A. The class SimpleJunction has a method add that allows the addition of cars to the cars list. However, this method does not check whether the addition conflicts with the invariant that cars cannot overlap. In the SimpleJunction class, the method add does not check whether the addition conflicts with the invariant. Therefore, a class is not sure.

If someone noted that the class is sure because the cars list is not synchronized, they get all the points.

B. The invariant of the class is not preserved: - Because of the reason noted in paragraph A, - There is no constraint in the code on the colors of the lights, they change according to the timer, so it is possible for two lights of intersecting directions to be green.

D. We use the OrientationSemaphore mechanism from part A, but instead of forcing the vehicle movement in the SimpleJunction class, we force the change of light color in the class TrafficLight.

If someone puts the mechanism in the move method, they get at the risk of five points - This does not ensure preservation of the invariant.

The invariant now forces the color of the lights, not the vehicle movement - This is a horizon point!

The changes in the code from solution A are highlighted in yellow.

```java
enum Light {RED, YELLOW, GREEN}

public class TrafficLight implements Runnable {
    public static trafficLight create(Direction dir, OrientationSemaphore orientation, int redInterval, int yellowInterval, int greenInterval) {
        TrafficLight tl = new TrafficLight(dir, orientation, redInterval, yellowInterval, greenInterval);
        new Thread(tl).start();
        return tl;
    }

    private TrafficLight(Direction dir, OrientationSemaphore orientation, int redInterval, int yellowInterval,
                           int greenInterval) {
        // Implementation
    }

    public void run() {
        // Implementation
    }
}
```
int greenInterval) {
    _light = Light.RED;
    _redInterval = redInterval; _yellowInterval = yellowInterval; _greenInterval = greenInterval;
    _moveThreads = new HashSet<Thread>();
    _orientation = orientation;
    _dir = dir;
}

public synchronized void addMoveThread(Thread moveThread) {
    _moveThreads.add(moveThread);
}

public synchronized Light getLight() {
    return _light;
}

public void run() {
    try {
        Thread.sleep(_redInterval);
        synchronized (this) {_light = Light.YELLOW; }
        Thread.sleep(_yellowInterval);
        _orientation.acquire(_dir);
        synchronized (this) {_light = Light.GREEN; notifyAll(); }
        Thread.sleep(_greenInterval);
        synchronized (this) {
            _light = Light.RED;
            _orientation.release(_dir);
            for (Thread moveThread : _moveThreads)
                moveThread.interrupt();
        }
    } catch (InterruptedException e) {
        return;
    }
}

Light _light;
Set<Thread> _moveThreads;
final int _redInterval, _yellowInterval, _greenInterval;
OrientationSemaphore _orientation;
Direction _dir;
enum Orientation {HORIZONTAL, VERTICAL, NONE}

class OrientationSemaphore {
    OrientationSemaphore() {
        _orientation = Orientation.NONE;
        _locks = 0;
    }

    public synchronized void acquire(Direction dir) throws InterruptedException {
        if (dir == Direction.LEFT || dir == Direction.RIGHT) {
            while (_orientation != Orientation.HORIZONTAL && _orientation != Orientation.NONE)
                wait();
            _orientation = Orientation.HORIZONTAL;
            _locks++;
        } else {
            if (dir == Direction.UP || dir == Direction.DOWN) {
                while (_orientation != Orientation.VERTICAL && _orientation != Orientation.NONE)
                    wait();
                _orientation = Orientation.VERTICAL;
                _locks++;
            }
        }
    }

    public synchronized void release() throws InterruptedException {
        _locks--;
        if (_locks == 0) {
            _orientation = Orientation.NONE;
            notifyAll();
        }
    }

    private Orientation _orientation;
    private int _locks;
}

public class SimpleJunction implements Junction {

    public SimpleJunction(TrafficLight leftLight, TrafficLight rightLight,
                           TrafficLight upLight, TrafficLight downLight) {
        _cars = new HashMap<Direction, List<Car>>();
        _cars.put(Direction.LEFT, new LinkedList<Car>());
        _cars.put(Direction.RIGHT, new LinkedList<Car>());
        _cars.put(Direction.UP, new LinkedList<Car>());
        _cars.put(Direction.DOWN, new LinkedList<Car>());
    }

    public void acquireCars(Car car) {
        _cars.get(car.direction).add(car);
    }

    public void releaseCars(Car car) {
        _cars.get(car.direction).remove(car);
    }

    public void releaseCars() {
        _cars.values().forEach(list -> list.clear());
    }

    private List<Car> _cars;
}

public class SimpuleJunction implements Junction {

    public SimpuleJunction(TrafficLight leftLight, TrafficLight rightLight,
                           TrafficLight upLight, TrafficLight downLight) {
        _cars = new HashMap<Direction, List<Car>>();
        _cars.put(Direction.LEFT, new LinkedList<Car>());
        _cars.put(Direction.RIGHT, new LinkedList<Car>());
        _cars.put(Direction.UP, new LinkedList<Car>());
        _cars.put(Direction.DOWN, new LinkedList<Car>());
    }

    public void acquireCars(Car car) {
        _cars.get(car.direction).add(car);
    }

    public void releaseCars(Car car) {
        _cars.get(car.direction).remove(car);
    }

    public void releaseCars() {
        _cars.values().forEach(list -> list.clear());
    }

    private List<Car> _cars;
}
_cars.put(Direction.UP, new LinkedList<Car>());
_cars.put(Direction.DOWN, new LinkedList<Car>());
_lights = new HashMap<Direction,TrafficLight>();
_lights.put(Direction.LEFT, leftLight);
_lights.put(Direction.RIGHT, rightLight);
_lights.put(Direction.UP, upLight);
_lights.put(Direction.DOWN, downLight);
_orientation = new OrientationSemaphore();

public void add(Car car, Direction dir) {
    List<Car> cars = _cars.get(dir);
synchronized (cars) {
        cars.add(car);
        cars.notifyAll();
    }
}

public void move(Direction dir) throws MoveException {
    // advance cars from the given direction it possible
    // 1. wait for a green light
    TrafficLight light = _lights.get(dir);
synchronized (light) {
        try{
            while (light.getLight() != Light.GREEN)
                light.wait();
        } catch (InterruptedException e) {
            return;
        }
    }
}

    //2. pass cars, as long as the light is green, if possible
    List<Car> cars = _cars.get(dir);
    while (true) {
        synchronized (cars) {
            Car car = null;
            //2.1 wait for cars
            try{
                while (cars.isEmpty())
                    cars.wait();
                car = cars.remove(0);
            } catch (InterruptedException e) {
                return;
            }
        }
    }
try{
    //2.2.1 wait for a legal orientation
    _orientation.acquire(dir);
    if (light.getLight() == Light.GREEN)
        try {
            //2.2.2 advance the car
            synchronized (car) { car.advance(); }
            _orientation.release();
        } catch (CarAdvancedException e) {
            _orientation.release();
            throw new MoveException(e);
        }
    else {
        // The TrafficLight interrupts all move threads when the light turns RED
        // In case the light turned to RED before the car was advanced,
        // return car back to the waiting list and release the orientation semaphore
        synchronized (cars) { cars.add(0, car); }
        _orientation.release();
        return;
    }
} catch (InterruptedException e) {
    // The TrafficLight interrupts all move threads when the light turns to RED
    // In case the light turned to RED during the waiting for legal orientation,
    // return car back to the waiting list
    synchronized (cars) { cars.add(0, car); }
    return;
}
}

Map<Direction,List<Car>> _cars;
Map<Direction,TrafficLight> _lights;
OrientationSemaphore _orientation;
BlockScene::BlockScene(int nrows, int ncols)
{
    _nrows=nrows;
    _ncols=ncols;

    _rowscols = new Cell**[nrows];

    for (int i=0; i<nrows; i++)
    {
        _rowscols[i] = new Cell*[ncols];

        for (int j=0; j<ncols; j++)
        {
            _rowscols[i][j] = new BlockCell();
        }
    }
}

BlockScene::~BlockScene()
{
    for (int i=0; i<_nrows; i++)
    {
        for (int j=0; j<_ncols; j++)
        {
            delete _rowscols[i][j];
        }
        delete _rowscols[i];
    }
    delete _rowscols;
}

BlockScene::BlockScene(const BlockScene &other)
{
    _nrows=other._nrows;
    _ncols=other._ncols;

    _rowscols = new Cell**[_nrows];

    for (int i=0; i<_nrows; i++)
    {
_rowscols[i] = new Cell*[_ncols];

    for (int j=0; j<_ncols; j++)
    {
        _rowscols[i][j] = new BlockCell(other._rowscols[i][j]);
    }
}

BlockScene& BlockScene::operator=(BlockScene &other)
{
    for (int i=0; i<_nrows; i++)
    {
        for (int j=0; j<_ncols; j++)
        {
            delete _rowscols[i][j];
        }
        delete _rowscols[i];
    }
    delete _rowscols;

    _nrows=other._nrows;
    _.ncols=other._ncols;

    _rowscols = new Cell**[_nrows];

    for (int i=0; i<_nrows; i++)
    {
        _rowscols[i] = new Cell*[_ncols];

        for (int j=0; j<_ncols; j++)
        {
            _rowscols[i][j] = new BlockCell(other._rowscols[i][j]);
        }
    }

    return *this;
}
סעיף ג

<table>
<thead>
<tr>
<th>vtable BlockCell</th>
<th>Cell virtual methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell fields</td>
<td>BlockCell (only) virtual methods</td>
</tr>
<tr>
<td>vtable Statistics</td>
<td>Statistics virtual methods</td>
</tr>
<tr>
<td>Statistics fields</td>
<td></td>
</tr>
<tr>
<td>BlockCell fields</td>
<td></td>
</tr>
</tbody>
</table>

תשובה 3

8
public void subscribe (String group) throws java.rmi.RemoteException, IOException {
    _writer.print("SUBSCRIBE\ndestination: "+group+"\n\n0");
}
public void unsubscribe (String group) throws java.rmi.RemoteException, IOException {
    _writer.print("UNSUBSCRIBE\ndestination: "+group+"\n\n\n\n0");
}
public void send (String group, String str) throws java.rmi.RemoteException, IOException {
    _writer.print("SEND\ndestination: "+group+"\n\n\n\n"+str+"\n\n\n\n0");
}

String host = args[0];
int port = Integer.parseInt(args[1]);
StompConnector stompConnector = new StompConnectorImpl(host, port);
java.rmi.Naming.rebind("/132.23.5.8:2010/StompConnector", stompConnector);

String login = args[0];
String passcode = args[1];
StompConnector stompConnector = (StompConnector)java.rmi.Naming.lookup("/132.23.5.8:2010/stompConnector");
StompOperator stompOperator = stompConnector.connect(login, passcode);
stompOperator.subscribe("q1");
stompOperator.send("q3", "Suzy Se");
send 2 פעולות (פתוחות למטא, פורמטים לשיחה)

פעולה 7 פעולות.

סעיף ה

לא נכון, מספר פעולות תקשורת לא תלוי בניהול חישובים מקביליים בשרת 2.

לא נכון, השרת אמין בשני המקרים (אך גם תשובת "נכון" התקבלה בהנחה שהתכוונת שהherits thread-per-client

נכון, אחד המטרות של Reactor pattern - ניצול יעיל יותר של המעבד (והזכרון)

שאלת 4

(10 נקודות)