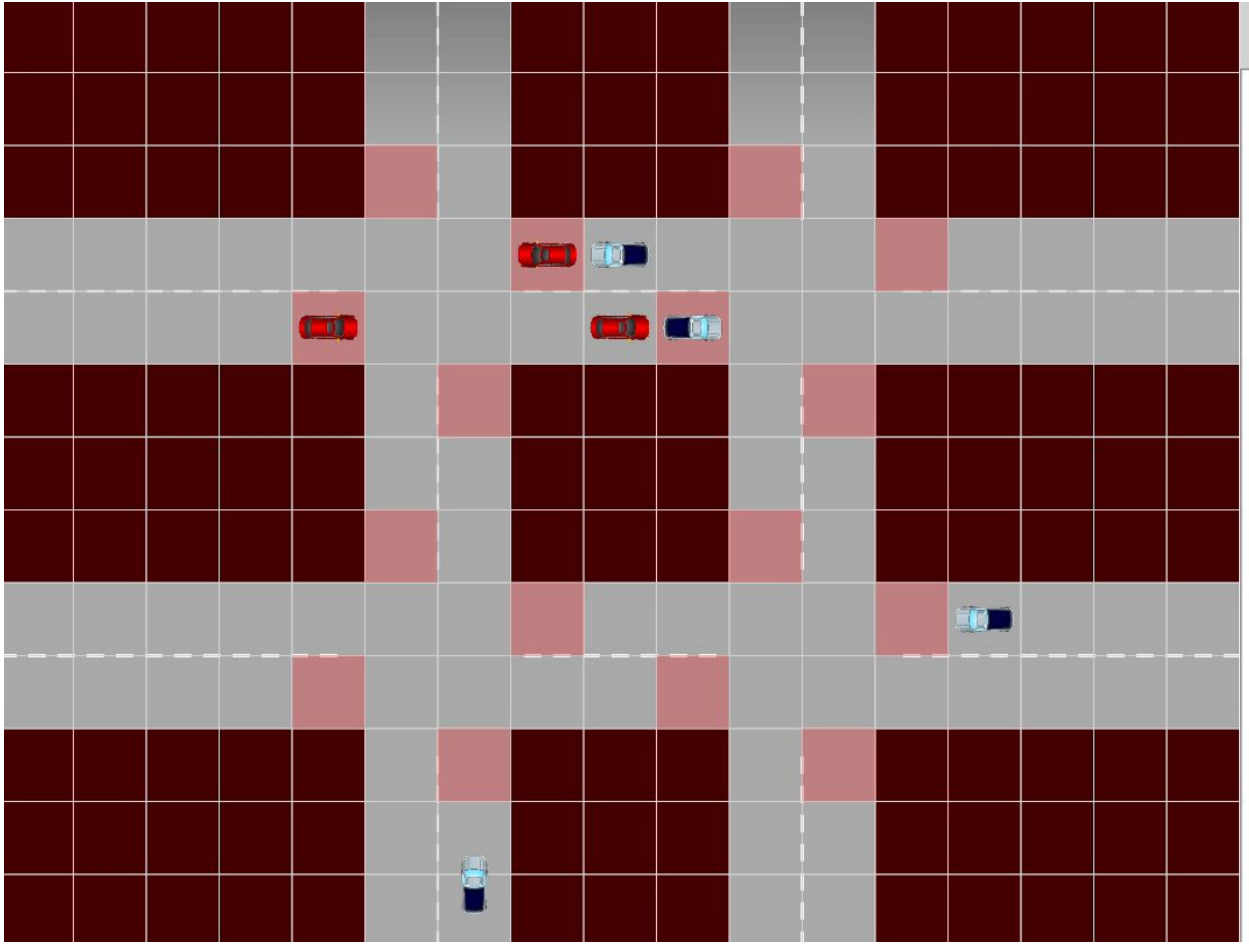

SMART CROSSROADS
TAXI
POLICE

TABLE OF CONTENTS

THE PROJECT

Smart Crossroads is a project aimed at simulating the automatic management of traffic in a certain area of a city in the presence of traffic light intersections. It foresees that cars and traffic lights are intelligent agents, able to communicate to avoid (or at least better manage) traffic queues.



Monitoring

Car moving
Car 31 is mowing to (13, 8)

Traffic light updating
Traffic light 0 changed to GREEN

Traffic light updating
Traffic light 0 changed to RED

Traffic light updating
Traffic light 5 changed to GREEN

Traffic light updating
Traffic light 5 changed to RED

Traffic light updating
Traffic light 15 changed to GREEN

Traffic light updating
Traffic light 15 changed to RED

Traffic light updating
Traffic light 11 changed to RED

Car spawned
Car 32 spawned in (6, 12)

Car moving
Car 32 is mowing to (6, 11)

TECHNOLOGIES

For the development of this project, different **technologies** were chosen, and each choice was made for experience or for convenience related to the frameworks.

Java

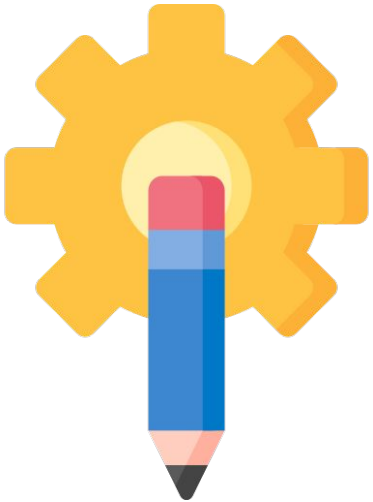
Java was chosen as the main programming language, because it was the simplest and fastest choice, and the one where experience would help the most. The other possible choice was Kotlin, but it would probably take more time to understand additional mechanisms.

JavaFX

JavaFX was chosen as the graphics library, preferred to Swing for various reasons (especially because of its more modern graphics and better performance). Even if Swing is undoubtedly simpler, JavaFX is clearly more advanced, and the possibilities are greater.

Jason

Jason is an interpreter for AgentSpeak, a language for programming intelligent agents based on the Belief-Desire-Intention (BDI) model. It allows developing multi-agent systems (MAS) in which agents make rational decisions. Jason was chosen because of the ease of creating and managing intelligent agents, as well as its integration with Java.



DESIGN

Design choices had to be made both for the agents and the GUI. For the agents, state diagrams were created to define their behaviors.

Agents

It was decided to create three types of agents: car, traffic light and creator. Cars must communicate with each other to avoid collisions, and they must also communicate with traffic lights to respect the current color. Traffic lights have a regular cycle, but they must adapt to the number of cars in the queue.

Environment

It was necessary to use an **environment** to manage the communications from the agents to the logic and graphics. An agent model was created to abstract them and better manage the agents' perceptions (for example the target calculation). This model then contains all the agents' data, exploiting the mechanisms of OOP, which are then translated into perceptions when requested by the various agents.

GUI

As for the GUI, the idea is to have a map composed of tiles, to simplify the car routes. Each car occupies a tile, and a tile can obviously contain only one car. As for the traffic lights, the idea is to represent them on the road instead of on the side, to make the simulation easier to understand.

IMPLEMENTATION

After making the design choices, the next step was the **implementation**, precisely defining the behaviors of the agents and the mechanisms for drawing the graphics.

Creator Agent

The **Creator Agent** is responsible for initially creating all the traffic lights and initializing them and then creating cars every n seconds and initializing them.

```
+!create_next_car : cars(C) <-
    .random(R);
    RInt = (R * 8) - ((R * 8) mod 1);
    !get_spawn_position(RInt, PosX, PosY, D);
    .concat("car_", C, N);
    .create_agent(N, "car_agent.asl");
    .send(N, achieve, start(PosX, PosY, D));
    -+cars(C + 1);
    .wait(2000);
    !create_next_car.
```

```
+!start(PosX, PosY, D) <-
    .my_name(Me);
    .broadcast(achieve, ask_position);
    spawn_car(PosX, PosY, Me, D).
```

```
+!check_target(PosX, PosY) : not(position(PosX, PosY)[source(Other)]) &
    (Other \== percept) & position(X, Y)[source(percept)] <-
    if (tl(S, X, Y)) {
        .concat("traffic_light_", S, TL);
        .send(TL, askOne, is_green(GREEN), is_green(GREEN));
        if (not(GREEN)) {
            !check_target(PosX, PosY);
        } else {
            !go(PosX, PosY);
        }
    } else {
        !go(PosX, PosY);
    }.
```

Car Agent

The Car Agent is initialized by the Creator. At the beginning it asks for the position of all the other cars, then it starts its path. At each step it checks if it can go forward (so if there is a car in front or if it is on a traffic light) and decides whether to stay still or move. At the end of the path, the agent ends its execution.

Traffic Light Agent

The Traffic Light Agent is initialized by the Creator. The various traffic lights within the same neighborhood cycle regularly, but the duration of the green light varies depending on the cars in the queue: the traffic lights in fact receive signals of the movements of the cars, and only the cars they are interested in are saved; when it is time to turn green, the duration varies depending on the cars saved.

```
+!cycle(green) <-
    .my_name(Me);
    update_traffic_light(green, Me);
    .count(position(_, _), N);
    if (N = 0) {
        !cycle(red);
    } else {
        TIME = N * 1000;
        .wait(TIME);
        !cycle(yellow);
    }.

+!cycle(yellow) <-
    .my_name(Me);
    update_traffic_light(yellow, Me);
    .wait(500);
    !cycle(red).

+!cycle(red) : number(N) <-
    .my_name(Me);
    update_traffic_light(red, Me);
    .wait(1000);
    if (not(N = 0) & N mod 4 = 3) {M = N - 3;}
    else { M = N + 1;};
    .concat("traffic_light_", M, Res);
    .send(Res, achieve, cycle(green)).
```

```java
@Override
public void init(final String[] args) {
    model = new TrafficModelImpl();
}

@Override
public Set<Literal> getPercepts(String agName) {
    return this.model.getPercepts(agName);
}

@Override
public boolean executeAction(final String ag, final Structure action) {
    String actionName = action.getFunctor();
    switch (actionName) {
        case Actions.SPAWN_CAR:
            ...
        case Actions.SPAWN_TRAFFIC_LIGHT:
            ...
        case Actions.UPDATE_TRAFFIC_LIGHT:
            ...
        case Actions.MOVE_CAR:
            ...
        case Actions.REMOVE_CAR:
            ...
        default:
            ...
    }
}
```

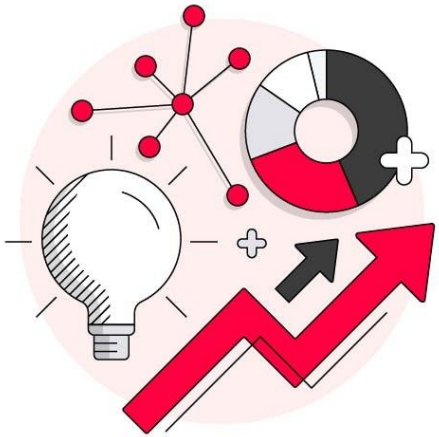
Traffic Environment

The Traffic Environment represents the Jason environment. Here some methods are overridden: init (where the model is initialized), getPercepts (where a certain agent's perceptions are retrieved) and executeAction (where the actions of the people on the environment are managed). Inside the latter, the model is updated and the communication with the GUI takes place.

Traffic GUI

The Traffic GUI has a list of car models and a list of tile models, so that they can be drawn at each cycle. The methods called by the environment modify these lists (obviously managing concurrency), so that the drawing cycles always update based on the current values. In particular, for the car movement, respond to the environment once the drawing is finished.

```
@Override
public void moveCar(int carId, int posX, int posY, int dir) {
    synchronized (cars) {
        for (var car : cars) {
            if (car.getId() == carId) {
                logMessage(...);
                ScheduledExecutorService scheduler =
                    Executors.newSingleThreadScheduledExecutor();
                scheduler.scheduleAtFixedRate(() -> {
                    boolean finished = car.move(posX, posY);
                    if (finished) {
                        environment.notifyAnimationFinished(...);
                        car.setPosition(posX, posY);
                        scheduler.shutdown();
                    }
                }, 0, 30, TimeUnit.MILLISECONDS);
                break;
            }
        }
    }
}
```



CONCLUSIONS

The project allowed me to explore agent-based programming, and to implement many new concepts and experiment a lot. The application is very expandable: you can add a large variety of features, such as double lanes, overtaking, roundabouts, etc., also experimenting with further aspects of agent-based programming.