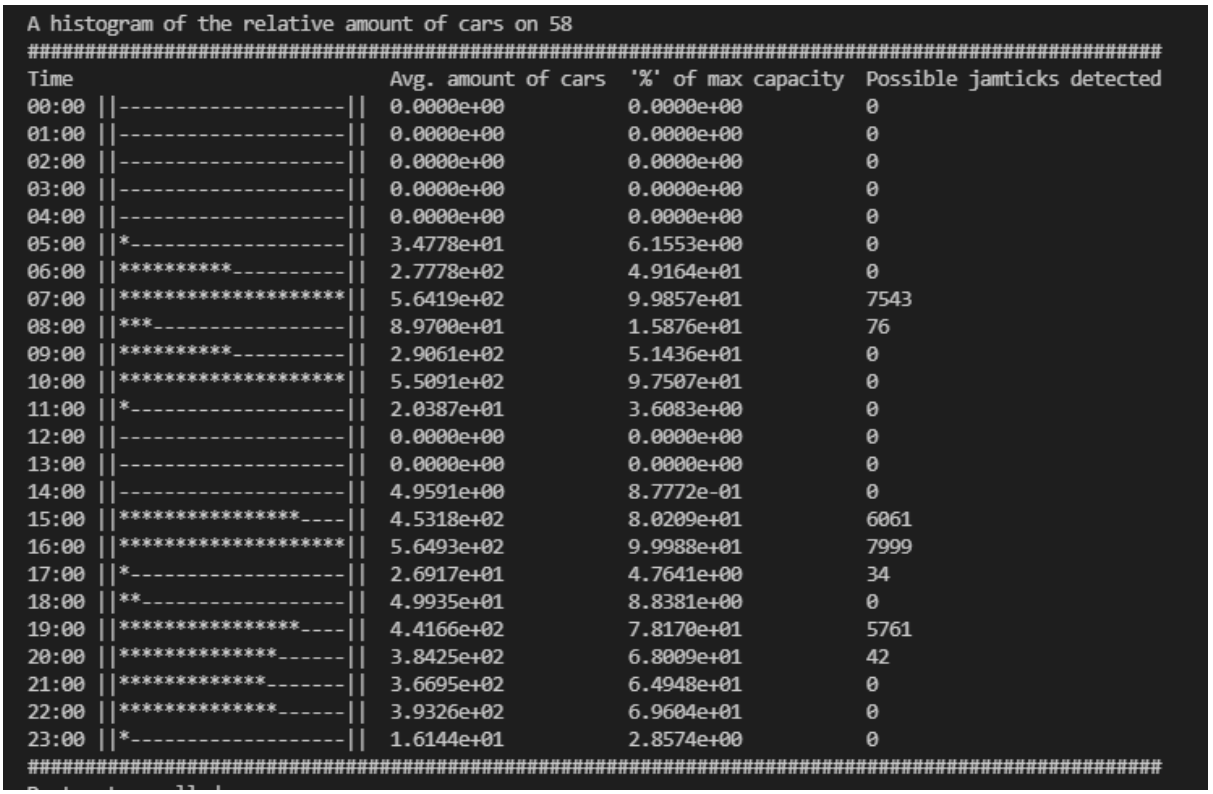
Project documentation

Traffic simulator - Group 1



```
A histogram of the relative amount of cars on 58
#################################################################################################
Time                          Avg. amount of cars  '%' of max capacity  Possible jamticks detected
00:00 ||--------------------||  0.0000e+00          0.0000e+00          0
01:00 ||--------------------||  0.0000e+00          0.0000e+00          0
02:00 ||--------------------||  0.0000e+00          0.0000e+00          0
03:00 ||--------------------||  0.0000e+00          0.0000e+00          0
04:00 ||--------------------||  0.0000e+00          0.0000e+00          0
05:00 ||*-------------------||  3.4778e+01          6.1553e+00          0
06:00 ||*********-----------||  2.7778e+02          4.9164e+01          0
07:00 ||********************||  5.6419e+02          9.9857e+01          7543
08:00 ||***-----------------||  8.9700e+01          1.5876e+01          76
09:00 ||*********-----------||  2.9061e+02          5.1436e+01          0
10:00 ||********************||  5.5091e+02          9.7507e+01          0
11:00 ||*-------------------||  2.0387e+01          3.6083e+00          0
12:00 ||--------------------||  0.0000e+00          0.0000e+00          0
13:00 ||--------------------||  0.0000e+00          0.0000e+00          0
14:00 ||--------------------||  4.9591e+00          8.7772e-01          0
15:00 ||***************-----||  4.5318e+02          8.0209e+01          6061
16:00 ||********************||  5.6493e+02          9.9988e+01          7999
17:00 ||*-------------------||  2.6917e+01          4.7641e+00          34
18:00 ||**------------------||  4.9935e+01          8.8381e+00          0
19:00 ||***************-----||  4.4166e+02          7.8170e+01          5761
20:00 ||*************-------||  3.8425e+02          6.8009e+01          42
21:00 ||*************-------||  3.6695e+02          6.4948e+01          0
22:00 ||*************-------||  3.9326e+02          6.9604e+01          0
23:00 ||*-------------------||  1.6144e+01          2.8574e+00          0
#################################################################################################
```

Nikita Tarutin, Emma Rauhala, Alexey Serous, Nikolai Semin
Aalto University, 2021

Table of contents

1. Overview

The aim of this software project was to build a system that can simulate city traffic throughout a day. The software has a modular structure where each and every class is supposed to be able to function separately and their interaction is what produces the simulation output.
The software is structured so that it is possible for the end user to provide various infrastructural input and then simulate city traffic with varying population sizes and timescales. By default, the initial parameters are set so that it best serves simulations of one day (24 h), and the default starttime is set to 00:00 at midnight.
The output of the software is a csv file containing the average hourly traffic details of all input roads. In addition, the user can specify a specific road to analyze. For this road the hourly averages are outputted as a relative histogram and possible traffic jams are detected separately.

2. Software structure

The software was designed to have a modular structure so that each group member would be able to work independently on their respective classes. There are 7 main classes, some of which are divided into smaller, inherited subclasses as in the listing below:

Class	**Subclasses**	**Functionality and containers**
RoadObjectClass	RoadLineClass	std::vector<Vehicle*> m_cells A container that represents car-places on a road. If there is no car, the value is nullptr. The class is to move vehicles along cells until they reach the end crossroad
	CrossroadClass ● TrafficLightCrossroad ● UnregulatedCrossroad ● BuildingExitCrossroad	std::vector<RoadObjectClass*> m_exitRoads; std::map<RoadObjectClass*, Vehicle*> m_entryRoads; The containers represent what roads it is possible to go to from this crossroad and what roads end up at this crossroads. A crossroad checks if there is a car in an entry road, if it is allowed to go to the desired exit road and if there is free space in the destination road. If yes, pushes it to the desired exit road.
	Building ● RecreationalBuilding ● CommercialBuilding ● ResidentialBuilding ● IndustrialBuilding	std::vector<Vehicle*> vehicles_; std::vector<Person*> people_; The class serves as a container for people and their vehicles and is used to alter the people's state at each timestep, according to the type of the Building.
Person		Person's specific need and habits specified by their internal parameters serve as a driving force for the city traffic. They have no containers but a huge variety of getters and setters to access their parameters.
Vehicle	Car	Vehicle serves as a baseclass for different types of transport in the city. In the current implementation, only personal vehicles (Cars) are supported.
Navigator		Used for pathfinding in the city. Stores the graph-information of the city infra in std::vector<RoadLineClass*> roads_; std::vector<CrossroadClass*> crossroads_; and is used to produce a route std::vector<RoadLineClass*> from Crossroad to Crossroad.

CityClass		std::map<std::string,CrossroadClass*> m_crossroads; std::vector<std::pair<std::string,RoadLineClass*>> m_roads; std::vector<std::pair<std::string,Building*>> m_RECbuildings; std::vector<std::pair<std::string,Building*>> m_RESbuildings; std::vector<std::pair<std::string,Building*>> m_INDbuildings; std::vector<std::pair<std::string,Building*>> m_COMbuildings; The class represents the geographical data provided by user: -Roads -Crossroads -Buildings
Simulation		A simple class to take care of the simulation flow. Has two methods: Init(): initializes random population based on the desired number, randomly assigns necessary buildings to every person. Simulate(): calls every object to perform the next action until the simulation time is reached and saves output every hour.

1. Input format and city infrastructure

The input file is to be a plain text containing the following syntax.

1.1. Crossroads

The declaration of crossroads is to start with the line containing only the following text:

CROSSROADS

Then, every line has to represent a crossroad with the following parameters separated by spaces:

CROSSROADTYPE CROSSROADNAME COORDINATE X COORDINATE Y TRAFFICLIGHTTIME

Where:

- CROSSROADTYPE can be B, T or U, (First letter in the name of the desired type)
- CROSSROADNAME is a user-defined name, any string without spaces
- COORDINATE X (and Y) is an integer value representing the position of a crossroad on the map in meters

- TRAFFICLIGHTTIME is an integer time in seconds after which the lights are switched. The field is optional, the default value is 30 seconds. Other crossroad types do not need this field.

Example lines:
- B cr9 1470 -20
- T crossroad10 1700 0 50

1.2. Roads

The declaration of roads is to start with the line containing only the following text:

ROADS

Then, every line has to represent a road with the following parameters separated by spaces:

ROADNAME STARTCROSSROADNAME ENDCROSSROADNAME HAVEYIELDSIGN SPEEDLIMIT

Where:
- ROADNAME is a user-defined name, any string without spaces.
- STARTCROSSROADNAME (and END) is the name of the crosroad that the road starts with.
- HAVEYIELDSIGN needs to have HASTRIANGLE if there is a yield triangle sign in this road. If there is no such string, the road does not have the sign.
- SPEEDLIMIT is an integer value in km/h to act as a speed limit. The default (if empty) is 40. The software can have only the limits divisible by 40, like 80, 120, which nearly corresponds to real life.

Example lines:
- road28 cr10 cr61
- line29 cr29 cr30 HASTRIANGLE
- road30 cr29 cr60 120

NOTE: a reverse direction road is added automatically for every road.

1.3. Buildings

The declaration of buildings is to start with the line containing only the following text:

BUILDINGS

Then, every line has to represent a building with the following parameters separated by spaces:

BUILDINGTYPE BUILDINGNAME EXITCROSSROADNAME MAXCAPACITY

Where:

- BUILDINGTYPE is one of the following:
    - RES for ResidentialBuilding
    - REC for RecreationalBuilding
    - COM for CommercialBuilding
    - IND for IndustrialBuilding
- BUILDINGNAME is a user-defined name, any string without spaces
- EXITCROSSROADNAME is the name of the crossroad that building is located at, which must exist in crossroads and be of type BuildingExitCrossroad
- MAXCAPACITY is an integer value specifying the maximum Vehicle-capacity of the building

Example lines:

- RES home cr9 1000

2. Initialization

The software is structured in such a way that a user can load a city from a file and then run multiple simulations with different populations and time ranges. The *CityClass* is designed to load the city infrastructure objects (crossroads, roads and buildings) from a file and store them in it's containers. At this stage, each *Road* and each *Building* is connected to their respective *Crossroads* (start + end for Roads or exit for Buildings) via pointers, to form a graph-like structure. The graph-structure is later used for pathfinding by *Navigator*.
The *Simulation::Init()* then, uses this CityClass-object, to first initialize the start state by creating a user defined amount of random People, assigning them a Vehicle and a Navigator, and putting them inside their homes (*ResidentialBuildings*). People are randomly assigned a workplace *(IndustrialBuilding*), a favorite food shop (*CommercialBuilding*) and a favorite place to spend their freetime (*RecreationalBuilding*). At construction, they also get random times for leaving- and coming home from work, from the time ranges 05:00-10:00 and 14:00 and 19:00 respectively. Their parameters for *money, hunger, happiness and food* are initialized using default values.

3. Simulation flow

The simulation is based on time ticks of 0.45 seconds: this corresponds to a time necessary for a car to cover its size distance (5m) with the speed of 40km/h. Thus, an hour of simulation has 8000 ticks. At each tick, *Simulation's Simulate()* will call *performTimestep*-function for each of the *CityClass* objects and each *Person.*

3.1. People

People (*Person*-class) and their needs and habits are the driving force of the simulation process. At each tick, a *Person* decides, based on their habits (working) and needs (hunger, happiness, money) where they want to go next - or if they possibly need to stay where they are. A *Person* is always forced to go to work on their personally set ticktime, and stay there

until their set time-to-leave comes. If a person should not be able to leave to work at their assigned ticktime, due to excess traffic, they are set to wait till the next tick and try again. To increase randomness, there is however one exception: if a person happens to already be on the road to somewhere else as their work shift should start, they will just skip the day of work. At ticks where a person is not supposed to be at work, they are free to go either shopping, freetime activities, or home, according to their current needs. At each tick, a *Person's performTimestep(unsigned int tickTime)* will, in addition to increasing their hunger and decreasing their happiness, call their *set_destination(unsigned int tickTime)*-method. *Set_destination* is used to evaluate if and where a person wants to go. If they want, and are allowed to go, they will call *Building's* respective *TakeVehicleAndLeave(Person* person)* described in the next section.

3.2. Buildings

People - if not currently on the road on their way to somewhere - are stored in *Building's* containers. The buildings have access to people's internal variables (*money, hunger, happiness and food)* through getters and setters, which are used to alter the state of the person, according to the type of the building. While at their home (*ResidentialBuilding*) people will increase their happiness and decrease their hunger by eating the food they have. At work *(IndustrialBuildings)* people will decrease their happiness, but gain money. In *CommercialBuildings* they lose some of their money to gain food and in *RecreationalBuildings* (sports,culture etc.) they lose money to gain happiness.
Buildings are also the unit to set people on the move when required. Once a Person wants to leave, they will call a *Building's TakeVehicleAndLeave(Person* person)*-method. This method then,assigns the *Person* a *Vehicle* and evaluates whether it is currently possible to push the Vehicle to their BuildingExitCrossroad, and eventually send them on the road. If it is possible, the *Person's* personal *Navigator* will be asked to produce a route to their wanted destination, and this route is then passed on to their assigned *Vehicle*. After this, pointers to both the *Person* and the *Vehicle* will be removed from the *Building's* containers.

3.3. RoadObjectClass

Since *Roads, Buildings* and *Crossroads* need to be aware of each other and utilize mostly the same functions, it was decided to avoid cross-dependency by introducing a single base class RoadObjectClass, that stores the declarations of the required methods. So, the roads and crossroads can interact with each other by calling a relevant *RoadObjectClass::function*, that is overridden in the object referred.
The key basic functions are:

- *void performTimeStep* - an object is to manipulate the cars moving them further in time
- *bool takeVehicle* - takes a Vehicle*, but first checks if it can be accepted. In other words, if there is enough space.

3.3.1. Roads

A RoadLineClass object represents a single line where the cars go from the start crossroad to the end crossroad (*performTimeStep()*). When a car reaches the end, the end

crossroad is asked to take it. If the end crossroad is busy and false is returned, all cars have to wait. This is how traffic jams are formed.

3.3.2. Crossroads

3.3.2.1. Abstract CrossroadClass

A crossroad is a point on a map with two coordinates x and y. It stores two containers with pointers to entry roads (the roads that end with this crossroad) and exit roads (the roads that start with this crossroad). The key function is bool *CheckRightToGo*, that takes the pointers to an entry and an exit road. Then, based on the crossroad type, it returns whether the desired turn is allowed.

When *performTimeStep()* is called, every entry road is checked for its right to go. Then, the cars (if any) in all allowed to go entry roads are tried to be pushed to their desired exit roads.

3.3.2.2. BuildingExitCrossroad

A crossroad that represents a connection between a building yard and a normal road. It is possible to connect several buildings and roads. However, the priority between real roads is decided randomly, so it is advised to have as few real roads as possible.

CheckRightToGo always returns true for real roads. As for the buildings, a car entering a real road always have to give way to all the cars before joining. That is why, for buildings, the functions first ensures that there are no cars on real roads, only then returns true.

3.3.2.3. UnregulatedCrossRoad

A typical crossroad without any traffic lights. The cars follow the right side rule. There is also an option for a road to have a yield-triangle sign, then it has the lowest priority irrespectively of its geographical position. *CheckRightToGo* ensures there are no cars on the right and that there is no triangle sign (both for the entry road and all roads on the right).

3.3.2.4. TrafficLightCrossroad

A typical traffic light crossroad. Every road is assigned its traffic light, for +/- opposite roads the same traffic light is used. *CheckRightToGo()* returns the light color (boolean). In addition, *performTimeStep()* is overridden. It calls the base class function, but also moves the time for traffic lights. When the time exceeds a certain value specified by the user, the lights are switched.

4. Navigation

To produce routes for *People* and *Vehicles*, we needed to implement pathfinding in our graph-like city. The class *Navigator* serves this purpose. The *Simulation* creates a *Navigator*, pointer to which is then assigned to each *Person* of the city. As the Persons want to go somewhere, they call their *Navigator* for a route from their current location to their destination. Navigator's *MakeRoute()* uses a specialized algorithm to produce the route and return it as a vector of pointers to *RoadObjects.*

During the project, two different algorithms were proposed for pathfinding: Djikstra's algorithm and A*.
The final version uses A*, a pathfinding algorithm that uses a heuristic - in our case the euclidean distance between crossroad coordinates - for efficiency and optimality. The weakness of A* however is, that it relies somewhat on the coordinates, and thus on the user specified input file, whereas Djikstra's would require no information about absolute location.

5. Output

The simulation process is set to produce hourly outputs. Every 8000 ticks (which corresponds to one hour), the simulator calculates the average amount of cars on each road during the past hour, and writes them into a file. Additionally, for the user specified road-to-be-analyzed, it counts the number of ticks where - during the respective hour - the road has been totally or almost totally full. This indicates a possible traffic jam on the road.
Finally, the program has two outputs: a histogram of the average amount of cars on the user specified analyze-road at each simulation hour (printed in cout) and an output.csv that contains the hourly averages of the amount of cars on all roads. The histogram is scaled so that a full bar corresponds to the daily maximum of cars on the road in question. An example histogram is shown on the cover page and the output csv is structured as shown below.

Time (starting hour)	**Roadname 1**	**Roadname N…**
0	*avg. amount of cars on road 1 during first hour =* $\frac{sum\ of\ cars\ on\ road\ 1\ over\ all\ ticks\ of\ first\ hour}{the\ number\ of\ ticks\ per\ hour\ (8000)}$	…
1	*avg. amount of cars on road 1 during second hour =* $\frac{sum\ of\ cars\ on\ road\ 1\ over\ all\ ticks\ of\ second\ hour}{the\ number\ of\ ticks\ per\ hour\ (8000)}$	…
…	…	…

3. Instructions for building and using the software

3.1. Basic command line interface

The simplest way to access the functionalities of the software is the basic command line interface. The software can be built with cmake simply by running the following commands in the project folder (traffic-simulator-1).

```
mkdir build
cd build
cmake ../
cmake –build ./
```

Then running .\simulation will start the program and make it ask for user input. The recommended way is to straightforwardly type i.e.

```
./simulation input_file/samplecity.txt 8000 25 output_file/output.csv 58
```

which will run a 24 h simulation with 8000 residents on the sample city, with a more specific analysis on a heavily trafficked road number 58.

3.2. GUI

The graphical user interface is done using Qt framework. The most crucial part is the map of the city being displayed after the simulation is done running: a user can see the city's structure: roads and crossroads and zoom in and out to get a better look.

From programming point of view, the GUI is implemented with Qt UI class that includes:

3.2.1. simulationView.h

The header file declares simulationView class that inherits from QMainWindow and includes Q_OBJECT macro that enables use of Signals and Slots that will be described later.

To store coordinates of the objects to be rendered two containers are used:

- **std::vector<std::pair<int, int>> crossRoads_**
- **std::vector<std::pair<std::pair<int, int>,std::pair<int, int>>> roads_;**

*crossRoads_* vector stores coordinates of the crossroads and *roads_* stores coordinate of end and start of a road.

GraphicsView framework is used to draw objects inside the window. The objects are lines that represent roads and circles that represent crossroads.

3.2.2. simulationView.cpp

Contains implementation of the simulationViewconstructor, including the constructor and the following methods:

- **void drawCity()**
  Loops through data inside *crossRoads_* and *roads_* containers
  and renders the city map
- **void getCross(std::vector<std::pair<int,int>>)** and **void getRoads(std::vector< std::pair<std::pair<int, int>,std::pair<int, int>>>);**
  Is a slot to retrieve data of city roads and crossroads from cityClass object and save it.

3.2.3. simulationview.ui

Contains visualisation of the window. More specifically, GraphicsView and how it should keep its size in relation to the window (vertical and horizontal alignment).

If any errors occur due to the *ui_simulationview.h* file, user should ensure that *CMAKE_AUTOUIC* CMake flag is on. The remark might be irrelevant to the Qmake build system.

As was mentioned earlier, slots were used, However, these require signals too; hence, to send data from cityClass object, the cityClass was made to inherit from QObject and use Q_OBJECT macro that enables signals and slots.

NB! moc_cityclass.cpp should be included at the end of cityclass.cpp and the CMAKE_AUTOMOC CMake flag should be on.

Two signals are implemented in *cityclass*:

- void sendX(std::vector<std::pair<int, int>>)
- void sendR(std::vector<std::pair<std::pair<int, int>,std::pair<int, int>>>)

4. Testing

For unit testing Google Tests framework was used. Even though it allows a lot of opportunities like Suites, basic functionality to test classes methods was used. Tests cover most of essential classes.

Usage: after make file was run to initialize executable, it's possible to run test by shell script: ./tests.sh

```
TEST(person_test, Constructor){
    Person guy1(n);
    Person guy2(n);
    Person guy3(n);
    EXPECT_EQ( val1: guy1.get_id(),  val2: 1);
    EXPECT_EQ( val1: guy2.get_id(),  val2: 2);
    EXPECT_EQ( val1: guy3.get_id(),  val2: 3);
}


TEST(person_test, Money) {
Person guy(n);
    EXPECT_EQ( val1: guy.get_money(),  val2: 500);
    guy.remove_money( money: 500);
    EXPECT_EQ( val1: guy.get_money(),  val2: 0);
    guy.add_money( money: 10000);
    EXPECT_EQ( val1: guy.get_money(),  val2: 10000);
    EXPECT_EQ( val1: guy.has_money(),  val2: true);
}
```

Above is an example of two tests for Person class shown. It is important that Person instance is initialised inside one TEST; if one instance is needed for multiple tests, test suites would be a better choice. However, being unfamiliar with the framework initially and having limited time, it was decided just to use test for verifying classes' functionality.

5. Work log

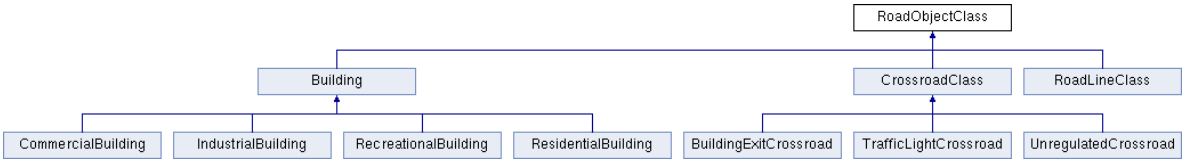
Week	Member	Work log
44	*Planning*	
45	Nikita	Basic Vehicle class
	Emma	Getting used to working with git, Initialized Building-class
	Alexey	Basic RoadLineClass
	Nikolai	Basic Person
46	Nikita	Basic Navigator class, its interaction with crossroads
	Emma	Further developing Building, adding subclasses of different building types
	Alexey	Basic RoadObjectClass, CrossroadClass, BuildingExitCrossroad
	Nikolai	Started looking oh how to connect and use Google Tests
47	Nikita	Implementing pathfinding algorithm based on Dijkstra algorithm
	Emma	Producing first sketches of Building-Person interaction
	Alexey	UnregulatedCrossroad
	Nikolai	Finalising Google Tests
48	Nikita	Integrating navigator into Person class, initializing and randomizing population in Simulation, debugging
	Emma	Writing a new Navigator, initializing Simulation-class and producing output, contributing to CityClass, initializing the project documentation
	Alexey	TrafficLightCrossroad, CityClass
	Nikolai	Persons behaviour
49	Nikita	Debugging software, tests, resolving git issues, documentation
	Emma	Debugging software, updating Person's behavior and interactions with Buildings, improving Simulation output, planning documentation structure
	Alexey	CityClass, debugging the software, applying fixes
	Nikolai	Finalising the GUI, investigating possibilities for more advanced GUI features

Appendix: Doxygen Class Reference

RoadObjectClass Class Reference abstract

An interface to be inherited by the RoadLineClass and CrossroadClass * contact information - Alexey Serous

Inheritance diagram for RoadObjectClass:



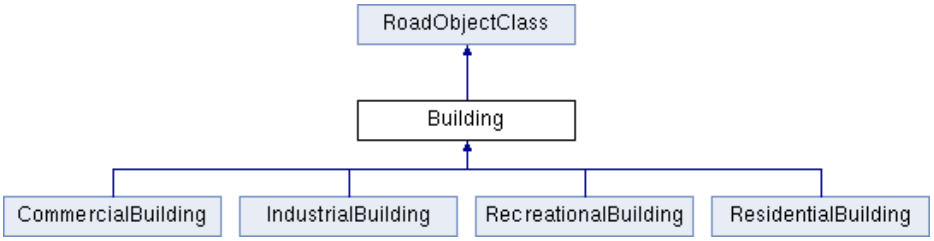
Public Member Functions

	RoadObjectClass (bool hasTriangle=false)
	RoadObjectClass (const **RoadObjectClass** &a)=delete
bool	**hasTriangle** () const
virtual bool	**takeVehicle** (**Vehicle** *ptrToCar, const **RoadObjectClass** *ptrToRoadObject)=0
virtual void	**performTimeStep** ()=0
RoadObjectClass &	**operator=** (const **RoadObjectClass** &a)=delete

Building Class Reference abstract

Abstract class Building (cannot be initialized). Works as a base class for RecreationalBuilding ResidentialBuilding IndustrialBuilding CommercialBuilding

Inheritance diagram for Building:



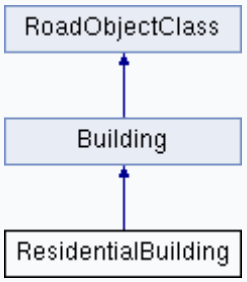
Public Member Functions

	Building (**CrossroadClass** *exitCrossRoad, unsigned int vehiclecapacity, const std::string &type)
	Building (const **Building** &a)=delete
bool	**takeVehicle** (**Vehicle** *ptrToCar, const **RoadObjectClass** *ptrToRoadObject) override
virtual void	**performTimeStep** () override=0
bool	**TakeVehicleAndLeave** (**Person** *person)
void	**TakePerson** (**Person** *person)
unsigned int	**GetID** () const
unsigned int	**GetVehicleCapacity** () const
const std::string &	**GetType** () const
CrossroadClass *	**GetExit** () const
std::vector< **Vehicle** * >	**GetVehicles** () const

std::vector< **Person** * >	**GetPeople** () const
Building &	**operator=** (const **Building** &a)=delete

ResidentialBuilding Class Reference

Inheritance diagram for ResidentialBuilding:

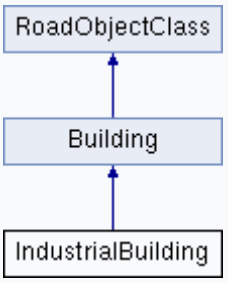


Public Member Functions

	ResidentialBuilding (const ResidentialBuilding &a)=delete
ResidentialBuilding &	**operator= (const ResidentialBuilding &a)=delete**
	ResidentialBuilding (CrossroadClass *exitCrossRoad, unsigned int vehiclecapacity)
virtual void	**performTimeStep () override**

IndustrialBuilding Class Reference

Inheritance diagram for IndustrialBuilding:

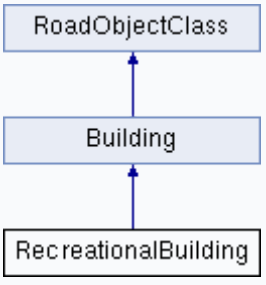


Public Member Functions

	IndustrialBuilding (CrossroadClass *exitCrossRoad, unsigned int vehiclecapacity)
	IndustrialBuilding (const IndustrialBuilding &a)=delete
IndustrialBuilding &	**operator= (const IndustrialBuilding &a)=delete**
virtual void	**performTimeStep () override**

RecreationalBuilding Class Reference

Inheritance diagram for RecreationalBuilding:

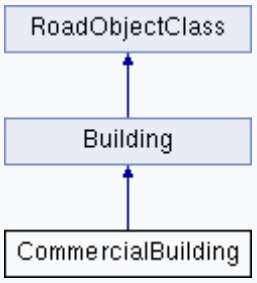


Public Member Functions

	RecreationalBuilding (CrossroadClass *exitCrossRoad, unsigned int vehiclecapacity)
	RecreationalBuilding (const RecreationalBuilding &a)=delete
RecreationalBuilding &	**operator= (const RecreationalBuilding &a)=delete**
virtual void	**performTimeStep () override**

CommercialBuilding Class Reference

Inheritance diagram for CommercialBuilding:



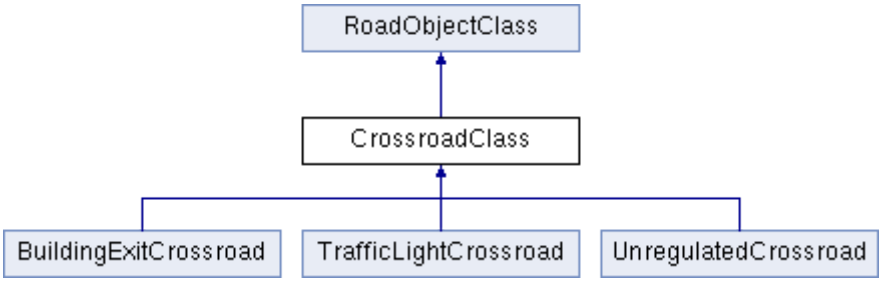
	CommercialBuilding (CrossroadClass *exitCrossRoad, unsigned int vehiclecapacity)
	CommercialBuilding (const CommercialBuilding &a)=delete
CommercialBuilding &	**operator= (const CommercialBuilding &a)=delete**

virtual void	**performTimeStep** () **override**

CrossroadClass Class Reference abstract

The class acts as a moderator between the roads * Consists of two containers * 1) <RoadObjectClass*, Vehicle*> for entry roads * 2) <RoadObjectClass*> for exit roads * If there is currently no vehicle standing on the position of entry roads, then it is nullptr * * * contact information - Alexey Serous

Inheritance diagram for CrossroadClass:



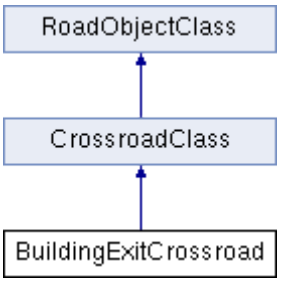
Public Member Functions

	CrossroadClass (int x, int y)
	CrossroadClass (const **CrossroadClass** &a)=delete
bool	**takeVehicle** (**Vehicle** *ptrToCar, const **RoadObjectClass** *ptrToRoadObject) override
void	**performTimeStep** () override
bool	**addEntryRoad** (const **RoadObjectClass** *ptrToEntryRoad)
bool	**addExitRoad** (**RoadObjectClass** *ptrToExitRoad)

virtual void	**addEntryAngle** (const **RoadObjectClass** *ptrToEntryRoad, double angle)
void	**addExitAngle** (**RoadObjectClass** *ptrToExitRoad, double angle)
void	**setRealEntryRoad** (**RoadObjectClass** *ptrToEntry)
int	**getX** ()
int	**getY** ()
CrossroadClass &	**operator=** (const **CrossroadClass** &a)=delete
bool	**operator==** (const **CrossroadClass** &a)
bool	**operator==** (**CrossroadClass** &a)

BuildingExitCrossroad Class Reference

Inheritance diagram for BuildingExitCrossroad:



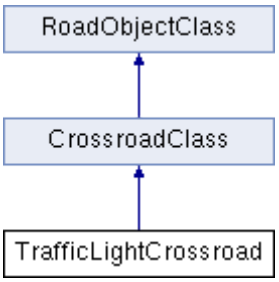
Public Member Functions

BuildingExitCrossroad (int x, int y)

	BuildingExitCrossroad (const **BuildingExitCrossroad** &a)=delete
BuildingExitCrossroad &	**operator=** (const **BuildingExitCrossroad** &a)=delete

TrafficLightCrossroad Class Reference

Inheritance diagram for TrafficLightCrossroad:

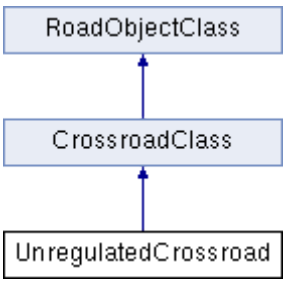


Public Member Functions

	TrafficLightCrossroad (int x, int y, double trafficLightTimeInSeconds=defaultTrafficLightTime)
	TrafficLightCrossroad (const **TrafficLightCrossroad** &a)=delete
void	**performTimeStep** () override
void	**addEntryAngle** (const **RoadObjectClass** *ptrToEntryRoad, double angle) override
TrafficLightCrossroad &	**operator=** (const **TrafficLightCrossroad** &a)=delete

UnregulatedCrossroad Class Reference

Inheritance diagram for UnregulatedCrossroad:



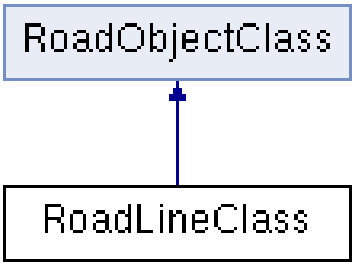
Public Member Functions

	UnregulatedCrossroad (int x, int y)
	UnregulatedCrossroad (const **UnregulatedCrossroad** &a)=delete
UnregulatedCrossroad &	**operator=** (const **UnregulatedCrossroad** &a)=delete

RoadLineClass Class Reference

One line of a road, a piece between two crossroads * In case there is no crossroad (e.g. a road starts), then it is nullptr * * The road is divided in cells (total number of m_size) * Every cell contains a pointer to a car. If the cell is currently empty, * then the value is nullptr * * When moved by a timestep, every car starting from the end is moved * one cell to the next one towards the end. If a cell is occupied (e.g. the * end cross road is busy, red-traffic light), no movement is performed. * The end car is gived to the end cross-road as a start of the moving function

Inheritance diagram for RoadLineClass:



Public Member Functions

	RoadLineClass (CrossroadClass *start, CrossroadClass *end, bool hasTriangle=false, std::size_t speedLimit=40)
	RoadLineClass (const RoadLineClass &a)=delete
bool	takeVehicle (Vehicle *ptrToCar, const RoadObjectClass *ptrToRoadObject) override
void	performTimeStep () override
CrossroadClass *	GetStart () const
CrossroadClass *	GetEnd () const
const size_t	GetSize () const
std::size_t	getNumberOfCars () const
RoadLineClass &	operator= (const RoadLineClass &a)=delete

Navigator Class Reference

Public Member Functions

	Navigator (std::vector< **RoadLineClass** * > roads_, std::vector< **CrossroadClass** * > crossroads_)
std::vector< **RoadLineClass** * >	**MakeRoute** (**CrossroadClass** *start, **CrossroadClass** *finish)

Person Class Reference

Base class for persons inhabiting a city. Might become abstract when scaling * Contact info - Nikolai.

Public Member Functions

	Person (**Navigator** *n, **Building** *home=nullptr, **Building** *fav_recreational=nullptr, **Building** *fav_commercial=nullptr, **Building** *work=nullptr)
unsigned int	**get_id** () const
double	**get_money** () const
int	**get_time_leaving** () const

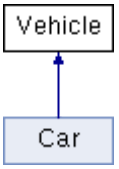
int	**get_time_coming** () const
int	**get_happiness** () const
int	**get_food** () const
int	**get_hunger** () const
Building *	**get_destination** () const
void	**reset_destination** ()
Navigator *	**getNavigator** ()
bool	**is_hungry** () const
bool	**is_happy** () const
bool	**has_money** () const
void	**eat_food** (int)
void	**add_food** (int)
void	**add_money** (int)
void	**remove_money** (int)
void	**increase_happiness** (int)
void	**increase_hunger** (int)
void	**decrease_happiness** (int)
void	**set_destination** (unsigned int)

void	**set_current_place** (**Building** *)
Building *	**get_current_place** () const
void	**performTimeStep** (unsigned int)

Vehicle Class Reference

A base Vehicle class. Vehicle objects are stored within city objects, occupying their load capacity with the value of size_. Contact info - Nikita

Inheritance diagram for Vehicle:



Public Member Functions

	Vehicle (int size)
const int &	**GetSize** () const
void	**setRoute** (std::vector< **RoadLineClass** * > newRoute)
std::vector< **RoadLineClass** * >	**getRoute** ()
void	**setDestination** (**RoadObjectClass** *ptrToBuilding)
void	**removePassenger** (**Person** *person)
void	**takePassenger** (**Person** *person)

std::vector< Person * >	**getPassengers** ()
RoadObjectClass *	**getDestination** ()
RoadObjectClass *	**FindNextRoad** (CrossroadClass *currentCross)

Simulation Class Reference

A class that performs a simulation by reading input file, initializing a city, processing results and writing them to output file.

Public Member Functions

Simulation (CityClass *city, unsigned int npeople, double endtime, std::string outputfile, std::string RoadToAnalyze)

Init ()

	Simulate ()

CityClass Class Reference

A class that represents a city with roads, crossroads, buildings, etc. * Contact info - Alexey

Public Member Functions

	CityClass (std::string fileName)
std::map< std::string, **CrossroadClass** * > &	**GetCrossroads** ()
std::vector< std::pair< std::string, **RoadLineClass** * > > &	**GetRoads** ()
std::vector< std::pair< std::string, **Building** * > > &	**GetRECBuildings** ()
std::vector< std::pair< std::string, **Building** * > > &	**GetRESBuildings** ()
std::vector< std::pair< std::string, **Building** * > > &	**GetINDBuildings** ()

```
std::vector< std::pair<
std::string, Building * > > & GetCOMBuildings ()
```