehicles
                        int maxSpeed = (Constants.speed[rand.nextInt(Constants.s
peed.length)]);
                        Vehicle vehicle;
                        if (maxSpeed < 100) {
                                // truck
                                vehicle = new TruckVehicle(network, geography, r
oads, maxSpeed);
                        }
                        else {
                                vehicle = new CarVehicle(network, geography, roa
ds, maxSpeed);
                        }

                        context.add(vehicle);
                        Junction origin = junctions.get(rand.nextInt(junctions.s
ize()));
                        GeometryFactory geomFac = new GeometryFactory();
                        Point p = geomFac.createPoint(origin.getCoords());
            geography.move(vehicle, p);
            vehicle.initVehicle(origin);
                }
        }
}
```

```
package signalGreen;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Queue;
import java.awt.*;

import com.vividsolutions.jts.geom.Coordinate;
import com.vividsolutions.jts.geom.GeometryFactory;
import com.vividsolutions.jts.geom.Point;

import repast.simphony.space.continuous.ContinuousSpace;
import repast.simphony.space.gis.Geography;
import repast.simphony.space.graph.Network;
import repast.simphony.space.grid.Grid;
import repast.simphony.context.Context;
import repast.simphony.engine.schedule.ScheduledMethod;
import signalGreen.Constants.Signal;

/**
 * TrafficLight object is a subclass of the Junction object. It has
 * a traffic light dedicated to each lane linked to the Junction.
 *
 * It contains the Step() method that performs the light changing
 * algorithm.
 *
 * @author Waqar, Adeela
 *
 */
public class TrafficLight extends Junction{

	// List of lights for each lane linking from a Junction
	// to this Junction
	private Map<Junction, Light> lights;

	/**
	 * @param network
	 */
	public TrafficLight(Network<Junction> network, Geography geography) {
		super(network, geography);
		this.lights = new HashMap<Junction, Light>();
	}

	/**
	 * Add traffic light for new lane to given Junction. If it is the
	 * first light, set state to GREEN, else RED.
	 *
	 * @see signalGreen.Junction#addJunction(signalGreen.Junction)
	 */
	@Override
	public void addJunction(Junction junc) {
		super.addJunction(junc);

		Light light = new Light(Signal.RED);

		if(lights.size() == 0)  {
			light.setSignal(Signal.GREEN);
		}

		lights.put(junc, light);
	}

	/**
	 * Remove traffic light for lane to given Junction.
	 *
	 * @see signalGreen.Junction#removeLane(signalGreen.Junction, boolean)
	 */
```



```
	@Override
	public void removeLane(Junction junc, boolean out) {
		lights.remove(getJunctions().indexOf(junc));
		super.removeLane(junc, out);
	}

	/**
	 * Remove all traffic lights.
	 *
	 * @see signalGreen.Junction#removeAllLanes()
	 */
	@Override
	public void removeAllLanes() {
		lights.clear();
		super.removeAllLanes();
	}

	/**
	 * Step() method to perform light changing algorithm with the
	 * scheduled method annotation by Repast.
	 */
	@ScheduledMethod(start = 1, interval = 25)
	public void step() {
		// TODO find optimal interval
		if (lights.size() != 0) {
			toggleNextLight();
		}
		// DEBUG
		// debugLights();
	}

	public void debugLights() {
		for (Map.Entry<Junction, Light> entry : lights.entrySet()) {
			// System.out.println("key=" + entry.getKey() + ", value=" +
entry.getValue());
			Light l = entry.getValue();
			Junction j = entry.getKey();
			if (l.getSignal() == Signal.GREEN)
			    System.out.println(l.toString() + ":GREEN");
			if (l.getSignal() == Signal.AMBER)
			    System.out.println(l.toString() + ":AMBER");
			if (l.getSignal() == Signal.RED)
			    System.out.println(l.toString() + ":RED");
		}
	}

	/**
	 * Get a List of all Lights, their indexes match junctions indexes.
	 * @return lights
	 */
	public Map<Junction, Light> getLights() {
		return lights;
	}

	/**
	 * Toggle to the next traffic light.
	 */
	public void toggleNextLight() {
		int lastGreenLightIndex = 0;

		List<Light> l = new ArrayList<Light>(lights.values());

		for (Light light : l) {
			if (light.getSignal() == Signal.GREEN) {
				lastGreenLightIndex = l.indexOf(light);
			}
		}

		l.get(lastGreenLightIndex).toggleSignal();
```

```
                if (l.size() - 1 == lastGreenLightIndex) {
                        l.get(0).toggleSignal();

                } else {
                        l.get(lastGreenLightIndex + 1).toggleSignal();
                }

        }
}
```

```
package signalGreen;

import java.util.Map;

import repast.simphony.space.gis.Geography;
import repast.simphony.space.graph.Network;
import repast.simphony.space.graph.RepastEdge;

/**
 * Trucks are the slowest vehicles in Signal Green.
 * Max speed defaults to 80 Km/h.
 *
 * @author Yoann
 *
 */
public class TruckVehicle extends Vehicle {

        /**
         * Constructor for trcuk vehicle.
         *
         * @param network
         * @param geography
         * @param roads
         * @param maxVelocity
         */
        public TruckVehicle(Network<Junction> network, Geography geography,
                        Map<RepastEdge<Junction>, Road> roads, int maxVelocity)
{
                super(network, geography, roads, maxVelocity);
                setMaxVelocity(Constants.TRUCK_DEFAULT_MAX_VELOCITY);
        }

}
```

```
package signalGreen;

import java.awt.geom.Point2D;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Random;

import org.geotools.referencing.GeodeticCalculator;
import repast.simphony.space.gis.Geography;
import repast.simphony.space.graph.Network;
import repast.simphony.space.graph.RepastEdge;

import com.vividsolutions.jts.geom.Coordinate;

/**
 * Most utility functions are used for geographic
 * calculations, such as distance between coordinates
 * or azimuth of a road segment.
 *
 * @author Signal Green team
 *
 */
public class Utils {

	/**
	 * Method selects a random Junction from the road network.
	 * Vehicles call this method every time they need a new destination.
	 * @param roadNetwork
	 * @return junction
	 */
	public static Junction getRandJunction(Network<Junction> roadNetwork) {
		// get all edges and put them into list for random access
		Iterator<RepastEdge<Junction>> it = roadNetwork.getEdges().itera
tor();
		List<RepastEdge<Junction>> l = new ArrayList<RepastEdge<Junction
>>();
		while(it.hasNext()) {
			l.add(it.next());
		}

		if (l.size() > 0) {
			Random rand = new Random();
			int index = rand.nextInt(l.size());
			// we know that each edge has a source and target Juncti
on
			Junction j = (Junction) l.get(index).getTarget();
			// System.out.println("Random Junction: " + j.toString()
);
			return j;
		}
		return null;
	}

	public static void debugCoordinate(Coordinate c) {
		System.out.println("Coordinate: " + c.toString() + ", x: "
				+ c.x + ", y: " + c.y);
	}

	/**
	 * Method iterates though a List of Junctions to find most popular one.
	 * This is the Junction with the most number of cars.
	 * @param junctions List of junctions
	 * @return Popular junction
	 */
	public static Junction getPopularJunction(List<Junction> junctions)
	{
		if (junctions == null) return null;
```



```
		Junction popularJunc = junctions.get(0);

		for (int i = 0; i < junctions.size(); i++) {
			if(junctions.get(i).vehicles.size() > popularJunc.vehicl
es.size()) {
				popularJunc = junctions.get(i);
			}
		}
		return popularJunc;
	}

    /**
     * Distance between two coordinates in metres
     *
     * @param c1 coordinate1
     * @param c2 coordinate2
     * @param g geography
     */
	public static double distance(Coordinate c1, Coordinate c2, Geography g)
 {
		GeodeticCalculator calculator = new GeodeticCalculator(g.getCRS());
		calculator.setStartingGeographicPoint(c1.x, c1.y);
		calculator.setDestinationGeographicPoint(c2.x, c2.y);
		return calculator.getOrthodromicDistance();
		}

		/**
		 * Returns the angle in radians given two coordinates.
		 * Radians to degrees conversion = angle * 2 * PI
		 *
		 * @param c1 first coord
		 * @param c2 second coord
		 * @param g geography
		 * @return angle in radians
		 */
		public static double getAngle(Coordinate c1, Coordinate c2, Geography g)
 {
		double angle = Math.toRadians(Utils.getAzimuth(c1, c2, g)); // Angle in
range -PI to PI
		// credits: https://code.google.com/p/repastcity/source/browse/branches/
sim_comp_sys_model/src/repastcity3/environment/Route.java
		// Need to transform azimuth (in range -180 -> 180 and where 0 points no
rth)
		// to standard mathematical (range 0 -> 360 and 90 points north)
		if (angle > 0 && angle < 0.5 * Math.PI) { // NE Quadrant
			angle = 0.5 * Math.PI - angle;
		} else if (angle >= 0.5 * Math.PI) { // SE Quadrant
			angle = (-angle) + 2.5 * Math.PI;
		} else if (angle < 0 && angle > -0.5 * Math.PI) { // NW Quadrant
			angle = (-1 * angle) + 0.5 * Math.PI;
		} else { // SW Quadrant
			angle = -angle + 0.5 * Math.PI;
		}
		return angle;
		}

		/**
		 * Given two coordinates it returns the azimuth.
		 * Azimuth = angle in range of (+-) 180 degrees
		 *
		 * @param c1 coordinate
		 * @param c2 coordinate
		 * @param g  geography
		 * @return azimuth in double precision
		 */
		public static double getAzimuth(Coordinate c1, Coordinate c2, Geography
g) {
		GeodeticCalculator calculator = new GeodeticCalculator(g.getCRS());
		calculator.setStartingGeographicPoint(c1.x, c1.y);
```

```
        calculator.setDestinationGeographicPoint(c2.x, c2.y);
        return calculator.getAzimuth();
        }

        /**
         * Returns two coordinates that are perpendicular (+−90 degrees) to a gi
ven angle
         * from a starting point on Earth, at a given distance from c.
         * Used for creating Lanes on the left and right of a Road.
         *
         * @param c the coordinate
         * @param azimuth
         * @param distance
         * @param g the geography
         * @return array of coordinates
         */
        public static Coordinate[] createCoordsFromCoordAndAngle(Coordinate c, d
ouble azimuth, double distance, Geography g) {
        // on the GIS display the do not look 90 degrees because
                // GIS is actually a sphere (the Earth..)
        double angle = Math.toRadians(azimuth);
        double a1, a2;

        // LEFT LANES in respect to road
        // −90 degrees angle
        if (angle > 0 && angle < 0.5 * Math.PI) { // NE Quadrant
                a1 = angle − 0.5 * Math.PI;
            } else if (angle >= 0.5 * Math.PI) { // SE Quadrant
                a1 = angle − 0.5 * Math.PI;
            } else if (angle < 0 && angle > −0.5 * Math.PI) { // NW Quadrant
                a1 = angle − 0.5 * Math.PI;
            } else { // SW Quadrant
                a1 = angle + 1.5 * Math.PI;
            }

        // RIGHT LANES
        // +90 degrees angle
        if (angle > 0 && angle < 0.5 * Math.PI) { // NE Quadrant
                a2 = angle + 0.5 * Math.PI;
            } else if (angle >= 0.5 * Math.PI) { // SE Quadrant
                a2 = angle − 1.5 * Math.PI;
            } else if (angle < 0 && angle > −0.5 * Math.PI) { // NW Quadrant
                a2 = angle + 0.5 * Math.PI;
            } else { // SW Quadrant
                a2 = angle + 0.5 * Math.PI;
            }

        // convert back to azimuth
        a1 = Math.toDegrees(a1);
        a2 = Math.toDegrees(a2);

        GeodeticCalculator calculator = new GeodeticCalculator(g.getCRS());
        calculator.setStartingGeographicPoint(c.x, c.y);

        /*
         * Generate set of coordinates as follows using either j1 or j2 as
         * starting coordinate c:
         *
         *                                      j1
j2
             *  lft outer               o−−−−−−−−−−−−−−−−−−>o
             *  lft inner               o−−−−−−−−−−−−−−−−−−>o
             *  road topology   o<==================>o
             *  rgt inner               o<−−−−−−−−−−−−−−−−−−o
             *  rgt outer               o<−−−−−−−−−−−−−−−−−−o
             *
             */

        // Left lanes
```



```
        calculator.setDirection(a1, (distance * 1.5));
            Point2D dest1 = calculator.getDestinationGeographicPoint();
        calculator.setDirection(a1, distance * 0.5);
            Point2D dest2 = calculator.getDestinationGeographicPoint();

            // Right lanes
            calculator.setDirection(a2, distance * 0.5);
            Point2D dest3 = calculator.getDestinationGeographicPoint();
            calculator.setDirection(a2, (distance * 1.5));
            Point2D dest4 = calculator.getDestinationGeographicPoint();

            // return coords
            Coordinate[] coords = {
                        new Coordinate(dest1.getX(), dest1.getY()),     // lft o
uter
                        new Coordinate(dest2.getX(), dest2.getY()),     // lft i
nner
                        new Coordinate(dest3.getX(), dest3.getY()),     // rgt i
nner
                        new Coordinate(dest4.getX(), dest4.getY())      // rgt o
uter
            };
                return coords;
        }

        /**
         * Returns the angle in degrees
         *
         * @param c1
         * @param c2
         * @param g
         * @return angle
         */
        public static double getAngleDeg(Coordinate c1, Coordinate c2,
                        Geography g) {
                return Math.toDegrees(Utils.getAngle(c1, c2, g));
        }

        /**
         * Returns a correct angle for displaying graphics
         * correctly on the GIS projection. Angle of orientation
         * of graphics must be computed from grid north as
         * opposed to true north, which may not point to the
         * top of the display.
         *
         * @param c1
         * @param c2
         * @param geography
         * @return azimuth
         */
        public static double getAngleForIcons(Coordinate c1, Coordinate c2, Geog
raphy g) {
                // angle must be computed from grid north (NOT true north of pro
jection)
                double atan = Math.atan2(c1.y − c2.y, c1.x − c2.x) * 180.0 / Mat
h.PI;
        double azimuth = (450.0 − atan) % 360;
                return azimuth;
        }


}
```

```
package signalGreen;

import java.util.*;

import com.vividsolutions.jts.geom.Coordinate;
import com.vividsolutions.jts.geom.GeometryFactory;
import com.vividsolutions.jts.geom.Point;

import repast.simphony.engine.schedule.ScheduledMethod;
import repast.simphony.space.gis.Geography;
import repast.simphony.space.graph.Network;
import repast.simphony.space.graph.RepastEdge;
import repast.simphony.space.graph.ShortestPath;
import signalGreen.Constants.*;

/***
 * Generic class for vehicles of the Traffic Simulator.<br />
 * Cars, ambulances, trucks are subclasses of Vehicle,
 * and have special behaviour such as cars having reckless or cautious drivers.
 * Vehicle implements the Comparable interface because they are
 * held in queues of vehicles for each road segment, to know their ordinal
 * position. Thus, Vehicles are compared according to their distance
 * to the next junction.
 *
 * @author Yoann
 *
 */
public class Vehicle extends GisAgent implements Comparable<Vehicle> {

	// position of vehicle in the GIS projection
	private Coordinate realPos; // This is the real position for display pur
pose only
	private Coordinate networkPos; // logical position used to do all comput
ations

	// holds mapping between repast edges and actual GIS roads
	private Map<RepastEdge<Junction>, Road> roads;

	private int velocity;
	private int maxVelocity;
	// displacement is used by other vehicles: they can compare it to their
displ.
	// and stop, slow down or accelerate accordingly.
	private double displacement;

	// Simulation is based on Origin Destination pattern.
	// Vehicles have an origin (x, y) starting point
	// and a destination point which is randomly reset to another
	// destination as it reaches it.
	private Junction origin;
	private Junction next;
	private Junction destination;
	private Lane lane;

	// holds the full path from origin to destination
	// each edge of the route is a directed link between Junctions
	private List<RepastEdge<Junction>> vehicleRoute;

	private double angle;

	/**
	 * Generic Vehicle constructor.
	 *
	 * @param network
	 * @param geography
	 * @param roads
	 * @param maxVelocity
	 */
	public Vehicle(Network<Junction> network,
```



```
			Geography geography, Map<RepastEdge<Junction>, Road> roa
ds, int maxVelocity)
	{
		super(network, geography);
		this.roads = roads;
		this.velocity = 0;
		this.maxVelocity = maxVelocity;
	}

	@Override
	public boolean equals(Object obj) {
	if (!(obj instanceof Vehicle)) {
	    return false;
	}
	Vehicle v = (Vehicle) obj;
	return  (v.getID() == this.getID());
	}

	/**
	 * Initialises Vehicle: set Origin, find random Destination
	 * and compute best route.
	 */
	public void initVehicle(Junction origin) {
		// System.out.println("*** initVehicle: " + this.toString());
		// set origin and destination of vehicle
		this.origin = origin;
		this.destination = Utils.getRandJunction(getNetwork()); // may r
eturn null!
		this.lane = Lane.OUTER; // always start outer

		// check if we have't chosen same origin and destination
		// unlikely to happen but...
		ifVehicleAtDestination();

		// get best route from origin to destination
		findBestRoute();

		// set the next junction, so vehicle knows what is the next step
 on the route
		this.next = this.getNextJunctionRoute();

		// set positions of vehicle
		this.networkPos = origin.getCoords();
		this.realPos = this.getRealPosFromNetworkPos(this.lane);
		moveTo(realPos);

		// register vehicle to next junction
		this.next.enqueueVehicle(this.origin, this);
		this.angle = getAngle();
	}

	/**
	 * step() is called at each iteration of the
	 * simulation, starting from iteration 1.
	 * Vehicle behavior takes place here.
	 */
	@ScheduledMethod(start = 1, interval = 1)
	public void step() {
		Vehicle v; // vehicle ahead
		double tmpDisplacement;

		// following happens only when network topology contains more th
an one graph.
		// It is the case when a vehicle tries to reach a destination on
 the other graph.
		if (this.vehicleRoute.size() == 0) {
			System.out.println("Vehicle is stuck in impasse. Cannot move...");
		}
```

```
                // get location of next Junction along the route
                this.next = this.getNextJunctionRoute();

                // compute how many meters we would like to move
                tmpDisplacement = this.computeDisplacement();

                // selects the lane this vehicle wants to go to
                v = laneSelection(tmpDisplacement);

                // find optimal displacement checking the gap between vehicles
                this.displacement = gapAcceptance(v, tmpDisplacement);
                if (this.displacement == 0) {
                        return; // no need to perform displacement
                }

                // move the vehicle on the GIS geography on the selected
                // lane using optimal displacement
                executeDisplacement(this.displacement);
        }

        /**
         * Method selects the best lane (incentive) by checking
         * for vehicles ahead of current vehicle V
         * on both INNER and OUTER lanes. It then checks if
         * there are no vehicles approaching on that lane
         * with speed greater than V's speed.
         *
         * @param displacement
         * @return vehicle ahead on selected lane
         */
        private Vehicle laneSelection(double tmpDisplacement) {
                Vehicle v = null;
                Lane targetLane = this.lane;
                // see if there is a vehicle ahead within vision range
                Vehicle[] veh = getVehiclesAhead(tmpDisplacement + Constants.DIS
T_VEHICLES);
                v = null; // reset vehicle ahead

                // Vehicle now decides if he wants to change lane or not.
                // case: we are on the outer lane
                if (this.lane == Lane.OUTER) {
                        // if there is a vehicle ahead on the OUTER but none on
 the INNER
                        // we can overtake safely, but only if outer's lane vehi
cle's speed < ours
                        // and their speed != 0 --> causes horrible overtaking a
t traffic lights...
                        if ((veh[0] != null)
                                        && (this.getVelocity() > veh[0].getVeloc
ity())
                                        && (veh[0].getVelocity() != 0)) {
                                if (veh[1] == null) {
                                        // move to inner lane
                                        targetLane = Lane.INNER;

                                        v = veh[1]; // set next vehicle to the o
ne on the INNER lane
                                }
                        }
                        else {
                                // we stay on the OUTER lane
                                targetLane = Lane.OUTER;
                                v = veh[0];
                        }

                }
                // case: we are on the INNER so if there are no cars on the
                // OUTER lane we might want to go back to the OUTER lane
                // if it is clear
```



```
                else {
                        // check if the outer lane is clear, meaning
                        // no vehicles behind are approaching
                        Vehicle vBehind = getVehiclesBehind(tmpDisplacement + Co
nstants.DIST_VEHICLES)[0];
                        if ((veh[0] == null) && (vBehind == null)) {
                                targetLane = Lane.OUTER;
                                v = veh[0];
                        }
                        else {
                                targetLane = Lane.INNER;
                                v = veh[1];
                        }
                }
                this.lane = targetLane;
                return v;
        }

        /**
         * Method checks if there is enough space
         * between the current vehicle and the
         * vehicle ahead (road is safe), in order
         * to execute a displacement.
         * Vehicle tries to accelerate or slow down
         * according to the outcome of the gap acceptance.
         * In that case displacement is recomputed.
         *
         * @param v the vehicle
         * @param tmpDisplacement
         * @return optimal displacement
         */
        private double gapAcceptance(Vehicle v, double tmpDisplacement) {
                // check vehicle ahead's displacement to know how to
                // adjust velocity and displacement
                if (v != null) {
                        // distance between current vehicle and leader
                        // check if we need to stop: vehicle ahead is close enou
gh and stopped
                        if ((v.getDisplacement() == 0) || v.getVelocity() == 0)
{
                                this.velocity = 0;
                                return 0; // no need to perform the displacement
                        }

                        // adjust to optimal velocity/displacement

                        while ((tmpDisplacement + Constants.DIST_VEHICLES) >= (v
.getDisplacement() /* + vDistance */)) {

                                this.slowDown();
                                tmpDisplacement = this.computeDisplacement();

                                // manage limit cases
                                if (this.velocity == 0) {
                                        return 0;
                                }
                        }
                }
                else {
                        // no vehicles, accelerate if we are allowed to
                        this.accelerate();
                }
                return tmpDisplacement;
        }

        /**
         * Moves a vehicle on the GIS geography
         * towards the next junction. If the vehicle
         * can make it all to way to it, it drives
```

```
         * the remaining displacement towards the new
         * next junction. Traffic management policies
         * are evaluated if the vehicle is in proximity
         * to special types of junctions.
         *
         * @param displacement in meters
         */
        private void executeDisplacement(double tmpDisplacement) {
                boolean mustStopVehicle = false;
                // how far is the next junction?
                double juncDist = Utils.distance(getNetworkPos(), next.getCoords
(), getGeography());

                // if there is a junction that needs traffic
                // management policies we adjust the distance to it
                // so that we stop before the jam
                if ((next instanceof TrafficLight)
                                || (next instanceof GiveWaySign)){
                        juncDist = juncDist - Constants.DIST_LIGHTS;
                }

                // now update position of vehicle on GIS display:
                // might have changed because of change lane algorithm
                this.realPos = this.getRealPosFromNetworkPos(lane);
                this.moveTo(realPos);

                // following algorithm is for moving vehicles along
                // the road network towards the next Junction.
                do {
                        // we cannot reach the next junction on the road network
                        // because it is too far... just move towards it.
                        if (tmpDisplacement < juncDist) {
                                moveTowards(next.getCoords(), tmpDisplacement);
                                tmpDisplacement = 0;
                        }
                        // we are going to move more than
                        // the next junction
                        else if (tmpDisplacement >= juncDist) {
                                // check traffic policies
                                mustStopVehicle = evaluateTrafficManagementPolic
ies();
                                if (mustStopVehicle == true) {
                                        return;
                                }

                                // road is clear: move to next junction
                                // then we keep moving towards the next one.
                                moveTowards(next.getCoords(), juncDist);
                                tmpDisplacement = tmpDisplacement - juncDist;
                                displacement = tmpDisplacement;
                                removeCurrentRoadSegmentFromRoute();
                                // recompute distance towards updated next junct
ion
                                juncDist = Utils.distance(getNetworkPos(), next.
getCoords(), getGeography());
                        }

                        // DEBUG
                        // debugRoute();

                        // This makes vehicles moving indefinitely:
                        // if they reached their destination, pick a new random
destination
                        ifVehicleAtDestination();

                } while (tmpDisplacement > 0); // keep iterating until the whole
 displacement has been covered

                // update queue held by junction to know order of vehicles on cu
```



```
rrent road segment
                next.reorderVehicle(origin, this);
                // update graphic's angle to match with road's horizontal axis
                this.angle = Utils.getAngleDeg(origin.getCoords(), next.getCoord
s(), getGeography());
        }

        /**
         * Evaluates traffic management policies and
         * decides if a vehicle must stop or continue
         * in proximity of a special instance of junction.
         *
         * @return true if vehicle must stop
         */
        private boolean evaluateTrafficManagementPolicies() {
                // Traffic Lights
                if (next instanceof TrafficLight) {
                        Light light = ((TrafficLight) next).getLights().get(orig
in);
                        if ((light.getSignal() == Constants.Signal.RED)) {
                                // easy case :)
                                this.setVelocity(0);
                                this.displacement = 0;
                                return true; // red t. light, must stop
                        }
                }

                // Give Way Signs
                if (next instanceof GiveWaySign) {
                        Vehicle closest = this;
                        // iterate through each queue of incoming vehicles
                        // and figure out if we are the closest vehicle to it
                        for (Map.Entry<Junction, PriorityBlockingDeque<Vehicle>>
 entry
                                        : next.vehicles.entrySet())
                        {
                                Vehicle v = entry.getValue().peek();
                                if ((v != null) && (v.getDistanceToNextJunction(
) <= closest.getDistanceToNextJunction())) {
                                        closest = v;
                                }
                                // check if we are NOT the closest, meaning
                                // we have to wait at the intersection
                                if ( ! closest.equals(this)) {
                                        this.setVelocity(0);
                                        this.displacement = 0;
                                        return true;
                                }
                        }
                }

                return false; // vehicle can keep moving
        }

        /**
         * Removes the last road segment that a Vehicle has just traveled
         * and updates the current route.
         */
        private void removeCurrentRoadSegmentFromRoute() {
                // current next junction (soon the origin) thinks we are
                // on his road segment. Need to dequeue vehicle from vehicle lis
t.
                this.next.dequeueVehicle(origin, this);

                if (this.vehicleRoute.size() <= 1) { // origin == destination?
                        // reset vehicle route
                        initVehicle(this.next);
                }
                else {
```

```
                        // Move to next road segment
                        this.origin = this.next;
                        // remove current road segment from current route
                        this.vehicleRoute.remove(0);
                        this.next = this.getNextJunctionRoute();
                        // tell next junction this vehicle is on his way
                        this.next.enqueueVehicle(this.origin, this);
                        // update position
                        this.networkPos = origin.getCoords();
                        this.realPos = this.getRealPosFromNetworkPos(this.lane);
                        moveTo(realPos);
                }
        }

        /**
         * Picks a new random destination if the vehicle has reach his current d
est.
         */
        private void ifVehicleAtDestination() {
                boolean isAtDestination = false;
                // check if Vehicle has reached destination
                while (this.origin.equals(this.destination)) {
                        isAtDestination = true;
                        // choose new random destination
                        this.destination = Utils.getRandJunction(getNetwork());
                }
                // update best route
                if (isAtDestination == true) {
                        findBestRoute();
                }
        }

        /**
         * Method uses origin and destination Junctions to find the best
         * path, using SPF algorithm.
         */
        private synchronized void findBestRoute() {
                ShortestPath<Junction> p = new ShortestPath<Junction>(getNetwork
());
                p.finalize();
                this.vehicleRoute = p.getPath(this.origin, this.destination);

                if (vehicleRoute.size() == 0) {
                        System.out.println("No route found because vehicle is on an impasse..."
                                        + "\nMake sure Road network has no impasses, ie. have al
ways "
                                        + "two-way roads.");
                }
                else {
                        this.next = this.getNextJunctionRoute();
                        // debugRoute();
                }
        }

        /**
         * Prints out Vehicle route data.
         */
        private void debugRoute() {
                System.out.println("\n***"
                                + "\nVehicle ID: " + this.getID()
                                + "\nOrigin: " + this.origin
                                + "\nNext Junction: " + this.next
                                + "\nDestination" + this.destination
                                + "\nCurrent route:\n");
                Iterator<RepastEdge<Junction>> it = vehicleRoute.iterator();
                while (it.hasNext()) {
                        RepastEdge<Junction> e = it.next();
                        System.out.println("\tOut Edge: " + e.getSource().toString(
)
```



```
                                        + " -> " + e.getTarget().toString());
                }
                System.out.println("\n***\n");
        }

        /**
         * Method finds the closest vehicle in vision range
         * ahead of the current vehicle, if any.<br />
         * Vision range distance is measured in meters, which
         * should vary depending on current speed of vehicle.
         * Usually this is the Vehicle displacement.<br />
         * Returns an array with size of 2 of vehicles ahead as follows:<br />
         * <code>v[0]</code> => Vehicle on <code>Lane.OUTER</code><br />
         * <code>v[1]</code> => Vehicle on <code>Lane.INNER</code>
         *
         * @see signalGreen.Vehicle#computeDisplacement()
         * @param x the vision range distance in meters
         * @return array of vehicles
         */
        private Vehicle[] getVehiclesAhead(double x) {

                Vehicle v[] = next.getNextVehicles(origin, this, true);

                // check if next vehicles are in vision range
                // Outer lane
                v[0] = validateVehicleWithinVisionRange(v[0], x);
                // Inner lane
                v[1] = validateVehicleWithinVisionRange(v[1], x);

                return v;
        }

        /**
         * Methods returns closest vehicles behind current vehicle.
         * @param x distance
         * @return array of vehicles
         * @see signalGreen.Vehicle#getVehiclesAhead(double)
         */
        private Vehicle[] getVehiclesBehind(double x) {

                Vehicle v[] = next.getNextVehicles(origin, this, false);

                // check if prev vehicles are in vision range
                // Outer lane
                v[0] = validateVehicleWithinVisionRange(v[0], x);

                // Inner lane
                v[1] = validateVehicleWithinVisionRange(v[1], x);

                return v;
        }

        /**
         * If vehicle is within vision range retuns a
         * reference to it, otherwise it is too far from
         * the current vehicle so we return null.
         *
         * @param v the vehicle
         * @param x the vision range
         * @return vehicle or null
         */
        private Vehicle validateVehicleWithinVisionRange(Vehicle v, double x) {
                if (v != null) {
                        // check if next vehicles are in vision range
                        Coordinate c = this.getNetworkPos(); // current position
                        Coordinate c1 = v.getNetworkPos();
                        double dist = Utils.distance(c, c1, getGeography());
                        if (dist >= x) {
                                v = null;
```

```
                        }
                }
                return v;
        }

        /**
         * Returns the real position on a vehicle on the GIS projection
         *
         * @return coordinate
         */
        public Coordinate getRealPos() {
                return realPos;
        }

        /**
         * Real position refers to the position
         * of a vehicle on a particular lane.
         *
         * @param realPos the real position of vehicle
         */
        public void setRealPos(Coordinate realPos) {
                this.realPos = realPos;
        }

        /**
         * Returns the logical position of a vehicle on the
         * road network topology, using real geodetic distances
         *
         * @return coordinate
         */
        public Coordinate getNetworkPos() {
                return networkPos;
        }

        public void setNetworkPos(Coordinate networkPos) {
                this.networkPos = networkPos;
        }

        /**
         * Moves a vehicle towards a given Coordinate.
         * Uses network position and then moves on the
         * real GIS projection.
         *
         * @param c Coordinate to move towards
         * @param x displacement in meters
         */
        @SuppressWarnings("unchecked")
        public void moveTowards(Coordinate c, double x)
        {

                double angle = Utils.getAngle(this.networkPos, c, getGeography()
);
                try {
                        getGeography().moveByVector(this, x, angle); // move age
nt
                }
                catch (IllegalArgumentException iae) {
                        System.out.println("Could not move vehicle for some reason.");
                        iae.printStackTrace();
                }

                // update positions
                realPos = getGeography().getGeometry(this).getCoordinate();
                this.networkPos = this.getNetworkPosFromRealPos(this.lane);
        }

        /**
         * Moves a vehicle to a given Coordinate.
         *
```



```
         * @param coordinate
         */
        @SuppressWarnings("unchecked")
        private void moveTo(Coordinate c) {
                GeometryFactory geomFac = new GeometryFactory();
                Point p = geomFac.createPoint(realPos);
                getGeography().move(this, p);
        }

        /**
         * Computes the displacement distance and adjusts
         * the velocity according to:<br />
         * 1. current velocity<br />
         * 2. max velocity<br />
         * Uses standard kinematics equations for this purpose.
         *
         * @return x the displacement in meters
         */
        private double computeDisplacement() {
                //      Equation to find displacement:
                //      x = v0 * t + 1/2 a * t^2
                //              Where:
                //              x = displacement
                //              v0 = initial velocity
                //              a = acceleration <-- add some constant values, t
he more acceleration, the more powerful. ex. trucks have smaller accel.
                //              t = time
                double accFactor = getAccelerationFactor();
                double x = Math.ceil(velocity + 0.5 * accFactor
                                * Constants.ACCELERATION * Math.pow(Constants.t,
 2));

                // adjust displacement for more realistic simulation
                x = x / Constants.CONV_RATIO_METERS;

                return x;
        }

        private double getAccelerationFactor() {
                if ((this instanceof CarVehicle) && (this.velocity <= Constants.
SLOW)) {
                        return Constants.CAR_SLOW_ACC;
                }
                else if (this instanceof CarVehicle) {
                        return Constants.CAR_FAST_ACC;
                }
                else if (this instanceof TruckVehicle) {
                        return Constants.TRUCK_ACC;
                }
                return Constants.CAR_SLOW_ACC;
        }

        /**
         * Returns four coordinates that are perpendicular (+-90 degrees)
         * to a logical or real position of the current vehicle.
         *
         * @see signalGreen.Utils#createCoordsFromCoordAndAngle(Coordinate, doub
le, double, Geography)
         * @param coordinate either real or network position of vehicle
         * @return array of coordinates
         */
        private Coordinate[] getPosition(Coordinate c) {
                double azimuth = Utils.getAzimuth(origin.getCoords(), next.getCo
ords(), getGeography());
                Coordinate position[] = Utils.createCoordsFromCoordAndAngle(c, a
zimuth, Constants.DIST_LANE, getGeography());
                return position;
        }
```

```
	public Coordinate getRealPosFromNetworkPos(Constants.Lane lane) {
		Coordinate position[] = this.getPosition(this.networkPos);
		if (lane == Lane.OUTER) {
			return position[0];
		}
		if (lane == Lane.INNER) {
			return position[1];
		}
		return null;
	}

	public Coordinate getNetworkPosFromRealPos(Constants.Lane lane) {
		Coordinate position[] = this.getPosition(this.realPos);
		if (lane == Lane.INNER) {
			return position[2];
		}
		if (lane == Lane.OUTER) {
			return position[3];
		}
		return null;
	}

	/**
	 * @return current velocity
	 */
	public int getVelocity() {
		return this.velocity;
	}

	/**
	 * @param currSpeed the currSpeed to set
	 */
	public void setVelocity(int currSpeed) {
		this.velocity = currSpeed;
	}

	protected void setMaxVelocity(int maxVelocity) {
		this.maxVelocity = maxVelocity;
	}


	public double getDisplacement() {
		return displacement;
	}

	/**
	 * Method computes new velocity
	 * according to acceleration and max velocity.<br />
	 * Uses standard kinematics equations for this purpose.
	 */
	private void accelerate() {
		// new velocity algorithm is:
		// V = V0 + a * t
		this.velocity += Math.ceil(Constants.ACCELERATION * Constants.t)
;
		// vehicle cannot go faster than its maxVelocity
		if (this.velocity > this.maxVelocity) {
			this.velocity = this.maxVelocity;
		}
	}

	/**
	 * Similar algorithm to the acceleration.
	 * Uses standard kinematics equations for this purpose.
	 */
	public void slowDown() {
		this.velocity -= Constants.ACCELERATION * Constants.t * 2;
		// velocity cannot be negative
		if (this.velocity < 0) {
```



```
			this.velocity = 0;
		}
	}

	/**
	 *
	 * @return the next junction on the route we are heading to
	 */
	private Junction getNextJunctionRoute() {
		Junction jNext = null;
		Iterator<RepastEdge<Junction>> it = this.vehicleRoute.iterator()
;
		if (it.hasNext()) {
			jNext = it.next().getTarget();
		}
		return jNext;
	}

	/**
	 *
	 * @return RepastEdge ie. the current road segment of the vehicle
	 */
	public RepastEdge<Junction> getNextRepastEdgeRoute() {
		RepastEdge<Junction> e = null;
		Iterator<RepastEdge<Junction>> it = this.vehicleRoute.iterator()
;
		if (it.hasNext()) {
			e = it.next();
		}
		return e;
	}

	/**
	 * @return Road the current road segment
	 */
	public Road getNextRoadSegmentRoute() {
		Road r = null;
		RepastEdge<Junction> e = getNextRepastEdgeRoute();
		if (e != null) {
			r = this.roads.get(e);
		}
		return r;
	}

	/**
	 * Simulates a vehicle using the blinker.
	 * Ie. the vehicle tells which lane is going to
	 * move to/stay on.
	 *
	 * @return OUTER or INNER lane
	 */
	public Lane getLane() {
		return lane;
	}

	public void setLane(Lane lane) {
		this.lane = lane;
	}

	public String getDebug() {
		return debug;
	}

	/**
	 * Method compares two vehicles based on their distance to the next junc
tion.
	 * Used to keep vehicles in a priority queue, in order
	 * to perform overtaking logic.
	 */
```

```
        @Override
        public int compareTo(Vehicle v) {
            double thisDist = this.getDistanceToNextJunction();
            double otherDist = v.getDistanceToNextJunction();
            if (thisDist < otherDist) return -1;
            if (thisDist > otherDist) return 1;
            return 0;
        }

        private double getDistanceToNextJunction() {
                return Utils.distance(this.getNetworkPos(), next.getCoords(), ge
tGeography());
        }

        /**
         * Used to display the vehicle's icon
         * using the correct angle. Uses convergence angle
         * from grid north + azimuth.
         * Called by the GIS display during simulation.
         *
         * @return angle in degrees
         */
        public double getAngle() {
                return Utils.getAngleForIcons(origin.getCoords(), next.getCoords
(), getGeography());
        }
}
```