

Advanced Programming: Concurrency INM420

Coursework 2015-16

SEMRA HAKAN

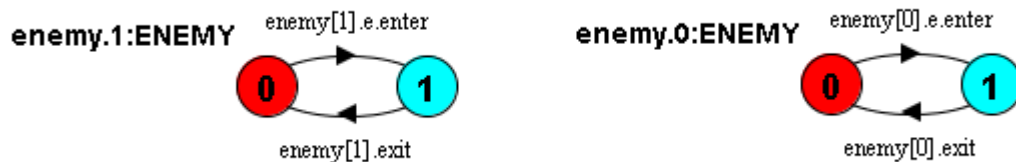
SOFTWARE ENGINEERING, MSc, (FT)

STUDENT ID: 150054430

Part A – Unsafe Model

a) The explanation of the unsafe model and FSP model

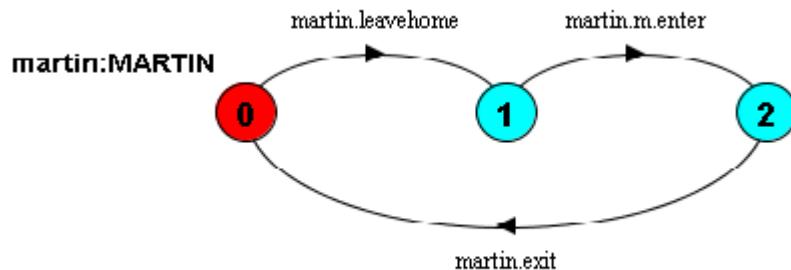
The UNSAFE_MARTIN model was created with warning indicator, enemy, and martin processes, to addition of given processes. I specified the expected behaviours of the processes, such as the task of an enemy is to enter and exit the road. This process is shown below.



“ENEMY = (e.enter-> exit -> ENEMY).”

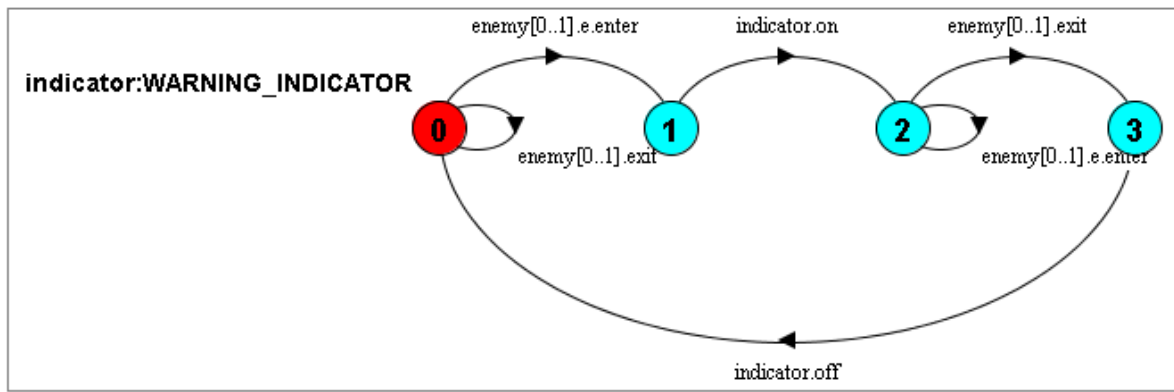
The number of enemies is declared as two to keep the system understandable in the composite process.

Similarly, the expected behaviour of mole (declared as martin process) is exiting his house, entering and exiting the road. Martin process is depicted below.



“MARTIN=(leavehome-> m.enter -> exit -> MARTIN).”

Warning indicator is related to enemy's actions so I synchronized them together. Warning indicator is defined with guard actions with the use of range. Zero indicates that there is no enemy on the road and if an enemy enters it turns on and increase the number of enemies. To let more than one enemy to enter at the same time, one more enter action added without having on action. The reason of doing this is warning indicator can be only turn on once for an enemy. The same logic is followed for exit action. In the end when the first enemy enters the road, warning indicator is turned on, when the second enemy comes to the road it does not turn on again. When any of these enemies exit the road, warning indicator turns off once again. Warning indicator process can be found below.



“WARNING_INDICATOR = WARNING [0],

// zero means no enemy on the road

WARNING[i:EnemyID]= (when(i==0) e.enter -> on-> WARNING[i+1]

// more than one car can enter, do not turn on again

|when(i==1) e.enter -> WARNING[i]

// if number of enemies are greater than zero, when they exit, decrease the numbers,and turn off just once.

|when(i==1) exit -> off-> WARNING[i-1]

|when(i==0) exit -> WARNING[i]).”

Exiting sensor is a shared sensor, it gives signal when anyone exits the road. Martin and enemy processes use it so these processes are declared inside of curly braces in the composite process. It looks like as follows:

“{martin,enemy[EnemyID]}::exitingS:EXITING_SENSOR”

To show the shared process in the structure diagram, I used two nested rectangular symbols. To make the model synchronous, I associated necessary actions together. These associated actions can be seen in the structure diagram.

The given entering sensor is related to sensor2 and sensor3. Since they are an entering sensors, I synchronized them with martin and enemy’s enter action. The name of entering sensor in the FSP model is enteringS, and it is synchronized as follows.

“martin.m.enter / enteringS.enter,
enemy[EnemyID].e.enter/ enteringS.enter”

In the unsafe martin model, there is no control on gate process. However the constraint is when enemy exits the road, it reappears in front of the gate. To do this, I associated gate’s pass action with enemy’s enter action, assuming gate is open when enemies enter. If gate’s lower action is selected, enemies can not enter the road. This rule is not affect martin’s entering action since there is nothing related to gate’s actions with martin’s process.

The FSP model for UNSAFE_MARTIN is shown below.

const Enemy_No = 2

range EnemyID = 0..Enemy_No-1

GATE = UP,

UP = (lower -> DOWN | pass -> UP),

DOWN = (raise -> UP).

ENTERING_SENSOR = (enter -> ENTERING_SENSOR).

EXITING_SENSOR = (exit -> EXITING_SENSOR).

WARNING_INDICATOR = WARNING [0],

WARNING[i:EnemyID]= (

when(i==0) e.enter -> on-> WARNING[i+1] // zero

means no enemy on the road

|when(i==1) e.enter -> WARNING[i] // more than

one car can enter, do not turn on again

|when(i==1) exit -> off-> WARNING[i-1] // if number

of enemies are greater than zero, when they exit, decrease the numbers just once.

|when(i==0) exit -> WARNING[i]).

ENEMY = (e.enter-> exit -> ENEMY).

MARTIN=(leavehome-> m.enter -> exit -> MARTIN).

|| UNSAFE_MARTIN =(indicator: WARNING_INDICATOR || martin:MARTIN ||
enteringS:ENTERING_SENSOR ||

{ martin,enemy[EnemyID]}::exitingS:EXITING_SENSOR // exiting sensor is a shared
process.

|| enemy[EnemyID]:ENEMY || gate:GATE)

/ {

martin.m.enter / enteringS.enter,

enemy[EnemyID].e.enter/ indicator.e.enter,

enemy[EnemyID].e.enter/ enteringS.enter,

enemy[EnemyID].exit/ exitingS.exit,

enemy[EnemyID].exit / indicator.exit ,

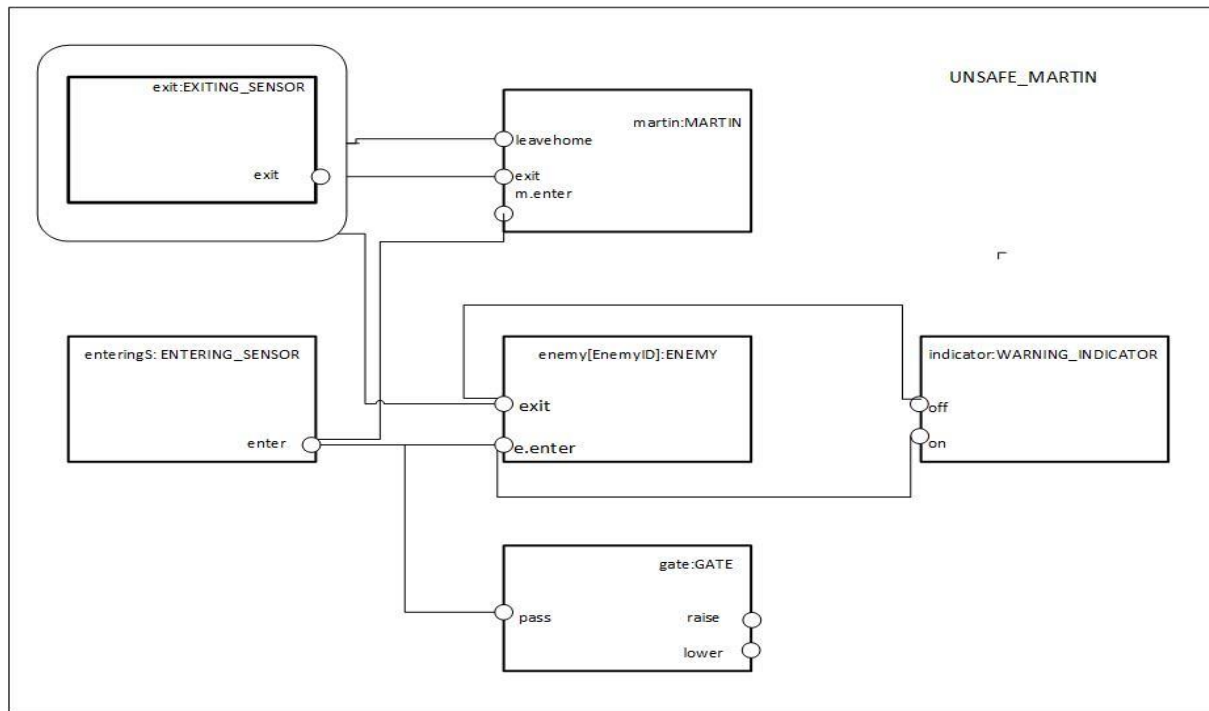
enemy[EnemyID].e.enter/gate.pass, // assuming to enter the road, gate should be open. there
is no control on the gate.

martin.exit/ martin.exitingS.exit,

martin.leavehome/ martin.exitingS.exit

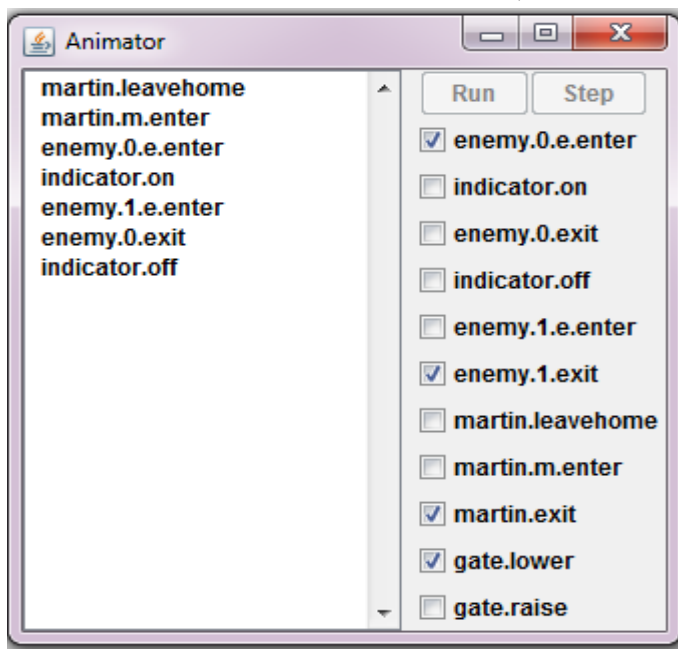
}.

Structure Diagram for UNSAFE_MARTIN composite process is depicted below.



b) LTSA animator window

It can be seen that when enemies enter, martin can also enter to the road.



c) Safety property NO_CRASH

In addition to the codes in section a, the required safety property is defined and composed this with UNSAFE_MARTIN process. We need to declare a safety property with the expected behaviours of the target system. The safety property is shown below.

```
property NO_CRASH = ENEMYCAR[0],
```

```
// martin can enter and exit the road
```

```
ENEMYCAR[i:0..Enemy_No] = ( martin.m.enter -> martin.exit -> ENEMYCAR[0]
```

```
|when (i<Enemy_No) enemy[EnemyID].e.enter->indicator.on -> ENEMYCAR[i+1]
```

```
|when (i > 0) enemy[EnemyID].exit -> indicator.off -> ENEMYCAR[i-1]) .
```

The required behaviour of the system is that martin should enter and exit the road. Since martin just a mole, we do not need to count it. However, to restrict and count the numbers of enemies I need to give range them. As it can be seen, if enemy is lower than its expected numbers, it should be able to enter. If it is not, it should be able to exit.

The other expected behaviour of the system is for warning indicator component, which indicates if there is an enemy on the road turn on and if there is not turn off. Therefore, this expected behaviour is added to the safety property in ENEMYCAR process.

d) Verification of collisions with safety property

When safety property is composed with UNSAFE_MARTIN composite process, I have a property violation which is shown below. As it can be seen, collision occurs between enemy.0.e.enter and martin.m.enter.

“ Trace to property violation in NO_CRASH:

```
    enemy.0.e.enter  
    martin.leavehome  
    martin.m.enter ”
```

The relationship between my model and the specification; We can divide the system into two parts to explain the relationship with the model and system specification. The first part is to describe martin actions with the specification. Due to sensor 1, which gives signal when martin leaves home, is an exiting sensor, I synchronized this sensor with the given exiting sensor process. Rather than giving the name as sensor1, simply I defined it as leavehome in martin process. Since both evoke the same meaning. There is one more sensor for martin before entering the road in the model. I defined this sensor as martin's enter action, and synchronized it with entering sensor process. Since entering sensor process has got just one action, called enter. The warning indicator is positioned before martin enters the road in the specification of the model. This indicator's actions are synchronous with enemy's actions so it turns on/off immediately when an enemy enters/exits.

The second part is for enemy actions to describe the relationship with the specification. In the unsafe model, there is no control mechanism on the gate process. Therefore, I assume that gate is open for enemies to enter. For this reason, I synchronized gate's pass action with enemy's enter action. At the same time, sensor3, which is entering sensor, should signal to indicate that enemies are on the road. Therefore, I synchronized enemy's enter action with entering sensor's enter action. The last sensor for martin and enemy called sensor 4 which is exiting sensor. I synchronized its action, exit, with enemy's and martin's exit action together, since it is a shared process.

The problematic issues and solutions; In addition to the gate process problem in the unsafe model, I tried to let more than one enemies to enter at the same time. I achieved this by modifying warning indicator process. Other issues are explained in the relevant sections with the problem.

Part B: Safe Model

The partners agreed to carry on with the model developed by Halil Keskin. The partners decided to modify the existing warning indicator process to make it more realistic. In order to achieve that an additional component was introduced to model called “car counter” which counts the cars currently on the road. The new warning indicator process became simpler.

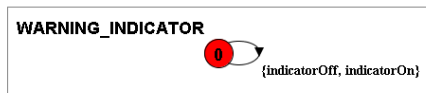
The modified warning indicator and the new car counter processes are defined as follows:

```
//on when counter >0 off when counter=0.
WARNING_INDICATOR={indicatorOn -> WARNING_INDICATOR
                    |indicatorOff -> WARNING_INDICATOR}.

//counts cars.
CAR_COUNTER=COUNT[0],
COUNT[i:0..MaxAllwdCarsOnRoad]={when(i<MaxAllwdCarsOnRoad
                                     increase -> COUNT[i+1]
                                     |when(i>0)
                                     decrease -> COUNT[i-1]
                                     |count[i]->COUNT[i])}.

```

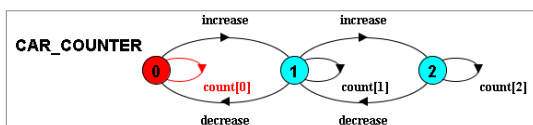
WARNING INDICATOR



indicatorOff: warning indicator is off.

indicatorOn: warning indicator is on.

CAR COUNTER (Counts the cars on the road)



increase: increase the counter by 1.

decrease: decrease the counter by 1.

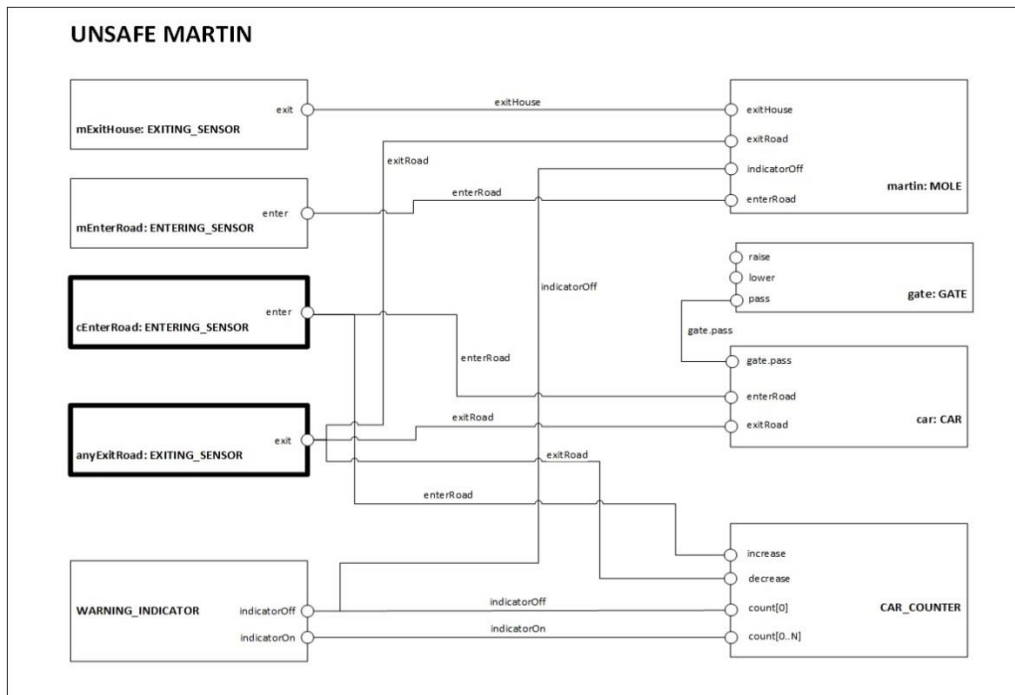
count: current state of the counter.

The new warning indicator’s *indicatorOn* and *indicatorOff* actions are synchronised with the car counter process’ *count* actions.

```
indicatorOff/{count[0],martin.indicatorOff},
indicatorOn/count[1..MaxAllwdCarsOnRoad],

```

The below is the updated structural diagram for UNSAFE_MARTIN model.



e) Controller which ensures that there won't be a collisions with Martin.

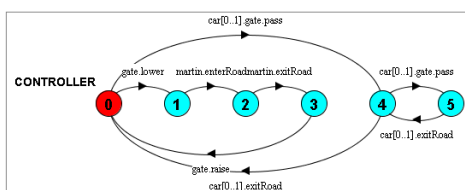
The first controller component that we developed was much simpler and made the model safe. However, we realized that, it did not let more than one cars to enter at the same time. Thus we use guard actions and the latest controller component can be seen below.

The below controller process made sure that no car can enter the road when Martin is on the road. This was achieved by synchronizing the gate's lower, raise and pass actions with the controller's relevant actions.

```

CONTROLLER = CONTROLLER[0],
CONTROLLER[i:0..MaxAllwdCarsOnRoad]= {when (i<MaxAllwdCarsOnRoad)
    car[CarID].gate.pass -> CONTROLLER[i+1]
|when (i>0)
    car[CarID].exitRoad->CONTROLLER[i-1]
|when (i==0)
    gate.lower -> martin.enterRoad -> martin.exitRoad -> gate.raise -> CONTROLLER[0]}.

```



As seen the drawing, controller allows two possible route: one starting with gate.pass action which allows cars to enter the road and the other one starting with gate.lower action which blocks cars to enter the road but allows martin to enter the road. The gate.lower is guarded so that it can only happen when there is no car in the road.

f) Formal verification of SAFE_MARTIN satisfying NO_CRASH safety property.

When the “safety check” is performed, the LTSA tool shows “No deadlocks/errors” for the new model (SAFE_MARTIN) with the controller, therefore the safety property is satisfied.

g) LIVE_MARTIN: Will Martin be able to eventually exit the road.

Progress property is used to verify that a specified action will eventually happen. To test whether Martin will be able to eventually exit the road a progress property LIVE_MARTIN with the below definition was added to the SAFE_MARTIN model.

```
progress LIVE_MARTIN= {martin.enterRoad}
```

When the “progress check” is performed, the LTSA tool shows “**No Progress violations detected.**” meaning Martin can eventually exit (first enter) the road.

h) LIVE_MARTIN: Will Martin be able to eventually exit the CONGESTED road.

In the SAFE_CONGESTED model, the priority is given to the cars’ gate.pass and enterRoad actions so that the road becomes congested. The definition for the SAFE_CONGESTED model is as follows.

```
||SAFE_CONGESTED = SAFE_MARTIN << {car[CarID].gate.pass,car[CarID].enterRoad }.
```

When the “progress check” is performed, the LTSA tool shows a “progress violation” in the SAFE_CONGESTED model. The trace to terminal set of states and the cycle in terminal set are as follows:

```
Progress Check...
-- States: 12 Transitions: 19 Memory used: 15769K
Finding trace to cycle...
Finding trace in cycle...
Progress violation: LIVE_MARTIN
Trace to terminal set of states:
    car.0.gate.pass
    car.0.enterRoad
    car.1.gate.pass
    car.1.enterRoad
    martin.exitHouse
Cycle in terminal set:
    car.0.exitRoad
    car.0.gate.pass
    car.0.enterRoad
Actions in terminal set:
    {car[0..1].{enterRoad, exitRoad}, gate.pass}, indicatorOn}
Progress Check in: 0ms
```

Part C: Safe Model in Java

The Car and Martin processes initiate entering and exiting actions so they were implemented as threads. Controller, gate and indicator processes are implemented as monitor since they respond these actions.

Car, martin, gate, controller, indicator and composite process, which is called SafeMartin, are created as classes in java to correspond these actions in the model. The indicator class represents the counter and the indicator processes in the FSP model. The Gate class has pass, raise and lower methods. The raise and lower methods modify gate’s state and therefore blocks or allows enemy cars’ passage. The run method of the Car class executes roadEnter and roadExit methods. Before executing the roadEnter method, the run method in the Car class calls the gate class’ pass method which is, depending on the state of the gate, executed immediately or suspended. The run method also calls the indicator class’s increase and decrease methods and modify the counter variable in the indicator class. The run method in the Mole class executes exitHouse, enterRoad and exitRoad methods. The enterRoad method is only executed if the indicator class’ counter variable is zero. Gate class’ lower method, which blocks pass method, is called by the run method in the Mole class before enterRoad method is executed. The run method in the Mole class finally calls the Gate class’ raise method which issues a notifyAll command to “unsuspend” the waiting threads therefore causes cars to proceed. (at least try to proceed.)

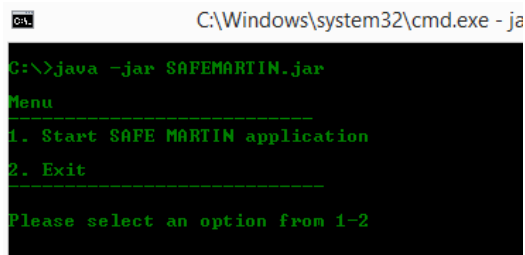
Our Java implementation has a command line interface. The methods in the classes simply generate relevant outputs in the command line.

To run the application please issue the below command:

```
java -jar SAFEMARTIN.jar
```

SAFEMARTIN.jar can be found in the zip folders. (i.e. under abvn955/**jar** folder)

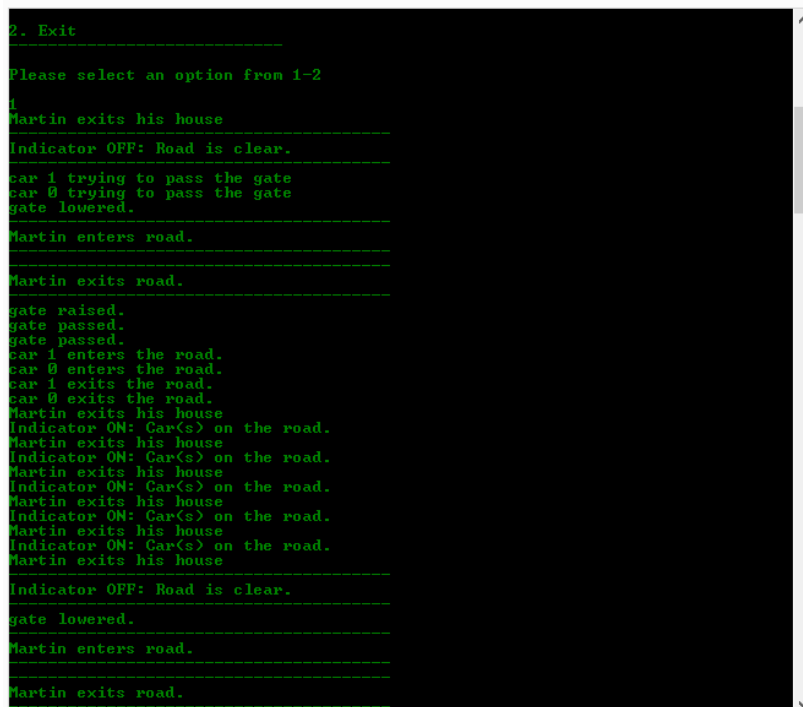
The CLI shows when the above command is issued. (In the below example, the jar file was saved in the location where the command was issued)



```
C:\Windows\system32\cmd.exe - ja
C:\>java -jar SAFEMARTIN.jar
Menu
-----
1. Start SAFE MARTIN application
2. Exit
-----
Please select an option from 1-2
```

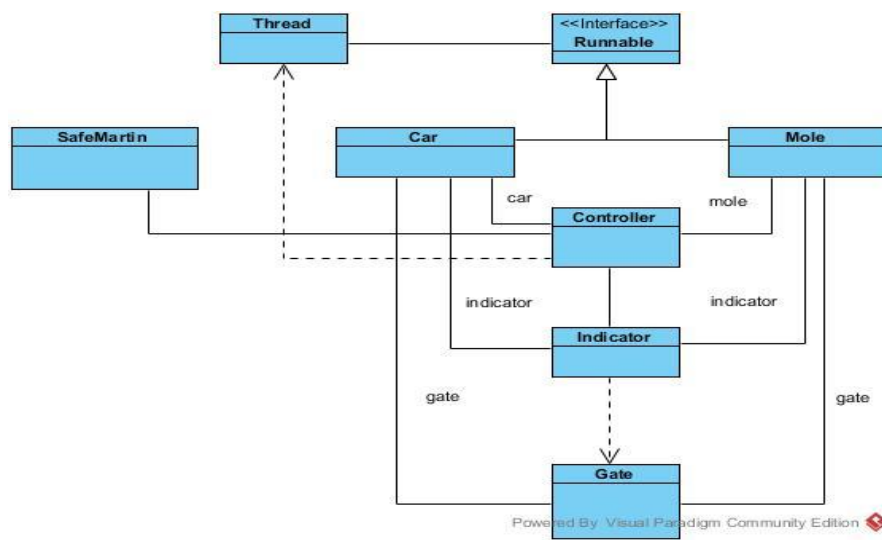
Please hit “1” and Enter keys to start the application. Once started, hitting 2 key will NOT stop it. Please stop it by closing the command line window or pressing Ctrl + C keys (Assuming the application will be tested on a windows computer. The application is configured to allow only two enemy cars. (It supports n cars).

All actions performed by Martin and the car threads are shown in the command line as shown in the below picture. Actions performed by Martin highlighted by printing a dashed line before and after his actions.



```
2. Exit
-----
Please select an option from 1-2
1
Martin exits his house
-----
Indicator OFF: Road is clear.
-----
car 1 trying to pass the gate
car 0 trying to pass the gate
gate lowered.
-----
Martin enters road.
-----
Martin exits road.
-----
gate raised.
gate passed.
gate passed.
car 1 enters the road.
car 0 enters the road.
car 1 exits the road.
car 0 exits the road.
Martin exits his house
Indicator ON: Car(s) on the road.
Martin exits his house
Indicator ON: Car(s) on the road.
Martin exits his house
Indicator ON: Car(s) on the road.
Martin exits his house
Indicator ON: Car(s) on the road.
Martin exits his house
Indicator ON: Car(s) on the road.
Indicator OFF: Road is clear.
-----
gate lowered.
-----
Martin enters road.
-----
Martin exits road.
-----
```

The UML class diagram for the Java classes are as follows:



The source code for the classes is as follows:

```

package safemartin;

/**
 *
 * @author Halil and Semra
 */
import java.io.*;

public class SAFEMARTIN {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here

        BufferedReader br;
        int selectedOption;
        showMainMenu();
        br = new BufferedReader(new InputStreamReader(System.in));
        try {
            selectedOption = Integer.parseInt(br.readLine());
            switch (selectedOption) {
                case 1:
                    Controller c = new Controller();
                    c.run();
                    break;
                case 2:
                    System.exit(1);
                    break;
            }
        } catch (Exception e) {
            System.out.println("Error: " + e);
            System.exit(1);
        }
    }

    public static void showMainMenu() {
        System.out.println("");
        System.out.println("Menu");
        System.out.println("-----");
        System.out.println("1. Start SAFE MARTIN application");
        System.out.println("");
        System.out.println("2. Exit");
        System.out.println("-----");
        System.out.println("");
        System.out.println("Please select an option from 1-2");
        System.out.println("");
        System.out.println("");
    }
}

package safemartin;

/**
 *
 * @author Halil and Semra

```

```

*/
public class Indicator {
    private int numCarsOnTheRoad = 0;
    Indicator() {
    }
    synchronized void increase() {
        numCarsOnTheRoad++;
    }
    synchronized void decrease() throws InterruptedException {
        numCarsOnTheRoad--;
        Thread.sleep(2000);
    }

    int count() {
        return numCarsOnTheRoad;
    }
}
/*

```

* multiple threads example : <http://stackoverflow.com/questions/9874587/how-to-create-threads-dynamically>

```

*/
package safemartin;

/**
 *
 * @author Halil and Semra
 */
public class Controller {

    private Mole mole;
    private Car car;
    private int maxAllwdCarsOnRoad = 2;

    Controller() {

    }

    void run() {
        Indicator indicator = new Indicator();
        Gate gate = new Gate(false, indicator);

        mole = new Mole("Martin", gate, indicator);
        Thread martin = new Thread(mole);
        //martin.setPriority(Thread.MAX_PRIORITY);
        martin.start();

        Thread[] threads = new Thread[maxAllwdCarsOnRoad];
        for (int i = 0; i < threads.length; i++) {
            car = new Car(i, gate, indicator);
            threads[i] = new Thread(car);
            //threads[i].setPriority(Thread.MAX_PRIORITY);
            threads[i].start();
        }

    }

}

```

```

package safemartin;

/**
 *
 * @author Halil and Semra
 */
public class Gate {

    private boolean isGateUp=false;
    private Indicator indicator;

    Gate(boolean p_isGateUp,Indicator indicator) {
        isGateUp = p_isGateUp;
        this.indicator=indicator;
    }
    synchronized void pass()
        throws InterruptedException {

        while (isGateUp) {
            wait();
        }
        System.out.println("gate passed.");

    }

    synchronized void raise() throws InterruptedException {
        isGateUp = false;
    }
}

```

```

        System.out.println("gate raised.");
        notifyAll();
    }
    synchronized void lower() throws InterruptedException {
        isGateUp = true;
        System.out.println("gate lowered.");
    }
}
package safemartin;

public class Car implements Runnable {
    private int carID;
    private Gate gate;
    private Indicator indicator;
    Car(int id, Gate gate, Indicator indicator) {
        this.carID = id;
        this.gate = gate;
        this.indicator = indicator;
    }
    private void pass() {
        System.out.println("car " + carID + " trying to pass the gate");
    }
    private void enterRoad() {
        System.out.println("car " + carID + " enters the road.");
    }
    private void exitRoad() {
        System.out.println("car " + carID + " exits the road.");
    }
    public void run() {
        while (true) {
            try {
                pass();
                gate.pass();
                enterRoad();
                indicator.increase();
                exitRoad();
                indicator.decrease();
            } catch (Exception e) {
                System.out.println(e.getMessage().toString());
            }
        }
    }
}

```

```

package safemartin;
public class Mole implements Runnable {
    private String name;
    private Gate gate;
    private Indicator indicator;
    Mole(String name, Gate gate, Indicator indicator) {
        this.name = name;
        this.gate = gate;
        this.indicator = indicator;
    }
    private void exitHouse() {
        System.out.println(name + " exits his house");
    }
    private void enterRoad() {
        System.out.println("-----");
        System.out.println(name + " enters road.");
        System.out.println("-----");
    }
    private void exitRoad() {
        System.out.println("-----");
        System.out.println(name + " exits road.");
        System.out.println("-----");
    }
    public void run() {
        try {
            while (true) {
                this.exitHouse();
                if (indicator.count() == 0) {
                    System.out.println("-----");
                    System.out.println("Indicator OFF: Road is clear.");
                    System.out.println("-----");
                    gate.lower();
                    this.enterRoad();
                    this.exitRoad();
                    gate.raise();
                    Thread.sleep(2000);
                } else {
                    System.out.println("Indicator ON: Car(s) on the road.");
                }
            }
        } catch (Exception e) {
            System.out.println(e.getMessage().toString());
        }
    }
}

```

