Lab #2 – UML State Machines

Work composed by

Nathan Gawargy (300232268) & James Couture (300076065)

For the course of Software Construction (SEG 2106)

Course Professor: Shiva Nejati

Submission date: February 10th, 2023

University of Ottawa

Winter 2023

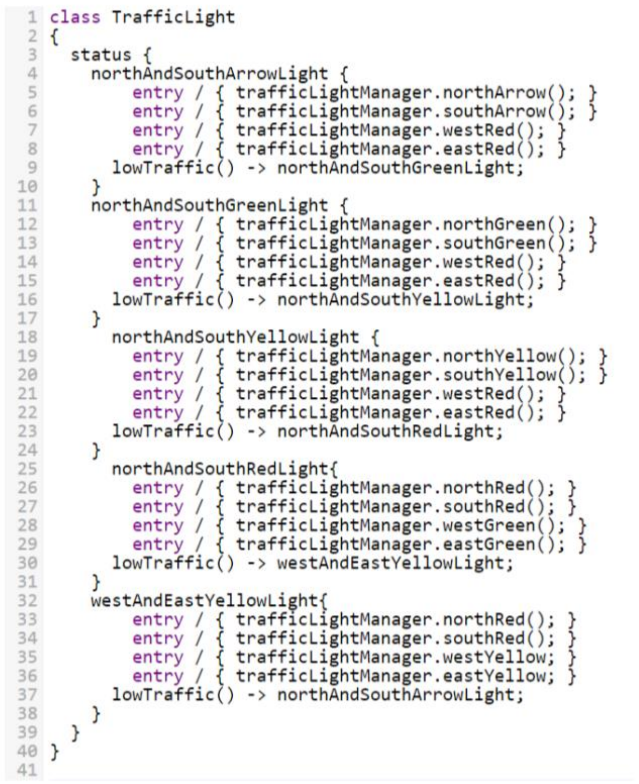
Objective

The purpose of the lab was to develop UML state machines used by UMPLE to generate Java code which had to be implemented in an existing traffic light program. It served as an important introduction to certain crucial components in software process models, such as behavioral modeling, and integration of a solution to a system.

Design

State Machines

The first step in the process was to design the state machines for the three different types of traffic (low, moderate, and high). Using the requirements for the new behavior of the traffic light system, we were able to successfully derive three different state machines for each mode. It is important to note that to facilitate the implementation of our solution, we considered all the transitions to be the same for each respective traffic mode *(see Figures below)* allowing us to group the transitions by their respective traffic mode *(see TrafficLight Java class)*. We also added the keyword: “Light”, “Mod”, or “High”, at the end of the names of the states to identify the different states with their respective traffic type (low, moderate, or high).



```
class TrafficLight
{
  status {
    northAndSouthArrowLight {
        entry / { trafficLightManager.northArrow(); }
        entry / { trafficLightManager.southArrow(); }
        entry / { trafficLightManager.westRed(); }
        entry / { trafficLightManager.eastRed(); }
      lowTraffic() -> northAndSouthGreenLight;
    }
    northAndSouthGreenLight {
        entry / { trafficLightManager.northGreen(); }
        entry / { trafficLightManager.southGreen(); }
        entry / { trafficLightManager.westRed(); }
        entry / { trafficLightManager.eastRed(); }
      lowTraffic() -> northAndSouthYellowLight;
    }
      northAndSouthYellowLight {
        entry / { trafficLightManager.northYellow(); }
        entry / { trafficLightManager.southYellow(); }
        entry / { trafficLightManager.westRed(); }
        entry / { trafficLightManager.eastRed(); }
      lowTraffic() -> northAndSouthRedLight;
    }
      northAndSouthRedLight{
        entry / { trafficLightManager.northRed(); }
        entry / { trafficLightManager.southRed(); }
        entry / { trafficLightManager.westGreen(); }
        entry / { trafficLightManager.eastGreen(); }
      lowTraffic() -> westAndEastYellowLight;
    }
    westAndEastYellowLight{
        entry / { trafficLightManager.northRed(); }
        entry / { trafficLightManager.southRed(); }
        entry / { trafficLightManager.westYellow; }
        entry / { trafficLightManager.eastYellow; }
      lowTraffic() -> northAndSouthArrowLight;
    }
  }
}
```

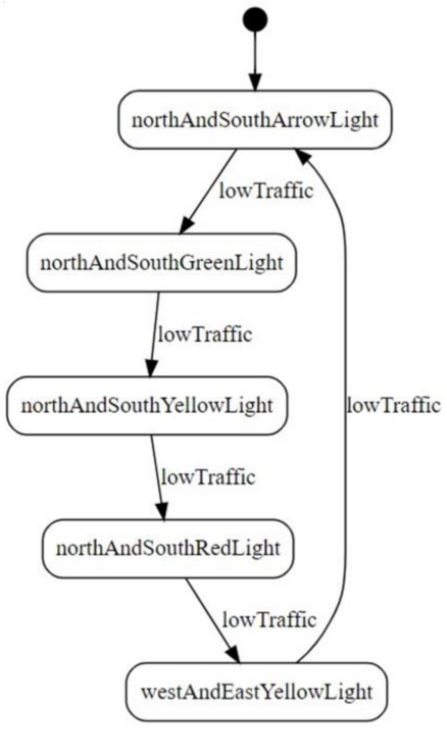
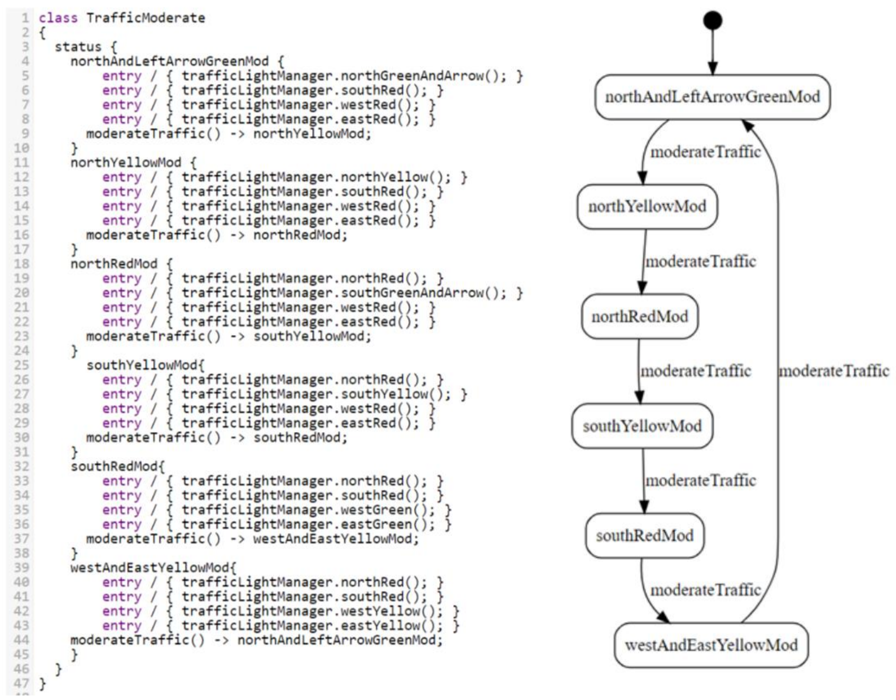
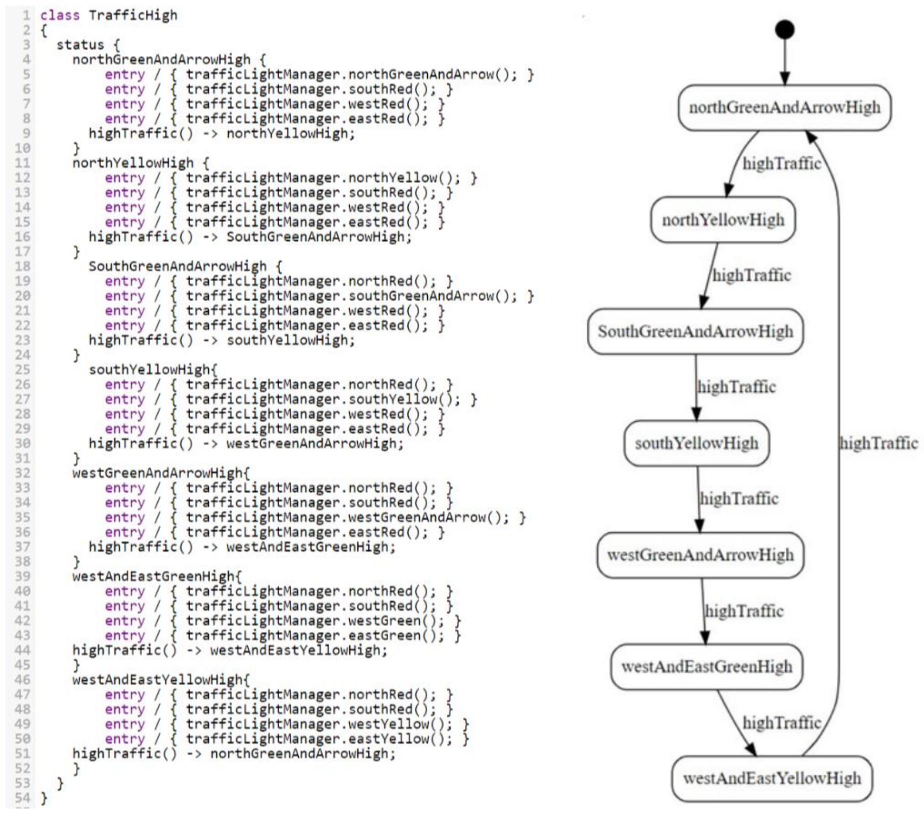


Figure 1: *Low traffic state machine (with UMPLE code)*



```
class TrafficModerate
{
  status {
    northAndLeftArrowGreenMod {
        entry / { trafficLightManager.northGreenAndArrow(); }
        entry / { trafficLightManager.southRed(); }
        entry / { trafficLightManager.westRed(); }
        entry / { trafficLightManager.eastRed(); }
      moderateTraffic() -> northYellowMod;
    }
    northYellowMod {
        entry / { trafficLightManager.northYellow(); }
        entry / { trafficLightManager.southRed(); }
        entry / { trafficLightManager.westRed(); }
        entry / { trafficLightManager.eastRed(); }
      moderateTraffic() -> northRedMod;
    }
    northRedMod {
        entry / { trafficLightManager.northRed(); }
        entry / { trafficLightManager.southGreenAndArrow(); }
        entry / { trafficLightManager.westRed(); }
        entry / { trafficLightManager.eastRed(); }
      moderateTraffic() -> southYellowMod;
    }
      southYellowMod{
        entry / { trafficLightManager.northRed(); }
        entry / { trafficLightManager.southYellow(); }
        entry / { trafficLightManager.westRed(); }
        entry / { trafficLightManager.eastRed(); }
      moderateTraffic() -> southRedMod;
    }
    southRedMod{
        entry / { trafficLightManager.northRed(); }
        entry / { trafficLightManager.southRed(); }
        entry / { trafficLightManager.westGreen(); }
        entry / { trafficLightManager.eastGreen(); }
      moderateTraffic() -> westAndEastYellowMod;
    }
    westAndEastYellowMod{
        entry / { trafficLightManager.northRed(); }
        entry / { trafficLightManager.southRed(); }
        entry / { trafficLightManager.westYellow(); }
        entry / { trafficLightManager.eastYellow(); }
    moderateTraffic() -> northAndLeftArrowGreenMod;
    }
  }
}
```

Figure 2: *Moderate traffic state machine (with UMPLE code)*

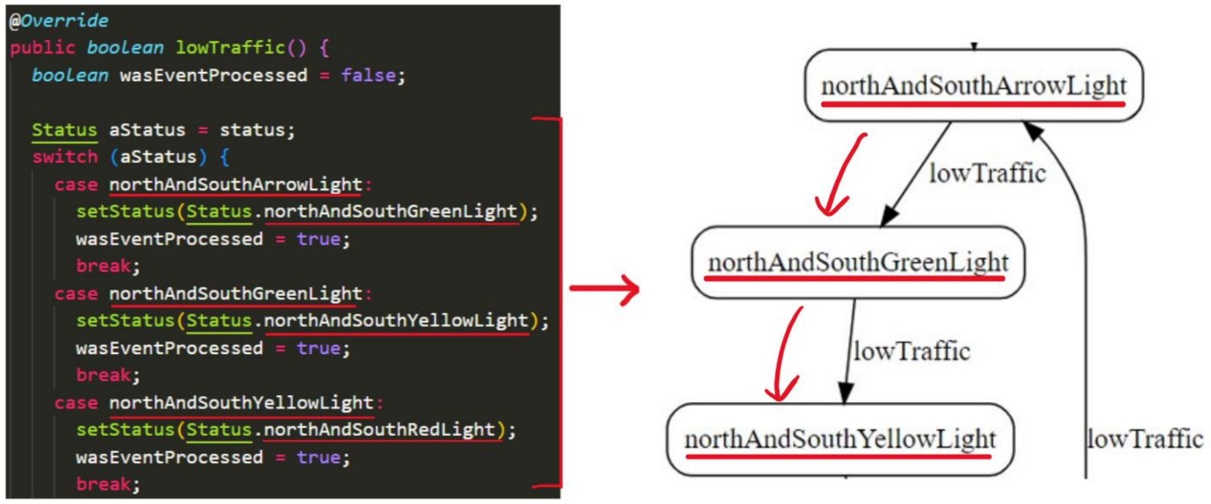


```
class TrafficHigh
{
  status {
    northGreenAndArrowHigh {
        entry / { trafficLightManager.northGreenAndArrow(); }
        entry / { trafficLightManager.southRed(); }
        entry / { trafficLightManager.westRed(); }
        entry / { trafficLightManager.eastRed(); }
      highTraffic() -> northYellowHigh;
    }
    northYellowHigh {
        entry / { trafficLightManager.northYellow(); }
        entry / { trafficLightManager.southRed(); }
        entry / { trafficLightManager.westRed(); }
        entry / { trafficLightManager.eastRed(); }
      highTraffic() -> SouthGreenAndArrowHigh;
    }
      SouthGreenAndArrowHigh {
        entry / { trafficLightManager.northRed(); }
        entry / { trafficLightManager.southGreenAndArrow(); }
        entry / { trafficLightManager.westRed(); }
        entry / { trafficLightManager.eastRed(); }
      highTraffic() -> southYellowHigh;
    }
      southYellowHigh{
        entry / { trafficLightManager.northRed(); }
        entry / { trafficLightManager.southYellow(); }
        entry / { trafficLightManager.westRed(); }
        entry / { trafficLightManager.eastRed(); }
      highTraffic() -> westGreenAndArrowHigh;
    }
    westGreenAndArrowHigh{
        entry / { trafficLightManager.northRed(); }
        entry / { trafficLightManager.southRed(); }
        entry / { trafficLightManager.westGreenAndArrow(); }
        entry / { trafficLightManager.eastRed(); }
      highTraffic() -> westAndEastGreenHigh;
    }
    westAndEastGreenHigh{
        entry / { trafficLightManager.northRed(); }
        entry / { trafficLightManager.southRed(); }
        entry / { trafficLightManager.westGreen(); }
        entry / { trafficLightManager.eastGreen(); }
    highTraffic() -> westAndEastYellowHigh;
    }
    westAndEastYellowHigh{
        entry / { trafficLightManager.northRed(); }
        entry / { trafficLightManager.southRed(); }
        entry / { trafficLightManager.westYellow(); }
        entry / { trafficLightManager.eastYellow(); }
    highTraffic() -> northGreenAndArrowHigh;
    }
  }
}
```

Figure 3: *High traffic state machine (with UMPLE code)*

Integration of Solution

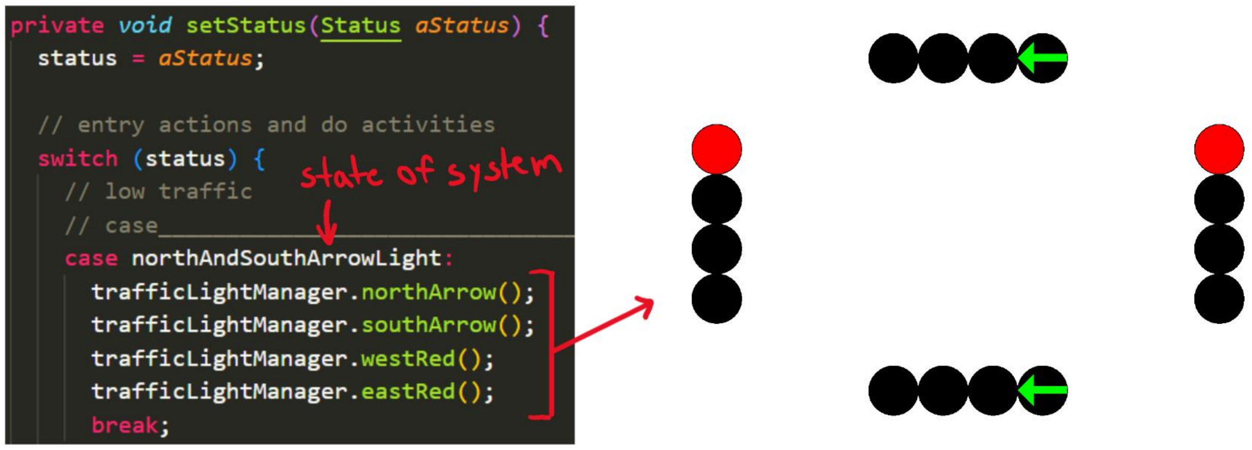
With the state machines, we were able to generate the corresponding Java code needed in the new traffic light system. This code dictated the sequence of states to be executed for each traffic mode. The image below is an example demonstrating how the state machine behaviour was modeled into code used in the program.



```
@Override
public boolean lowTraffic() {
  boolean wasEventProcessed = false;

  Status aStatus = status;
  switch (aStatus) {
    case northAndSouthArrowLight:
      setStatus(Status.northAndSouthGreenLight);
      wasEventProcessed = true;
      break;
    case northAndSouthGreenLight:
      setStatus(Status.northAndSouthYellowLight);
      wasEventProcessed = true;
      break;
    case northAndSouthYellowLight:
      setStatus(Status.northAndSouthRedLight);
      wasEventProcessed = true;
      break;
```

Figure 4: *Behaviour of the low traffic state machine in Java code*

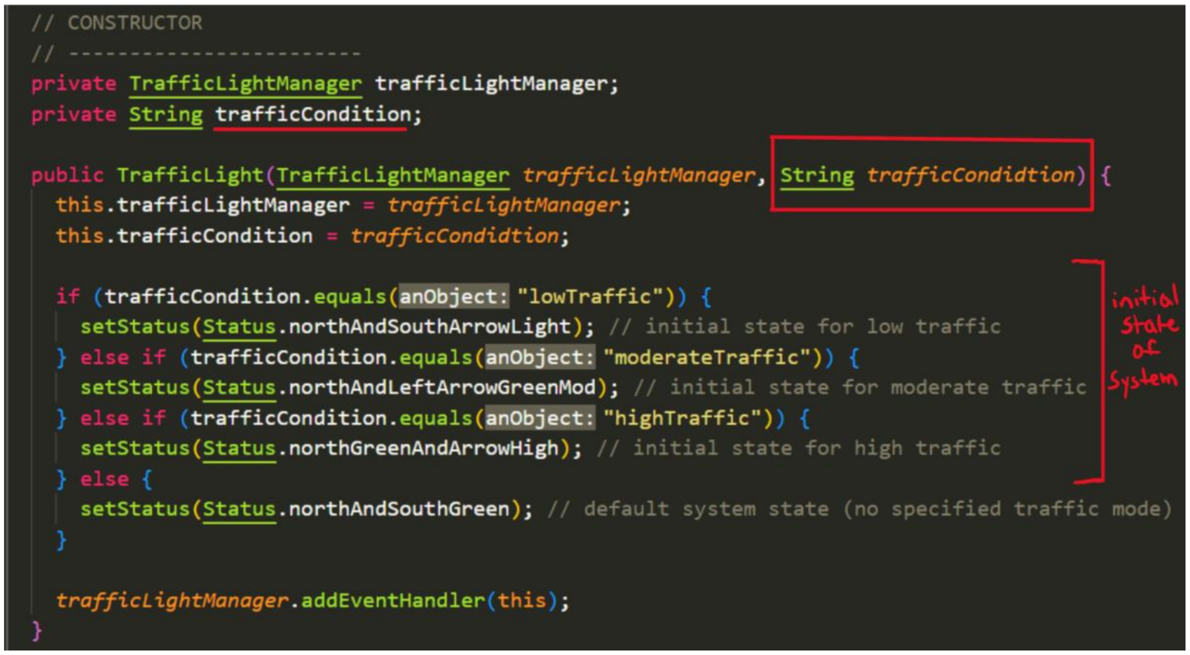
The auto-generated code also specified the behavior of each traffic light during their respective states (see image below).



```
private void setStatus(Status aStatus) {
  status = aStatus;

  // entry actions and do activities
  switch (status) {
    // low traffic
    // case
    case northAndSouthArrowLight:
      trafficLightManager.northArrow();
      trafficLightManager.southArrow();
      trafficLightManager.westRed();
      trafficLightManager.eastRed();
      break;
```

Figure 5: *Behaviour of each light during a certain state*

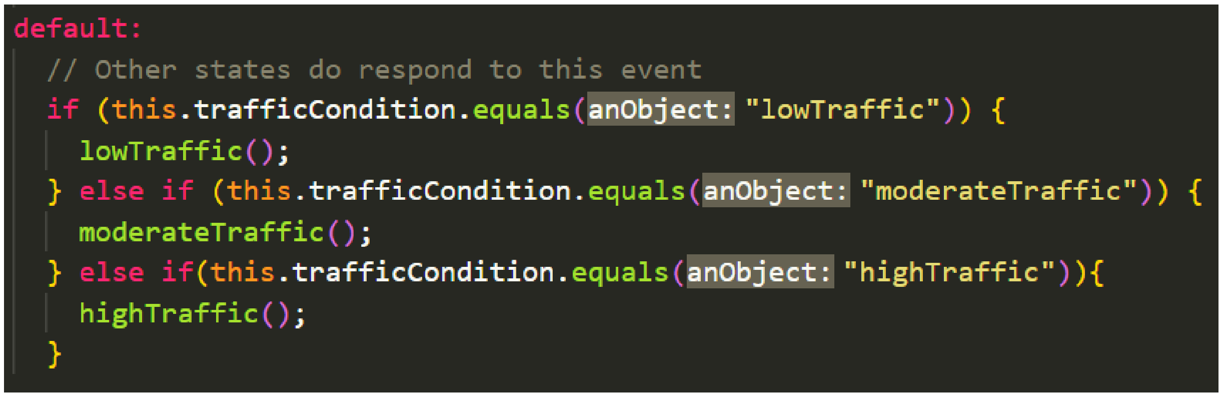
After adding the code responsible for the behaviour of the new traffic light system, it was necessary to modify the constructor of the Traffic Light Class. We added a new parameter for the constructor (trafficCondition), allowing the user to choose the type of traffic (low, moderate or high). Using this parameter, we specified the initial state of the system for each traffic mode.



```
// CONSTRUCTOR
// ------------------------
private TrafficLightManager trafficLightManager;
private String trafficCondition;

public TrafficLight(TrafficLightManager trafficLightManager, String trafficCondidtion) {
  this.trafficLightManager = trafficLightManager;
  this.trafficCondition = trafficCondidtion;

  if (trafficCondition.equals(anObject: "lowTraffic")) {
    setStatus(Status.northAndSouthArrowLight); // initial state for low traffic
  } else if (trafficCondition.equals(anObject: "moderateTraffic")) {
    setStatus(Status.northAndLeftArrowGreenMod); // initial state for moderate traffic
  } else if (trafficCondition.equals(anObject: "highTraffic")) {
    setStatus(Status.northGreenAndArrowHigh); // initial state for high traffic
  } else {
    setStatus(Status.northAndSouthGreen); // default system state (no specified traffic mode)
  }

  trafficLightManager.addEventHandler(this);
}
```

***Figure 6:** Modified constructor of the TrafficLight Class*

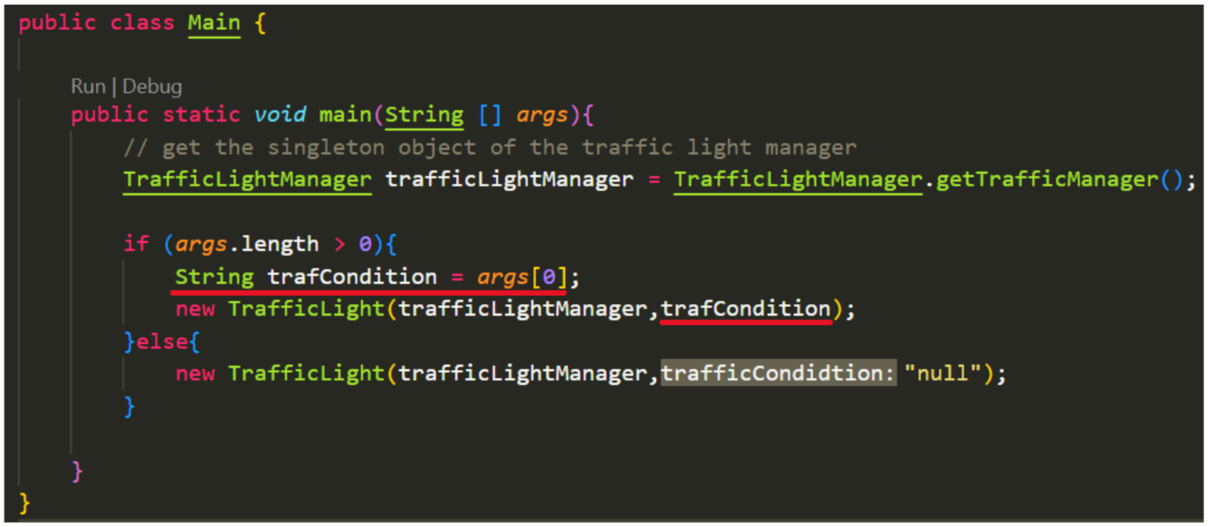
Using the trafficCondition variable, we then called the respective traffic mode in the timerGreen and timerYellow methods.



```
default:
  // Other states do respond to this event
  if (this.trafficCondition.equals(anObject: "lowTraffic")) {
    lowTraffic();
  } else if (this.trafficCondition.equals(anObject: "moderateTraffic")) {
    moderateTraffic();
  } else if(this.trafficCondition.equals(anObject: "highTraffic")){
    highTraffic();
  }
```

***Figure 7**: Modified timerGreen and timerYellow methods*

The final part in the integration of our solution was to modify the Main Class in order to allow the user to specify the traffic mode of the system (using the arguments).



```
public class Main {

    Run | Debug
    public static void main(String [] args){
        // get the singleton object of the traffic light manager
        TrafficLightManager trafficLightManager = TrafficLightManager.getTrafficManager();

        if (args.length > 0){
            String trafCondition = args[0];
            new TrafficLight(trafficLightManager,trafCondition);
        }else{
            new TrafficLight(trafficLightManager,trafficCondidtion: "null");
        }

    }
}
```

Figure 8: *Modified Main Class to allow traffic type to be specified*

Discussion

Our solution improved the organization of the system as the code for the different traffic types were partitioned in their own separate methods. A possible option was to use the pre-existing greenTimer and yellowTimer as transitions for the different traffic modes. However, this method would have over complexified the code structure of the program; reducing the code readability and making it more difficult for future modifications of the system. Instead, for each respective traffic mode, we used the same transition event *(see State Machines Diagrams)*. This allowed us to partition our code based on the corresponding traffic type leading to a more efficient, better organized and more flexible program.

To create the auto-generated Java code, we used the given UMPLE code as reference (see Instructions). We tried keeping a similar naming convention for each traffic mode for the states and the transitions. Since each state had only one possible transition and because of the circular structure of the state diagram, using the same transition for the entire cycle does not impact the behaviour of the system. This was a crucial decision that allowed us to integrate an efficient solution to the existing system.

The biggest problem we encountered during the lab was understanding how to correctly implement our solution for the new behaviour of the traffic light system. After debugging the system and verifying the code, we were able to successfully figure out how to integrate the new behaviour into the existing system. Initially, it was hard to find how the code worked in the background, but it was important to figure out to properly implement the new UMPLE generated code.

Conclusion

In conclusion, we were able to meet the objectives of this lab, as we derived the UML state machines, generated Java code using UMPLE, and successfully integrated the code into the existing program. Our implementation worked seamlessly, as the traffic light system now supports three different traffic modes: low, moderate and high. The implementation also still supports the default mode.