

## Traffic Simulator

Generated by Doxygen 1.9.1



<b>1 Main Page</b>	<b>1</b>
1.1 Introduction	1
1.2 Instructions on how to use the program	1
1.3 JSON file template for city loading	2
<b>2 Hierarchical Index</b>	<b>3</b>
2.1 Class Hierarchy	3
<b>3 Class Index</b>	<b>5</b>
3.1 Class List	5
<b>4 Class Documentation</b>	<b>7</b>
4.1 Analysis Class Reference	7
4.1.1 Detailed Description	7
4.1.2 Constructor & Destructor Documentation	7
4.1.2.1 Analysis()	7
4.1.3 Member Function Documentation	8
4.1.3.1 ExportToCSV()	8
4.1.3.2 GenerateHourlyHistogram()	8
4.1.3.3 GetData()	8
4.1.3.4 SpecifyRoad()	9
4.2 Building Class Reference	9
4.2.1 Detailed Description	10
4.2.2 Constructor & Destructor Documentation	10
4.2.2.1 Building()	10
4.2.3 Member Function Documentation	10
4.2.3.1 Draw()	10
4.2.3.2 GetLocation()	11
4.2.3.3 GetName()	11
4.2.3.4 GetType()	11
4.3 Car Class Reference	12
4.3.1 Detailed Description	12
4.3.2 Constructor & Destructor Documentation	13
4.3.2.1 Car()	13
4.3.3 Member Function Documentation	13
4.3.3.1 AddEvent()	13
4.3.3.2 AtDestination()	13
4.3.3.3 CarInFront()	14
4.3.3.4 CheckIntersection()	14
4.3.3.5 Dijkstra()	14
4.3.3.6 Draw()	15
4.3.3.7 GetDirection()	15
4.3.3.8 GetIntersection()	15

4.3.3.9 InitializeSchedule()	16
4.3.3.10 LanelFree()	16
4.3.3.11 SetColor()	16
4.3.3.12 SetDestination()	17
4.3.3.13 SetDirection()	17
4.3.3.14 SetSpeedLimit()	17
4.3.3.15 Update()	18
4.3.3.16 YieldRight()	18
4.4 Cell Class Reference	18
4.4.1 Detailed Description	19
4.4.2 Constructor & Destructor Documentation	19
4.4.2.1 Cell()	19
4.4.3 Member Function Documentation	19
4.4.3.1 Draw()	19
4.4.3.2 GetType()	20
4.4.3.3 GetX()	20
4.4.3.4 GetY()	20
4.4.3.5 IsOccupied()	21
4.4.3.6 Occupy()	21
4.5 City Class Reference	21
4.5.1 Detailed Description	23
4.5.2 Constructor & Destructor Documentation	23
4.5.2.1 City()	23
4.5.3 Member Function Documentation	23
4.5.3.1 AddBuilding()	23
4.5.3.2 AddClock()	23
4.5.3.3 AddEvent()	24
4.5.3.4 AddIntersection()	24
4.5.3.5 AddPersonAndCar()	24
4.5.3.6 AddRoad()	25
4.5.3.7 AddTrafficLight()	25
4.5.3.8 ChooseRoad()	25
4.5.3.9 DrawBuildings()	26
4.5.3.10 DrawCars()	26
4.5.3.11 DrawIntersections()	26
4.5.3.12 GetBuildingNodes()	27
4.5.3.13 GetCars()	27
4.5.3.14 GetGrid()	27
4.5.3.15 GetIntersection()	27
4.5.3.16 GetNode()	28
4.5.3.17 GetRoads()	28
4.5.3.18 IsBusy()	28

4.5.3.19 IsValidBuilding()	29
4.5.3.20 IsValidIntersection()	29
4.5.3.21 IsValidPerson()	29
4.5.3.22 IsValidRoad()	30
4.5.3.23 PrintCity()	30
4.5.3.24 TimeUntilNextEvent()	30
4.5.3.25 UpdateCars()	31
4.5.3.26 UpdateIntersections()	31
4.6 Commercial Class Reference	31
4.6.1 Detailed Description	32
4.6.2 Constructor & Destructor Documentation	32
4.6.2.1 Commercial()	32
4.6.3 Member Function Documentation	33
4.6.3.1 Draw()	33
4.7 Event Class Reference	33
4.7.1 Detailed Description	33
4.7.2 Constructor & Destructor Documentation	33
4.7.2.1 Event()	34
4.7.3 Member Function Documentation	35
4.7.3.1 CreateSchedule()	35
4.8 Grid Class Reference	35
4.8.1 Detailed Description	36
4.8.2 Constructor & Destructor Documentation	36
4.8.2.1 Grid()	36
4.8.3 Member Function Documentation	36
4.8.3.1 GetCell()	36
4.8.3.2 GetNeighborCells()	37
4.8.3.3 GetSizeX()	37
4.8.3.4 GetSizeY()	37
4.9 Industrial Class Reference	38
4.9.1 Detailed Description	38
4.9.2 Constructor & Destructor Documentation	38
4.9.2.1 Industrial()	38
4.9.3 Member Function Documentation	39
4.9.3.1 Draw()	39
4.10 Intersection Class Reference	39
4.10.1 Detailed Description	40
4.10.2 Constructor & Destructor Documentation	40
4.10.2.1 Intersection()	40
4.10.3 Member Function Documentation	40
4.10.3.1 AddTrafficLight()	40
4.10.3.2 AllowHorizontal()	40

4.10.3.3 AllowVertical()	41
4.10.3.4 Draw()	41
4.10.3.5 GetLocation()	41
4.10.3.6 Update()	41
4.11 InvalidCityException Class Reference	42
4.11.1 Detailed Description	42
4.11.2 Constructor & Destructor Documentation	42
4.11.2.1 InvalidCityException()	42
4.11.3 Member Function Documentation	43
4.11.3.1 GetError()	43
4.12 Node Class Reference	43
4.12.1 Detailed Description	43
4.12.2 Constructor & Destructor Documentation	44
4.12.2.1 Node()	44
4.12.3 Member Function Documentation	44
4.12.3.1 AddConnection()	44
4.12.3.2 GetConnections()	44
4.12.3.3 GetLocation()	45
4.12.3.4 GetType()	45
4.13 Person Class Reference	45
4.13.1 Detailed Description	46
4.13.2 Constructor & Destructor Documentation	46
4.13.2.1 Person()	46
4.13.3 Member Function Documentation	46
4.13.3.1 AddEvent()	47
4.13.3.2 BuyCar()	47
4.13.3.3 GetCar()	47
4.13.3.4 GetLocation()	47
4.13.3.5 GetName()	48
4.13.3.6 GetPersonType()	48
4.13.3.7 GetResidence()	48
4.13.3.8 GetSchedule()	48
4.13.3.9 GetWorkplace()	49
4.13.3.10 InitializeSchedule()	49
4.13.3.11 isAtHome()	49
4.13.3.12 UpdateLocationFromCar()	49
4.14 Residential Class Reference	50
4.14.1 Detailed Description	50
4.14.2 Constructor & Destructor Documentation	50
4.14.2.1 Residential()	50
4.14.3 Member Function Documentation	51
4.14.3.1 Draw()	51

4.15 Road Class Reference . . . . .	51
4.15.1 Detailed Description . . . . .	52
4.15.2 Constructor & Destructor Documentation . . . . .	52
4.15.2.1 Road() . . . . .	52
4.15.3 Member Function Documentation . . . . .	52
4.15.3.1 GetEnd() . . . . .	52
4.15.3.2 GetSpeedLimit() . . . . .	52
4.15.3.3 GetStart() . . . . .	53
4.15.3.4 IsHorizontal() . . . . .	53
4.15.3.5 IsVertical() . . . . .	53
4.16 SimulationClock Class Reference . . . . .	53
4.16.1 Detailed Description . . . . .	54
4.16.2 Member Function Documentation . . . . .	54
4.16.2.1 GetDayNumber() . . . . .	54
4.16.2.2 GetElapsedTime() . . . . .	54
4.16.2.3 GetSimulationTime() . . . . .	55
4.16.2.4 SetSimulationSpeed() . . . . .	55
4.17 Simulator Class Reference . . . . .	55
4.17.1 Detailed Description . . . . .	56
4.17.2 Member Function Documentation . . . . .	56
4.17.2.1 DrawSimulation() . . . . .	56
4.17.2.2 InputThread() . . . . .	56
4.17.2.3 LoadCity() . . . . .	57
4.17.2.4 SetCity() . . . . .	57
4.17.2.5 UpdateSimulation() . . . . .	57
4.18 TrafficLight Class Reference . . . . .	57
4.18.1 Detailed Description . . . . .	58
4.18.2 Constructor & Destructor Documentation . . . . .	58
4.18.2.1 TrafficLight() . . . . .	58
4.18.3 Member Function Documentation . . . . .	59
4.18.3.1 GetGreenDuration() . . . . .	59
4.18.3.2 GetLocation() . . . . .	59
4.18.3.3 GetRedDuration() . . . . .	59
4.18.3.4 GetStatus() . . . . .	59
4.18.3.5 GetYellowDuration() . . . . .	60
4.19 Visualization Class Reference . . . . .	60
4.19.1 Detailed Description . . . . .	60
4.19.2 Constructor & Destructor Documentation . . . . .	60
4.19.2.1 Visualization() . . . . .	60
4.19.3 Member Function Documentation . . . . .	61
4.19.3.1 GetWindow() . . . . .	61
4.19.3.2 PrintCars() . . . . .	61

4.19.3.3 PrintGrid()	61
----------------------	----

<b>Index</b>	<b>63</b>
--------------	-----------



# Chapter 1

## Main Page

### 1.1 Introduction

The purpose of this program is to simulate and analyze the traffic of a city model inputted as a JSON file.

This page contains instructions on how to use the program. The same information can also be found in the README file of the project.

### 1.2 Instructions on how to use the program

**Dependencies:** The simulator might require some additional libraries that are required to be installed through terminal, so run these commands:

- Linux: `sudo apt-get install libxrandr-dev libxcursor-dev libsFML-dev`
- MacOS: `brew install sfml`

**Building and running the program:**

- Clone <https://version.aalto.fi/gitlab/karhuj5/traffic-simulator-henrik-toikka-4>.↵  
`git`
- Navigate to the folder `cd build`
- Generate makefile with CMake: `cmake ..`
- Build the file in the same directory `make`
- Navigate to the root folder `cd ..`
- Run the file `build/./main`

**How to use the simulation:** In the beginning, the program asks for a JSON file name. If the file is not found in the root folder, or if it is invalid, the program will not proceed and it will ask for another file (see the JSON file template and requirements below). After the JSON file has been successfully loaded, the program asks for a road index to analyze during the simulation. Index *i* chooses the *i*:th road listed in the JSON file to be analyzed. After choosing the road to be analyzed, the program asks for simulation speed (1, 2, 4, 8, or 16). After this, the program asks if the gui should be enabled or not, and then it will start the simulation.

During the simulation you can use the commands:

- `status` Prints the current simulation time to the terminal
- `exit` Exits the program
- `analyze` Prints the usage data of the *i*:th road to the terminal
- `export` Exports the road analysis data to a CSV file

## 1.3 JSON file template for city loading

- The city is loaded from a JSON file which should be located in the root folder of the project. A template for the JSON files is provided below. If the JSON file is invalid, the program will continue asking for a new file until the file is valid. The root folder of this project contains two example JSON files: city.json and smallcity.json.
- Requirements:
  - The order of the elements must be the same as in the template.
  - All the locations of the buildings, intersections, roads, and traffic lights must be inside city boundaries.
  - The building type must be one of the following types: "Residential", "Industrial", "Gym", "Shop", or "Restaurant".
  - The starting and ending points of roads must be either intersections or buildings.
  - The PersonType parameter of person objects must be one of the following types: "Lazy", "Active", "Neutral", "Gentleman", or "Angry".
  - The workplace of a person must be one of the [Industrial](#) buildings added to the city (i.e. the name of the workplace must be found in the added [Industrial](#) buildings' names)
  - The home of a person must be one of the [Residential](#) buildings added to the city (i.e. the name of the home building must be found in the added [Residential](#) buildings' names)
  - The traffic lights must have the same coordinates as one of the intersections.
  - The objects must be placed in locations where there are no other objects (excluding traffic lights, which must be placed on top of intersections).

### The template:

```
{
  "x": vertical_city_size(int),
  "y": horizontal_city_size(int),
  "buildings": [
    { "name": [ "building type", [x-coordinate(int), y-coordinate(int)] ] }
  ],
  "intersections": [
    [x-coordinate(int), y-coordinate(int)]
  ],
  "roads": [
    [starting_x-coordinate(int), starting_y-coordinate(int), ending_x-coordinate(int),
      ending_y-coordinate(int)]
  ],
  "persons": [
    [ "name", "PersonType", "name of workplace", "name of home building" ]
  ],
  "trafficLights": [
    [[x-coordinate(int), y-coordinate(int)], red_and_green_light_duration(int), yellow_light_duration(int)]
  ]
}
```

## Chapter 2

# Hierarchical Index

### 2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Analysis . . . . .	7
Building . . . . .	9
Commercial . . . . .	31
Industrial . . . . .	38
Residential . . . . .	50
Car . . . . .	12
Cell . . . . .	18
City . . . . .	21
Event . . . . .	33
std::exception	
InvalidCityException . . . . .	42
Grid . . . . .	35
Intersection . . . . .	39
Node . . . . .	43
Person . . . . .	45
Road . . . . .	51
SimulationClock . . . . .	53
Simulator . . . . .	55
TrafficLight . . . . .	57
Visualization . . . . .	60



## Chapter 3

# Class Index

### 3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

<a href="#">Analysis</a>	Class used for analyzing the traffic data provided by the simulator . . . . .	7
<a href="#">Building</a>	An abstract class for representing the buildings in the simulated city. Each building is either a commercial, a residential, or an industrial building . . . . .	9
<a href="#">Car</a>	A class used for representing a car in the simulated city. The cars move around the city to various destinations during the day . . . . .	12
<a href="#">Cell</a>	A class for representing a single cell in the graphical simulation . . . . .	18
<a href="#">City</a>	A class used for keeping track of the different elements (e.g buildings, roads, intersections, etc.) in the simulated city . . . . .	21
<a href="#">Commercial</a>	One of the concretizing classes for the abstract class <a href="#">Building</a> . . . . .	31
<a href="#">Event</a>	Responsible for creating random schedules that determine the events a person will have over a period of time. It considers the person's type and adjusts the schedule accordingly . . . . .	33
<a href="#">Grid</a>	A class for storing all the cells of the graphical simulation . . . . .	35
<a href="#">Industrial</a>	One of the concretizing classes for the abstract class <a href="#">Building</a> . . . . .	38
<a href="#">Intersection</a>	A class used for representing an intersection in the simulated city . . . . .	39
<a href="#">InvalidCityException</a>	An exception class extending the std::exception class. Used in situations where the JSON file provided by the user is invalid . . . . .	42
<a href="#">Node</a>	A class for representing a node in the simulated city. A node can be either a <a href="#">Building</a> or an <a href="#">Intersection</a> , and two nodes can be connected with a <a href="#">Road</a> object . . . . .	43
<a href="#">Person</a>	A class for persons in the city. Each person has a name, type, working place and home. In addition they will have a car and a schedule, once they are added into the city. Starting location for person and its car is at home . . . . .	45
<a href="#">Residential</a>	One of the concretizing classes for the abstract class <a href="#">Building</a> . . . . .	50

<a href="#">Road</a>	A class for representing a road in the simulated city . . . . .	51
<a href="#">SimulationClock</a>	Class used for time management in the simulation. Includes a clock that runs with regard to system time. One second in real life equals one minute in simulation time . . . . .	53
<a href="#">Simulator</a>	Class that handles the crucial parts of simulation. This class includes the main threads that the user uses to control the simulation. The simulation is initialized/finished in this class . . . . .	55
<a href="#">TrafficLight</a>	A class for representing the traffic lights in the simulated city . . . . .	57
<a href="#">Visualization</a>	Class used for visualizing the traffic simulator in a separate GUI . . . . .	60

## Chapter 4

# Class Documentation

### 4.1 Analysis Class Reference

Class used for analyzing the traffic data provided by the simulator.

```
#include <analysis.hpp>
```

#### Public Member Functions

- [Analysis](#) ([City](#) \*city, [SimulationClock](#) \*clock)  
*Construct a new [Analysis](#) object.*
- void [Analyze](#) ()  
*The main function of this class, which is used to obtain the analysis data.*
- void [SpecifyRoad](#) (int roadIndex)  
*Specifies the road that is going to be analyzed.*
- void [GenerateHourlyHistogram](#) (std::vector< std::vector< int >> data)  
*Prints out a histogram of the current day with hourly statistics of road usage.*
- void [ExportToCSV](#) (const std::string &filename)  
*Exports the analysis data into a CSV file in the root folder of the project.*
- std::vector< std::vector< int > > [GetData](#) ()  
*Returns the hourly road data.*
- void [Init](#) ()  
*Initializes the previousCars\_ private variable.*

#### 4.1.1 Detailed Description

Class used for analyzing the traffic data provided by the simulator.

#### 4.1.2 Constructor & Destructor Documentation

##### 4.1.2.1 Analysis()

```
Analysis::Analysis (
    City * city,
    SimulationClock * clock )
```

Construct a new [Analysis](#) object.

**Parameters**

<i>city</i>	Pointer to the main city object
<i>clock</i>	Pointer to the main clock object

### 4.1.3 Member Function Documentation

#### 4.1.3.1 ExportToCSV()

```
void Analysis::ExportToCSV (
    const std::string & filename )
```

Exports the analysis data into a CSV file in the root folder of the project.

**Parameters**

<i>data</i>	<a href="#">Road</a> usage data that is provided by the GetData function in this class
-------------	--

#### 4.1.3.2 GenerateHourlyHistogram()

```
void Analysis::GenerateHourlyHistogram (
    std::vector< std::vector< int >> data )
```

Prints out a histogram of the current day with hourly statistics of road usage.

**Parameters**

<i>data</i>	<a href="#">Road</a> usage data that is provided by the GetData function in this class
-------------	--

#### 4.1.3.3 GetData()

```
std::vector< std::vector< int > > Analysis::GetData ( )
```

Returns the hourly road data.

**Returns**

The hourly road data



#### 4.1.3.4 SpecifyRoad()

```
void Analysis::SpecifyRoad (
    int roadIndex )
```

Specifies the road that is going to be analyzed.

##### Parameters

<code>roadIndex</code>	Index of the road that is going to be analyzed
------------------------	--

The documentation for this class was generated from the following files:

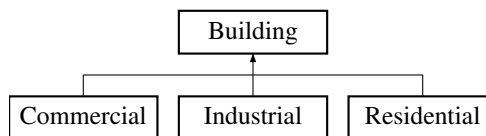
- src/analysis.hpp
- src/analysis.cpp

## 4.2 Building Class Reference

An abstract class for representing the buildings in the simulated city. Each building is either a commercial, a residential, or an industrial building.

```
#include <building.hpp>
```

Inheritance diagram for Building:



### Public Member Functions

- **Building** (const std::string &buildingName, const std::pair< int, int > &location, const std::string &type)  
*Construct a new **Building** object.*
- virtual **~Building** ()  
*Destroy the **Building** object.*
- const std::string & **GetName** () const  
*Get the name of the building.*
- virtual void **Draw** (sf::RenderWindow &window, int cellSize)=0  
*Draws the building to the SFML window given as a parameter. Each subclass defines the draw function themselves.*
- const std::pair< int, int > & **GetLocation** () const  
*Get the location of the building.*
- virtual std::string **GetType** () const =0  
*Get type of building.*

## Protected Attributes

- `std::string` **buildingName\_**
- `std::pair< int, int >` **location\_**
- `int` **personAmount\_**
- `int` **carAmount\_**
- `int` **maxPersonCapacity\_**
- `int` **maxCarCapacity\_**
- `std::string` **type\_**

### 4.2.1 Detailed Description

An abstract class for representing the buildings in the simulated city. Each building is either a commercial, a residential, or an industrial building.

### 4.2.2 Constructor & Destructor Documentation

#### 4.2.2.1 Building()

```
Building::Building (
    const std::string & buildingName,
    const std::pair< int, int > & location,
    const std::string & type )
```

Construct a new [Building](#) object.

#### Parameters

<i>buildingName</i>	The name of the building
<i>location</i>	The coordinates of the building
<i>type</i>	The type of the building ("industrial", "residential", "shop", "gym", or "restaurant")

### 4.2.3 Member Function Documentation

#### 4.2.3.1 Draw()

```
virtual void Building::Draw (
    sf::RenderWindow & window,
    int cellSize ) [pure virtual]
```

Draws the building to the SFML window given as a parameter. Each subclass defines the draw function themselves.

**Parameters**

<i>window</i>	A reference to the SFML window where the building should be drawn
<i>cellSize</i>	The size of the cell to be drawn

Implemented in [Residential](#), [Industrial](#), and [Commercial](#).

**4.2.3.2 GetLocation()**

```
const std::pair< int, int > & Building::GetLocation ( ) const
```

Get the location of the building.

**Returns**

The coordinates of the building as `std::pair<int, int>&`

**4.2.3.3 GetName()**

```
const std::string & Building::GetName ( ) const
```

Get the name of the building.

**Returns**

The name of the building as `std::string&`

**4.2.3.4 GetType()**

```
virtual std::string Building::GetType ( ) const [pure virtual]
```

Get type of building.

**Returns**

The type as a string.

Implemented in [Residential](#), [Industrial](#), and [Commercial](#).

The documentation for this class was generated from the following files:

- `src/building.hpp`
- `src/building.cpp`

## 4.3 Car Class Reference

A class used for representing a car in the simulated city. The cars move around the city to various destinations during the day.

```
#include <car.hpp>
```

### Public Member Functions

- **Car** (**Node** \*startingNode)  
*Constructor. Constructs a new car object.*
- void **Update** (float deltaTime, float currentTime, std::vector< **Node** \* > allNodes, std::vector< **Intersection** \* > intersections, std::vector< **Car** \* > cars, std::vector< **Road** \* > roads)  
*Updates the cars location and state. Checks the cars schedule to find the next destination.*
- **Intersection** \* **GetIntersection** (std::vector< **Intersection** \* > intersections)  
*Used to return the intersection at the vicinity of the car.*
- void **SetDestination** (**Node** \*destination)  
*Set a new destination for the car.*
- void **SetDirection** (std::pair< int, int > current, std::pair< int, int > destination)  
*Set the direction for the car.*
- void **SetSpeedLimit** (std::vector< **Road** \* > roads)  
*Set the speedlimit for the car according to the current road.*
- bool **AtDestination** (float destinationX, float destinationY)  
*Check if the car is at destination.*
- bool **CheckIntersection** (**Intersection** \*intersection, std::vector< **Car** \* > cars)  
*Check that the intersection is ok and car can proceed.*
- bool **LaneIsFree** (**Intersection** \*intersection, std::vector< **Car** \* > cars, std::string nextDirection)  
*Check that the lane that the car will go to after the intersection is free.*
- bool **YieldRight** (**Intersection** \*intersection, std::vector< **Car** \* > cars)  
*If intersection does not have traffic lights cars should yield to other cars coming from right.*
- std::string & **GetDirection** ()  
*Get the direction of the car.*
- void **Draw** (sf::RenderWindow &window, int cellSize)  
*Draw the car into the SFML-window.*
- void **AddEvent** (int time, **Node** \*node)  
*Add an event to the cars schedule.*
- void **InitializeSchedule** (std::map< int, **Node** \* > schedule)  
*Initialize the cars schedule.*
- bool **CarInFront** (std::vector< **Car** \* > cars)  
*Check if there is a car in front of this car.*
- void **SetColor** (PersonType pType)  
*Set the color for the car according to the PersonType.*
- std::pair< float, float > **GetLocation** ()  
*Get the location of the car.*
- std::vector< **Node** \* > **Dijkstra** (**Node** \*source, **Node** \*destination, std::vector< **Node** \* > allNodes)  
*An algorithm to calculate the shortest route from source node to destination node.*

### 4.3.1 Detailed Description

A class used for representing a car in the simulated city. The cars move around the city to various destinations during the day.

## 4.3.2 Constructor & Destructor Documentation

### 4.3.2.1 Car()

```
Car::Car (
    Node * startingNode )
```

Constructor. Constructs a new car object.

#### Parameters

<i>startingNode</i>	The <a href="#">Node</a> where the car starts at is the home of its owner by default.
---------------------	---

## 4.3.3 Member Function Documentation

### 4.3.3.1 AddEvent()

```
void Car::AddEvent (
    int time,
    Node * node )
```

Add an event to the cars schedule.

#### Parameters

<i>time</i>	What time should the car move to the destination
<i>node</i>	The node that the car should move to

### 4.3.3.2 AtDestination()

```
bool Car::AtDestination (
    float destinationX,
    float destinationY )
```

Check if the car is at destination.

#### Parameters

<i>destinationX</i>	The x-coordinate of the destination
<i>destinationY</i>	The y-coordinate of the destination

**Returns**

True if the car is at destination, false otherwise

**4.3.3.3 CarInFront()**

```
bool Car::CarInFront (
    std::vector< Car * > cars )
```

Check if there is a car in front of this car.

**Parameters**

<i>cars</i>	The cars in the city
-------------	----------------------

**4.3.3.4 CheckIntersection()**

```
bool Car::CheckIntersection (
    Intersection * intersection,
    std::vector< Car * > cars )
```

Check that the intersection is ok and car can proceed.

**Parameters**

<i>intersection</i>	The intersection in front of the car
<i>cars</i>	All the cars in the city

**Returns**

True if the car can proceed

**4.3.3.5 Dijkstra()**

```
std::vector< Node * > Car::Dijkstra (
    Node * source,
    Node * destination,
    std::vector< Node * > allNodes )
```

An algorithm to calculate the shortest route from source node to destination node.

## Parameters

<i>source</i>	The node where the car starts
<i>destination</i>	The node where the car should move next
<i>allNodes</i>	All the nodes in the city

**4.3.3.6 Draw()**

```
void Car::Draw (
    sf::RenderWindow & window,
    int cellSize )
```

Draw the car into the SFML-window.

## Parameters

<i>window</i>	The SFML-window
<i>cellSize</i>	The size of the cell in the GUI

**4.3.3.7 GetDirection()**

```
std::string & Car::GetDirection ( )
```

Get the direction of the car.

## Returns

The direction that the car is going to

**4.3.3.8 GetIntersection()**

```
Intersection * Car::GetIntersection (
    std::vector< Intersection * > intersections )
```

Used to return the intersection at the vicinity of the car.

## Parameters

<i>intersections</i>	All the intersections in the city.
----------------------	------------------------------------

**Returns**

[Intersection](#) near the cars location

**4.3.3.9 InitializeSchedule()**

```
void Car::InitializeSchedule (
    std::map< int, Node * > schedule )
```

Initialize the cars schedule.

**Parameters**

<i>schedule</i>	The schedule for the car
-----------------	--------------------------

**4.3.3.10 LaneIsFree()**

```
bool Car::LaneIsFree (
    Intersection * intersection,
    std::vector< Car * > cars,
    std::string nextDirection )
```

Check that the lane that the car will go to after the intersection is free.

**Parameters**

<i>intersection</i>	The intersection in front of the car
<i>cars</i>	The cars in the city
<i>nextDirection</i>	The direction that the car will go to after the intersection

**Returns**

True if the lane is free and false otherwise

**4.3.3.11 SetColor()**

```
void Car::SetColor (
    PersonType pType )
```

Set the color for the car according to the [PersonType](#).



## Parameters

<i>pType</i>	the PersonType of the car's owner
--------------	-----------------------------------

**4.3.3.12 SetDestination()**

```
void Car::SetDestination (
    Node * destination )
```

Set a new destination for the car.

## Parameters

<i>destination</i>	New destination Node.
--------------------	-----------------------

**4.3.3.13 SetDirection()**

```
void Car::SetDirection (
    std::pair< int, int > current,
    std::pair< int, int > destination )
```

Set the direction for the car.

## Parameters

<i>current</i>	Current location of the car.
<i>destination</i>	The location of the destination.

**4.3.3.14 SetSpeedLimit()**

```
void Car::SetSpeedLimit (
    std::vector< Road * > roads )
```

Set the speedlimit for the car according to the current road.

## Parameters

<i>roads</i>	All the roads in the city
--------------	---------------------------

#### 4.3.3.15 Update()

```
void Car::Update (
    float deltaTime,
    float currentTime,
    std::vector< Node * > allNodes,
    std::vector< Intersection * > intersections,
    std::vector< Car * > cars,
    std::vector< Road * > roads )
```

Updates the cars location and state. Checks the cars schedule to find the next destination.

##### Parameters

<i>deltaTime</i>	How much time has passed since the car was last updated.
<i>currentTime</i>	How much time has passed in the simulation in total. Used to check the schedule.
<i>allNodes</i>	All the nodes currently in the city.
<i>intersections</i>	All the intersections currently in the city.
<i>cars</i>	All the cars currently in the city.
<i>roads</i>	All the roads currently in the city.

#### 4.3.3.16 YieldRight()

```
bool Car::YieldRight (
    Intersection * intersection,
    std::vector< Car * > cars )
```

If intersection does not have traffic lights cars should yield to other cars coming from right.

##### Parameters

<i>intersection</i>	The intersection in front of the car
<i>cars</i>	All the cars in the city

##### Returns

True if the car should yield

The documentation for this class was generated from the following files:

- src/car.hpp
- src/car.cpp

## 4.4 Cell Class Reference

A class for representing a single cell in the graphical simulation.

```
#include <cell.hpp>
```

## Public Member Functions

- [Cell](#) (std::pair< int, int > location)  
*Construct a new [Cell](#) object.*
- bool [IsOccupied](#) () const  
*Returns a boolean value indicating if the cell is occupied or not.*
- void [Occupy](#) (std::string type)  
*Occupies the cell with an object of the type which is given as a parameter.*
- void [Clear](#) ()  
*Empties the cell (i.e. sets the value of the type\_ variable to "Empty").*
- std::string [GetType](#) () const  
*Get the type of the cell.*
- int [GetX](#) () const  
*Get the x-coordinate of the cell.*
- int [GetY](#) () const  
*Get the y-coordinate of the cell.*
- void [Draw](#) (sf::RenderWindow &window, int cellSize, int x, int y)  
*Draws the cell to the SFML window given as a parameter.*

### 4.4.1 Detailed Description

A class for representing a single cell in the graphical simulation.

### 4.4.2 Constructor & Destructor Documentation

#### 4.4.2.1 Cell()

```
Cell::Cell (
    std::pair< int, int > location )
```

Construct a new [Cell](#) object.

Parameters

<i>location</i>	the coordinates of the cell
-----------------	-----------------------------

### 4.4.3 Member Function Documentation

#### 4.4.3.1 Draw()

```
void Cell::Draw (
    sf::RenderWindow & window,
```

```
int cellSize,  
int x,  
int y )
```

Draws the cell to the SFML window given as a parameter.

#### Parameters

<i>window</i>	A reference to the SFML window where the cell should be drawn
<i>cellSize</i>	The size of the cell to be drawn
<i>x</i>	The x-coordinate of the cell to be drawn
<i>y</i>	The y-coordinate of the cell to be drawn

#### 4.4.3.2 GetType()

```
std::string Cell::GetType ( ) const
```

Get the type of the cell.

#### Returns

The type of the cell as std::string

#### 4.4.3.3 GetX()

```
int Cell::GetX ( ) const
```

Get the x-coordinate of the cell.

#### Returns

The x-coordinate as an integer

#### 4.4.3.4 GetY()

```
int Cell::GetY ( ) const
```

Get the y-coordinate of the cell.

#### Returns

The y-coordinate as an integer

#### 4.4.3.5 IsOccupied()

```
bool Cell::IsOccupied ( ) const
```

Returns a boolean value indicating if the cell is occupied or not.

##### Returns

True if the cell is occupied, false otherwise

#### 4.4.3.6 Occupy()

```
void Cell::Occupy (
    std::string type )
```

Occupies the cell with an object of the type which is given as a parameter.

##### Parameters

<i>type</i>	The type of the object being placed in the cell ("Horizontal road", "Vertical road", "Intersection", "industrial", "residential", "shop", "gym", or "restaurant")
-------------	---

The documentation for this class was generated from the following files:

- src/cell.hpp
- src/cell.cpp

## 4.5 City Class Reference

A class used for keeping track of the different elements (e.g buildings, roads, intersections, etc.) in the simulated city.

```
#include <city.hpp>
```

### Public Member Functions

- [City](#) (int sizeX, int sizeY)  
*Construct a new [City](#) object.*
- [~City](#) ()  
*Destroy the [City](#) object.*
- bool [IsValidRoad](#) (std::pair< int, int > start, std::pair< int, int > end) const  
*Checks if the road with the given start and end coordinates is a valid road. Requirements:*
- void [AddRoad](#) (std::pair< int, int > start, std::pair< int, int > end)  
*Adds a road to the city. Throws an [InvalidCityException](#) if the road is invalid.*
- bool [IsValidBuilding](#) ([Building](#) \*b) const

- Checks if the building that the pointer given as parameter is pointing to is valid. The building must be inside city boundaries and the location of the building must be unoccupied.*
- void [AddBuilding](#) (std::string name, std::pair< int, int > location, const std::string &buildingType)  
*Adds a building to the city if it is valid. Otherwise throws an [InvalidCityException](#).*
  - [Node](#) \* [GetNode](#) (std::pair< int, int > location) const  
*Get a pointer to the [Node](#) object in the given location.*
  - std::vector< [Node](#) \* > [GetBuildingNodes](#) () const  
*Get a list of the [Node](#) pointers that have type "Building".*
  - void [AddPersonAndCar](#) (std::string &name, PersonType personType, std::string &workplacename, std::string &homename)  
*Adds a person and its car to the city.*
  - bool [IsValidPerson](#) ([Person](#) \*p)  
*Checks if the person that is trying to be added is valid.*
  - bool [IsValidIntersection](#) ([Intersection](#) \*i) const  
*Checks if the intersection that the pointer given as parameter is pointing to is valid. The intersection must be inside city boundaries and the location of the intersection must be unoccupied.*
  - void [AddIntersection](#) (std::pair< int, int > location)  
*Attempts to create an intersection to the given location. Returns an [InvalidCityException](#) if the location is occupied already.*
  - [Intersection](#) \* [GetIntersection](#) (std::pair< int, int > location) const  
*Get the a pointer to the [Intersection](#) object in the given location.*
  - void [UpdateIntersections](#) (float deltaTime) const  
*Updates the status of all the traffic lights in the city.*
  - void [DrawIntersections](#) (sf::RenderWindow &window) const  
*Draws the intersections in the city to the SFML window given as parameter.*
  - void [ChooseRoad](#) (int roadIndex)  
*Chooses the road that the user wants to analyze and highlights it.*
  - void [AddTrafficLight](#) (std::pair< int, int > location, int redAndGreenDuration, int yellowDuration)  
*Adds a traffic light to the city.*
  - void [UpdateCars](#) (float deltaTime, float currentTime)  
*Updates the locations and destinations of all the cars in the city.*
  - void [DrawCars](#) (sf::RenderWindow &window) const  
*Draws the cars in the city to the SFML window given as parameter.*
  - void [DrawBuildings](#) (sf::RenderWindow &window) const  
*Draws the buildings in the city to the SFML window given as parameter.*
  - void [PrintCity](#) (sf::RenderWindow &window) const  
*Draws the city to the SFML window given as parameter.*
  - [Grid](#) \* [GetGrid](#) () const  
*Get a pointer to the [Grid](#) object of the city.*
  - std::vector< [Road](#) \* > [GetRoads](#) () const  
*Get a vector containing pointers to all the roads in the city.*
  - std::vector< [Car](#) \* > [GetCars](#) () const  
*Get a vector containing pointers to all the cars in the city.*
  - int [TimeUntilNextEvent](#) ([Person](#) \*p) const  
*Get time until next event.*
  - void [AddClock](#) ([SimulationClock](#) \*clock)  
*Add the simulation clock to city.*
  - bool [IsBusy](#) ([Person](#) \*p) const  
*Check if person has an event going on.*
  - void [AddEvent](#) ([Person](#) \*p)  
*Add event to schedule.*

### 4.5.1 Detailed Description

A class used for keeping track of the different elements (e.g buildings, roads, intersections, etc.) in the simulated city.

### 4.5.2 Constructor & Destructor Documentation

#### 4.5.2.1 City()

```
City::City (
    int sizeX,
    int sizeY )
```

Construct a new [City](#) object.

##### Parameters

<i>sizeX</i>	The vertical size of the city
<i>sizeY</i>	The horizontal size of the city

### 4.5.3 Member Function Documentation

#### 4.5.3.1 AddBuilding()

```
void City::AddBuilding (
    std::string name,
    std::pair< int, int > location,
    const std::string & buildingType )
```

Adds a building to the city if it is valid. Otherwise throws an [InvalidCityException](#).

##### Parameters

<i>name</i>	The name of the building
<i>location</i>	The coordinates of the building
<i>buildingType</i>	The type of the building (should be: industrial, residential, shop, gym, or restaurant)

#### 4.5.3.2 AddClock()

```
void City::AddClock (
```

```
SimulationClock * clock )
```

Add the simulation clock to city.

#### Parameters

<i>clock</i>	Clock implemented in <a href="#">SimulationClock</a> , used in simulator.cpp.
--------------	---

#### 4.5.3.3 AddEvent()

```
void City::AddEvent (
    Person * p )
```

Add event to schedule.

#### Parameters

<i>p</i>	This is the person that is going to have new event.
----------	---

#### 4.5.3.4 AddIntersection()

```
void City::AddIntersection (
    std::pair< int, int > location )
```

Attempts to create an intersection to the given location. Returns an [InvalidCityException](#) if the location is occupied already.

#### Parameters

<i>location</i>	The coordinates of the intersection
-----------------	-------------------------------------

#### 4.5.3.5 AddPersonAndCar()

```
void City::AddPersonAndCar (
    std::string & name,
    PersonType personType,
    std::string & workplacename,
    std::string & homename )
```

Adds a person and its car to the city.



## Parameters

<i>name</i>	Name of the person.
<i>personType</i>	Type of the person.
<i>workplacename</i>	Name of working place.
<i>homename</i>	Name of home.

#### 4.5.3.6 AddRoad()

```
void City::AddRoad (
    std::pair< int, int > start,
    std::pair< int, int > end )
```

Adds a road to the city. Throws an [InvalidCityException](#) if the road is invalid.

## Parameters

<i>start</i>	The start coordinates of the road
<i>end</i>	The end coordinates of the road

#### 4.5.3.7 AddTrafficLight()

```
void City::AddTrafficLight (
    std::pair< int, int > location,
    int redAndGreenDuration,
    int yellowDuration )
```

Adds a traffic light to the city.

## Parameters

<i>t</i>	A pointer to the traffic light object to be added
----------	---

#### 4.5.3.8 ChooseRoad()

```
void City::ChooseRoad (
    int roadIndex )
```

Chooses the road that the user wants to analyze and highlights it.

**Parameters**

<i>roadIndex</i>	The index of the chosen road
------------------	------------------------------

**4.5.3.9 DrawBuildings()**

```
void City::DrawBuildings (
    sf::RenderWindow & window ) const
```

Draws the buildings in the city to the SFML window given as parameter.

**Parameters**

<i>window</i>	A reference to an SFML window
---------------	-------------------------------

**4.5.3.10 DrawCars()**

```
void City::DrawCars (
    sf::RenderWindow & window ) const
```

Draws the cars in the city to the SFML window given as parameter.

**Parameters**

<i>window</i>	A reference to an SFML window
---------------	-------------------------------

**4.5.3.11 DrawIntersections()**

```
void City::DrawIntersections (
    sf::RenderWindow & window ) const
```

Draws the intersections in the city to the SFML window given as parameter.

**Parameters**

<i>window</i>	A reference to a SFML window
---------------	------------------------------

#### 4.5.3.12 GetBuildingNodes()

```
std::vector< Node * > City::GetBuildingNodes ( ) const
```

Get a list of the [Node](#) pointers that have type "Building".

##### Returns

A list of pointers to Nodes with type "Building".

#### 4.5.3.13 GetCars()

```
std::vector< Car * > City::GetCars ( ) const
```

Get a vector containing pointers to all the cars in the city.

##### Returns

```
std::vector<Car*>
```

#### 4.5.3.14 GetGrid()

```
Grid * City::GetGrid ( ) const
```

Get a pointer to the [Grid](#) object of the city.

##### Returns

```
Grid*
```

#### 4.5.3.15 GetIntersection()

```
Intersection * City::GetIntersection (
    std::pair< int, int > location ) const
```

Get the a pointer to the [Intersection](#) object in the given location.

##### Parameters

<i>location</i>	The location of the intersection
-----------------	----------------------------------

**Returns**

A pointer to an [Intersection](#) object if there is an [Intersection](#) in the given location, or a nullptr if there is no [Intersection](#) in the given location.

**4.5.3.16 GetNode()**

```
Node * City::GetNode (
    std::pair< int, int > location ) const
```

Get a pointer to the [Node](#) object in the given location.

**Parameters**

<i>location</i>	The location of the node
-----------------	--------------------------

**Returns**

A pointer to a [Node](#) object if there is a [Node](#) in the given location or a nullptr otherwise

**4.5.3.17 GetRoads()**

```
std::vector< Road * > City::GetRoads ( ) const
```

Get a vector containing pointers to all the roads in the city.

**Returns**

```
std::vector<Road*>
```

**4.5.3.18 IsBusy()**

```
bool City::IsBusy (
    Person * p ) const
```

Check if person has an event going on.

**Parameters**

<i>p</i>	Check if this exact person is busy.
----------	-------------------------------------

**Returns**

Boolean value if a person is busy at the moment.

**4.5.3.19 IsValidBuilding()**

```
bool City::IsValidBuilding (
    Building * b ) const
```

Checks if the building that the pointer given as parameter is pointing to is valid. The building must be inside city boundaries and the location of the building must be unoccupied.

**Parameters**

<i>b</i>	A pointer to a building object
----------	--------------------------------

**Returns**

true if the building is valid, false otherwise

**4.5.3.20 IsValidIntersection()**

```
bool City::IsValidIntersection (
    Intersection * i ) const
```

Checks if the intersection that the pointer given as parameter is pointing to is valid. The intersection must be inside city boundaries and the location of the intersection must be unoccupied.

**Parameters**

<i>i</i>	A pointer to a intersection object
----------	------------------------------------

**Returns**

true if the intersection is valid, false otherwise

**4.5.3.21 IsValidPerson()**

```
bool City::IsValidPerson (
    Person * p )
```

Checks if the person that is trying to be added is valid.

## Parameters

<i>p</i>	A pointer to the <a href="#">Person</a> object
----------	--

**4.5.3.22 IsValidRoad()**

```
bool City::IsValidRoad (
    std::pair< int, int > start,
    std::pair< int, int > end ) const
```

Checks if the road with the given start and end coordinates is a valid road. Requirements:

- the road must be inside the boundaries of the city
- the length of the road must be more than zero
- the road can only be horizontal or vertical (not diagonal)
- there cannot be any other roads or buildings between the start and end coordinates of the road

## Parameters

<i>start</i>	The start coordinates of the road
<i>end</i>	The end coordinates of the road

## Returns

true if the road is valid, false otherwise

**4.5.3.23 PrintCity()**

```
void City::PrintCity (
    sf::RenderWindow & window ) const
```

Draws the city to the SFML window given as parameter.

## Parameters

<i>window</i>	A reference to an SFML window
---------------	-------------------------------

**4.5.3.24 TimeUntilNextEvent()**

```
int City::TimeUntilNextEvent (
```

```
Person * p ) const
```

Get time until next event.

#### Parameters

<i>p</i>	Next event from a specified person.
----------	-------------------------------------

#### 4.5.3.25 UpdateCars()

```
void City::UpdateCars (
    float deltaTime,
    float currentTime )
```

Updates the locations and destinations of all the cars in the city.

#### Parameters

<i>deltaTime</i>	
<i>currentTime</i>	

#### 4.5.3.26 UpdateIntersections()

```
void City::UpdateIntersections (
    float deltaTime ) const
```

Updates the status of all the traffic lights in the city.

#### Parameters

<i>deltaTime</i>	
------------------	--

The documentation for this class was generated from the following files:

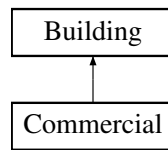
- src/city.hpp
- src/city.cpp

## 4.6 Commercial Class Reference

One of the concretizing classes for the abstract class [Building](#).

```
#include <commercial.hpp>
```

Inheritance diagram for Commercial:



## Public Member Functions

- [Commercial](#) (const std::string &buildingName, const std::pair< int, int > &location, const std::string &commercialType)  
*Construct a new [Commercial](#) object.*
- virtual void [Draw](#) (sf::RenderWindow &window, int cellSize) override  
*Draws the building to the SFML window given as a parameter. Each subclass defines the draw function themselves.*
- std::string [GetType](#) () const override  
*Prints information about the commercial building to the standard output.*

## Additional Inherited Members

### 4.6.1 Detailed Description

One of the concretizing classes for the abstract class [Building](#).

### 4.6.2 Constructor & Destructor Documentation

#### 4.6.2.1 Commercial()

```

Commercial::Commercial (
    const std::string & buildingName,
    const std::pair< int, int > & location,
    const std::string & commercialType )

```

Construct a new [Commercial](#) object.

#### Parameters

<i>buildingName</i>	The name of the commercial building
<i>location</i>	The coordinates of the commercial building
<i>commercialType</i>	The type of the commercial building ("gym", "restaurant", or "shop")



### 4.6.3 Member Function Documentation

#### 4.6.3.1 Draw()

```
void Commercial::Draw (
    sf::RenderWindow & window,
    int cellSize ) [override], [virtual]
```

Draws the building to the SFML window given as a parameter. Each subclass defines the draw function themselves.

##### Parameters

<i>window</i>	A reference to the SFML window where the building should be drawn
<i>cellSize</i>	The size of the cell to be drawn

Implements [Building](#).

The documentation for this class was generated from the following files:

- src/commercial.hpp
- src/commercial.cpp

## 4.7 Event Class Reference

The [Event](#) class is responsible for creating random schedules that determine the events a person will have over a period of time. It considers the person's type and adjusts the schedule accordingly.

```
#include <event.hpp>
```

### Public Member Functions

- [Event](#) ([Person](#) \*person, std::vector< [Node](#) \* > buildingNodes)  
*Construct a new [Event](#) object.*
- std::map< int, [Node](#) \* > [CreateSchedule](#) ()  
*Create a new random schedule for a person.*

#### 4.7.1 Detailed Description

The [Event](#) class is responsible for creating random schedules that determine the events a person will have over a period of time. It considers the person's type and adjusts the schedule accordingly.

#### 4.7.2 Constructor & Destructor Documentation

#### 4.7.2.1 Event()

```
Event::Event (
    Person * person,
    std::vector< Node * > buildingNodes )
```

Construct a new [Event](#) object.

## Parameters

<i>person</i>	The person whose schedule is created.
<i>buildingNodes</i>	The nodes that contain a building.

### 4.7.3 Member Function Documentation

#### 4.7.3.1 CreateSchedule()

```
std::map< int, Node * > Event::CreateSchedule ( )
```

Create a new random schedule for a person.

## Returns

Returns the schedule in map format, where a time value is mapped to a corresponding event node.

The documentation for this class was generated from the following files:

- src/event.hpp
- src/event.cpp

## 4.8 Grid Class Reference

A class for storing all the cells of the graphical simulation.

```
#include <grid.hpp>
```

### Public Member Functions

- [Grid](#) (int xSize, int ySize)  
*Construct a new [Grid](#) object.*
- [~Grid](#) ()  
*Destroy the [Grid](#) object.*
- [Cell \\* GetCell](#) (int x, int y)  
*Get a pointer to the cell at the given location.*
- [std::vector< Cell \\* > GetNeighborCells](#) (int x, int y)  
*Get the neighbouring cells of the cell in the given location.*
- int [GetSizeX](#) () const  
*Get the vertical size of the grid.*
- int [GetSizeY](#) () const  
*Get the horizontal size of the grid.*

### 4.8.1 Detailed Description

A class for storing all the cells of the graphical simulation.

### 4.8.2 Constructor & Destructor Documentation

#### 4.8.2.1 Grid()

```
Grid::Grid (
    int xSize,
    int ySize )
```

Construct a new [Grid](#) object.

##### Parameters

<i>xSize</i>	The vertical size of the grid
<i>ySize</i>	The horizontal size of the grid

### 4.8.3 Member Function Documentation

#### 4.8.3.1 GetCell()

```
Cell * Grid::GetCell (
    int x,
    int y )
```

Get a pointer to the cell at the given location.

##### Parameters

<i>x</i>	The x-coordinate of the cell
<i>y</i>	The y-coordinate of the cell

##### Returns

A pointer to a [Cell](#) object if there is one in the given location, a nullptr otherwise

#### 4.8.3.2 GetNeighborCells()

```
std::vector< Cell * > Grid::GetNeighborCells (
    int x,
    int y )
```

Get the neighbouring cells of the cell in the given location.

##### Parameters

<i>x</i>	The x-coordinate of the cell
<i>y</i>	The y-coordinate of the cell

##### Returns

A vector containing pointers to all the neighbouring cells

#### 4.8.3.3 GetSizeX()

```
int Grid::GetSizeX ( ) const
```

Get the vertical size of the grid.

##### Returns

The vertical size of the grid as an integer

#### 4.8.3.4 GetSizeY()

```
int Grid::GetSizeY ( ) const
```

Get the horizontal size of the grid.

##### Returns

The horizontal size of the grid as an integer

The documentation for this class was generated from the following files:

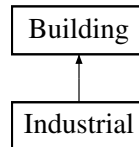
- src/grid.hpp
- src/grid.cpp

## 4.9 Industrial Class Reference

One of the concretizing classes for the abstract class [Building](#).

```
#include <industrial.hpp>
```

Inheritance diagram for Industrial:



### Public Member Functions

- [Industrial](#) (const std::string &buildingName, const std::pair< int, int > &location)  
*Construct a new [Industrial](#) object.*
- virtual void [Draw](#) (sf::RenderWindow &window, int cellSize) override  
*Draws the building to the SFML window given as a parameter. Each subclass defines the draw function themselves.*
- std::string [GetType](#) () const override  
*Prints information about the industrial building to the standard output.*

### Additional Inherited Members

#### 4.9.1 Detailed Description

One of the concretizing classes for the abstract class [Building](#).

#### 4.9.2 Constructor & Destructor Documentation

##### 4.9.2.1 Industrial()

```
Industrial::Industrial (
    const std::string & buildingName,
    const std::pair< int, int > & location ) [inline]
```

Construct a new [Industrial](#) object.

##### Parameters

<i>buildingName</i>	The name of the industrial building
<i>location</i>	The coordinates of the idnustrial building

### 4.9.3 Member Function Documentation

#### 4.9.3.1 Draw()

```
virtual void Industrial::Draw (
    sf::RenderWindow & window,
    int cellSize ) [inline], [override], [virtual]
```

Draws the building to the SFML window given as a parameter. Each subclass defines the draw function themselves.

#### Parameters

<i>window</i>	A reference to the SFML window where the building should be drawn
<i>cellSize</i>	The size of the cell to be drawn

Implements [Building](#).

The documentation for this class was generated from the following file:

- src/industrial.hpp

## 4.10 Intersection Class Reference

A class used for representing an intersection in the simulated city.

```
#include <intersection.hpp>
```

### Public Member Functions

- [Intersection](#) (std::pair< int, int > location)  
*Construct a new [Intersection](#) object.*
- std::pair< int, int > [GetLocation](#) () const  
*Get the location of the intersection.*
- void [Update](#) (float deltaTime)  
*Attempts to update the status of the traffic lights in the intersection, if there are any.*
- void [AddTrafficLight](#) ([TrafficLight](#) \*t)  
*Adds a traffic light to the intersection.*
- bool [HasTrafficLight](#) () const  
*Check if the intersection has a traffic light.*
- bool [AllowVertical](#) ()  
*Returns a boolean value indicating if cars are currently allowed to move vertically through the intersection.*
- bool [AllowHorizontal](#) ()  
*Returns a boolean value indicating if cars are currently allowed to move horizontally through the intersection.*
- void [Draw](#) (sf::RenderWindow &window, int cellSize, [Grid](#) \*grid)  
*Draws the intersection into the SFML window given as parameter.*

### 4.10.1 Detailed Description

A class used for representing an intersection in the simulated city.

### 4.10.2 Constructor & Destructor Documentation

#### 4.10.2.1 Intersection()

```
Intersection::Intersection (
    std::pair< int, int > location )
```

Construct a new [Intersection](#) object.

##### Parameters

<i>location</i>	the coordinates of the intersection as <code>std::pair&lt;int, int&gt;</code>
-----------------	---

### 4.10.3 Member Function Documentation

#### 4.10.3.1 AddTrafficLight()

```
void Intersection::AddTrafficLight (
    TrafficLight * t )
```

Adds a traffic light to the intersection.

##### Parameters

<i>t</i>	A pointer to a <a href="#">TrafficLight</a> object
----------	--

#### 4.10.3.2 AllowHorizontal()

```
bool Intersection::AllowHorizontal ( )
```

Returns a boolean value indicating if cars are currently allowed to move horizontally through the intersection.

##### Returns

bool



#### 4.10.3.3 AllowVertical()

```
bool Intersection::AllowVertical ( )
```

Returns a boolean value indicating if cars are currently allowed to move vertically through the intersection.

##### Returns

bool

#### 4.10.3.4 Draw()

```
void Intersection::Draw (
    sf::RenderWindow & window,
    int cellSize,
    Grid * grid )
```

Draws the intersection into the SFML window given as parameter.

##### Parameters

<i>window</i>	The SFML window to draw the intersection to
<i>cellSize</i>	The size of a single cell in the window

#### 4.10.3.5 GetLocation()

```
std::pair< int, int > Intersection::GetLocation ( ) const
```

Get the location of the intersection.

##### Returns

The coordinates of the intersection as `std::pair<int, int>`

#### 4.10.3.6 Update()

```
void Intersection::Update (
    float deltaTime )
```

Attempts to update the status of the traffic lights in the intersection, if there are any.

#### Parameters

<code>deltaTime</code>	
------------------------	--

The documentation for this class was generated from the following files:

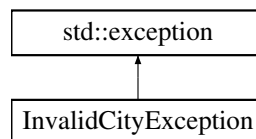
- `src/intersection.hpp`
- `src/intersection.cpp`

## 4.11 InvalidCityException Class Reference

An exception class extending the `std::exception` class. Used in situations where the JSON file provided by the user is invalid.

```
#include <invalidcityexception.hpp>
```

Inheritance diagram for `InvalidCityException`:



### Public Member Functions

- [InvalidCityException](#) (`std::string errorMessage`)  
*Construct a new [InvalidCityException](#) object.*
- `std::string GetError () const`  
*Return the error message describing what went wrong.*

#### 4.11.1 Detailed Description

An exception class extending the `std::exception` class. Used in situations where the JSON file provided by the user is invalid.

#### 4.11.2 Constructor & Destructor Documentation

##### 4.11.2.1 InvalidCityException()

```
InvalidCityException::InvalidCityException (  
    std::string errorMessage ) [inline]
```

Construct a new [InvalidCityException](#) object.

## Parameters

<code>errorMessage</code>	
---------------------------	--

### 4.11.3 Member Function Documentation

#### 4.11.3.1 GetError()

```
std::string InvalidCityException::GetError ( ) const [inline]
```

Return the error message describing what went wrong.

## Returns

std::string

The documentation for this class was generated from the following file:

- src/invalidcityexception.hpp

## 4.12 Node Class Reference

A class for representing a node in the simulated city. A node can be either a [Building](#) or an [Intersection](#), and two nodes can be connected with a [Road](#) object.

```
#include <node.hpp>
```

### Public Member Functions

- [Node](#) (NodeType type, const std::pair< int, int > location)  
*Construct a new [Node](#) object.*
- NodeType [GetType](#) () const  
*Returns the type of the node.*
- std::vector< std::pair< [Node](#) \*, int > > [GetConnections](#) () const  
*Get the nodes that this node is connected to.*
- void [AddConnection](#) ([Node](#) \*node, int weight)  
*Connects this node with the node that is given as parameter.*
- std::pair< int, int > [GetLocation](#) () const  
*Get the location of the node.*

#### 4.12.1 Detailed Description

A class for representing a node in the simulated city. A node can be either a [Building](#) or an [Intersection](#), and two nodes can be connected with a [Road](#) object.

## 4.12.2 Constructor & Destructor Documentation

### 4.12.2.1 Node()

```
Node::Node (
    NodeType type,
    const std::pair< int, int > location )
```

Construct a new [Node](#) object.

#### Parameters

<i>type</i>	The type of the node as NodeType ( <a href="#">Building</a> or <a href="#">Intersection</a> )
<i>location</i>	The coordinates of the node as std::pair<int, int>

## 4.12.3 Member Function Documentation

### 4.12.3.1 AddConnection()

```
void Node::AddConnection (
    Node * node,
    int weight )
```

Connects this node with the node that is given as parameter.

#### Parameters

<i>node</i>	The node that the connection will be made with
<i>weight</i>	The weight of the node that the connection is made with

### 4.12.3.2 GetConnections()

```
std::vector< std::pair< Node *, int > > Node::GetConnections ( ) const
```

Get the nodes that this node is connected to.

#### Returns

```
std::vector<std::pair<Node*, int>>
```

#### 4.12.3.3 GetLocation()

```
std::pair< int, int > Node::GetLocation ( ) const
```

Get the location of the node.

##### Returns

The coordinates of the node as `std::pair<int, int>`

#### 4.12.3.4 GetType()

```
NodeType Node::GetType ( ) const
```

Returns the type of the node.

##### Returns

A `NodeType` object ([Building](#) or [Intersection](#))

The documentation for this class was generated from the following files:

- `src/node.hpp`
- `src/node.cpp`

## 4.13 Person Class Reference

A class for persons in the city. Each person has a name, type, working place and home. In addition they will have a car and a schedule, once they are added into the city. Starting location for person and its car is at home.

```
#include <person.hpp>
```

### Public Member Functions

- [Person](#) (const std::string &name, PersonType personType, [Industrial](#) \*workplace, [Residential](#) \*home)  
*Construct a new [Person](#) object.*
- [Industrial](#) \* [GetWorkplace](#) () const  
*Get the person's working place.*
- [Residential](#) \* [GetResidence](#) () const  
*Get the person's home.*
- [Car](#) \* [GetCar](#) ()  
*Get the person's car.*
- std::string [GetName](#) () const  
*Get the person's name.*
- void [BuyCar](#) ([Car](#) \*car)  
*Buy a car for person. This is done when the person is added to city, so the person can travel.*

- void [InitializeSchedule](#) (std::map< int, [Node](#) \* > schedule)  
*Initialize the schedule for the person and its car. This is done after person is added to a city and it has "bought" a car.*
- bool [isAtHome](#) () const  
*Gives boolean value that tells, is the person at home (true), or not (false).*
- std::pair< int, int > [GetLocation](#) () const  
*Gives the location of the person.*
- PersonType [GetPersonType](#) () const  
*Gives type of the person.*
- std::map< int, [Node](#) \* > [GetSchedule](#) () const  
*Gives the schedule for the person.*
- void [AddEvent](#) (int time, [Node](#) \*node)  
*Adds a new event to the schedule.*
- void [UpdateLocationFromCar](#) (std::pair< float, float > location)  
*When car travels, person travels in the car. This makes it possible.*

### 4.13.1 Detailed Description

A class for persons in the city. Each person has a name, type, working place and home. In addition they will have a car and a schedule, once they are added into the city. Starting location for person and its car is at home.

### 4.13.2 Constructor & Destructor Documentation

#### 4.13.2.1 Person()

```
Person::Person (
    const std::string & name,
    PersonType personType,
    Industrial * workplace,
    Residential * home )
```

Construct a new [Person](#) object.

#### Parameters

<i>name</i>	The name of the person
<i>personType</i>	The type of person (Neutral, Angry, Lazy, Gentleman, Active and Nocturnal)
<i>workplace</i>	Pointer to a <a href="#">Industrial</a> object. Persons workplace
<i>home</i>	Pointer to a <a href="#">Residential</a> object. Persons home.

### 4.13.3 Member Function Documentation

#### 4.13.3.1 AddEvent()

```
void Person::AddEvent (
    int time,
    Node * node )
```

Adds a new event to the schedule.

##### Parameters

<i>time</i>	Time in minutes when this event occurs.
<i>node</i>	Pointer to the destination <a href="#">Node</a> object.

#### 4.13.3.2 BuyCar()

```
void Person::BuyCar (
    Car * car )
```

Buy a car for person. This is done when the person is added to city, so the person can travel.

##### Parameters

<i>car</i>	Pointer to a <a href="#">Car</a> object.
------------	--

#### 4.13.3.3 GetCar()

```
Car * Person::GetCar ( )
```

Get the person's car.

##### Returns

Pointer to a [Car](#) object.

#### 4.13.3.4 GetLocation()

```
std::pair< int, int > Person::GetLocation ( ) const
```

Gives the location of the person.

##### Returns

Coordinates of the location as a pair of ints.

#### 4.13.3.5 GetName()

```
std::string Person::GetName ( ) const
```

Get the person's name.

##### Returns

Name as string.

#### 4.13.3.6 GetPersonType()

```
PersonType Person::GetPersonType ( ) const
```

Gives type of the person.

##### Returns

The type of the person as PersonType, that is implemented in [car.hpp](#)

#### 4.13.3.7 GetResidence()

```
Residential * Person::GetResidence ( ) const
```

Get the person's home.

##### Returns

Pointer to a [Residential](#) object.

#### 4.13.3.8 GetSchedule()

```
std::map< int, Node * > Person::GetSchedule ( ) const
```

Gives the schedule for the person.

##### Returns

Schedule which is a map that contains minutes as keys, and pointers to [Node](#) objects as events.



#### 4.13.3.9 GetWorkplace()

```
Industrial * Person::GetWorkplace ( ) const
```

Get the person's working place.

##### Returns

Pointer to an [Industrial](#) object.

#### 4.13.3.10 InitializeSchedule()

```
void Person::InitializeSchedule (
    std::map< int, Node * > schedule )
```

Initialize the schedule for the person and its car. This is done after person is added to a city and it has "bought" a car.

##### Parameters

<i>schedule</i>	The random schedule that is given for a person.
-----------------	---

#### 4.13.3.11 isAtHome()

```
bool Person::isAtHome ( ) const
```

Gives boolean value that tells, is the person at home (true), or not (false).

##### Returns

Boolean value, false or true.

#### 4.13.3.12 UpdateLocationFromCar()

```
void Person::UpdateLocationFromCar (
    std::pair< float, float > location )
```

When car travels, person travels in the car. This makes it possible.

##### Parameters

<i>location</i>	The location of the car.
-----------------	--------------------------

The documentation for this class was generated from the following files:

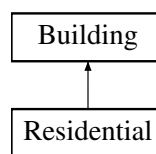
- src/person.hpp
- src/person.cpp

## 4.14 Residential Class Reference

One of the concretizing classes for the abstract class [Building](#).

```
#include <residential.hpp>
```

Inheritance diagram for Residential:



### Public Member Functions

- [Residential](#) (const std::string &buildingName, const std::pair< int, int > &location)  
*Construct a new [Residential](#) object.*
- virtual void [Draw](#) (sf::RenderWindow &window, int cellSize) override  
*Draws the building to the SFML window given as a parameter. Each subclass defines the draw function themselves.*
- std::string [GetType](#) () const override  
*Prints information about the residential building to the standard output.*

### Additional Inherited Members

#### 4.14.1 Detailed Description

One of the concretizing classes for the abstract class [Building](#).

#### 4.14.2 Constructor & Destructor Documentation

##### 4.14.2.1 Residential()

```
Residential::Residential (
    const std::string & buildingName,
    const std::pair< int, int > & location ) [inline]
```

Construct a new [Residential](#) object.

## Parameters

<i>buildingName</i>	The name of the residential building
<i>location</i>	The coordinates of the residential building

### 4.14.3 Member Function Documentation

#### 4.14.3.1 Draw()

```
virtual void Residential::Draw (
    sf::RenderWindow & window,
    int cellSize ) [inline], [override], [virtual]
```

Draws the building to the SFML window given as a parameter. Each subclass defines the draw function themselves.

## Parameters

<i>window</i>	A reference to the SFML window where the building should be drawn
<i>cellSize</i>	The size of the cell to be drawn

Implements [Building](#).

The documentation for this class was generated from the following file:

- src/residential.hpp

## 4.15 Road Class Reference

A class for representing a road in the simulated city.

```
#include <road.hpp>
```

### Public Member Functions

- [Road](#) (std::pair< int, int > start, std::pair< int, int > end, int speedLimit)  
*Construct a new [Road](#) object.*
- std::pair< int, int > [GetEnd](#) () const  
*Get the coordinates of the end of the road.*
- std::pair< int, int > [GetStart](#) () const  
*Get the coordinates of the start of the road.*
- int [GetSpeedLimit](#) () const  
*Get the speed limit of the road.*
- bool [IsVertical](#) () const  
*Returns a boolean value indicating if the road os vertical or not.*
- bool [IsHorizontal](#) () const  
*Returns a boolean value indicating if the road os horizontal or not.*

### 4.15.1 Detailed Description

A class for representing a road in the simulated city.

### 4.15.2 Constructor & Destructor Documentation

#### 4.15.2.1 Road()

```
Road::Road (
    std::pair< int, int > start,
    std::pair< int, int > end,
    int speedLimit )
```

Construct a new [Road](#) object.

##### Parameters

<i>start</i>	The starting coordinates of the road
<i>end</i>	The ending coordinates of the road
<i>speedLimit</i>	The speed limit on the road

### 4.15.3 Member Function Documentation

#### 4.15.3.1 GetEnd()

```
std::pair< int, int > Road::GetEnd ( ) const
```

Get the coordinates of the end of the road.

##### Returns

`std::pair<int, int>`

#### 4.15.3.2 GetSpeedLimit()

```
int Road::GetSpeedLimit ( ) const
```

Get the speed limit of the road.

##### Returns

The speed limit as an integer

#### 4.15.3.3 GetStart()

```
std::pair< int, int > Road::GetStart ( ) const
```

Get the coordinates of the start of the road.

##### Returns

`std::pair<int, int>`

#### 4.15.3.4 IsHorizontal()

```
bool Road::IsHorizontal ( ) const
```

Returns a boolean value indicating if the road os horizontal or not.

##### Returns

True if the road is horizontal, false otherwise

#### 4.15.3.5 IsVertical()

```
bool Road::IsVertical ( ) const
```

Returns a boolean value indicating if the road os vertical or not.

##### Returns

True if the road is vertical, false otherwise

The documentation for this class was generated from the following files:

- `src/road.hpp`
- `src/road.cpp`

## 4.16 SimulationClock Class Reference

Class used for time management in the simulation. Includes a clock that runs with regard to system time. One second in real life equals one minute in simulation time.

```
#include <simulationClock.hpp>
```

## Public Member Functions

- [SimulationClock](#) ()  
*Construct a new Simulation Clock object.*
- void [Start](#) ()  
*Function that starts the clock.*
- double [GetElapsedTime](#) ()  
*Returns the elapsed time in simulation.*
- int [GetDayNumber](#) ()  
*Returns the day number of the simulation.*
- std::string [GetSimulationTime](#) ()  
*Returns the simulation's time (1 second in real life = 1 minute in simulation time)*
- void [SetSimulationSpeed](#) (int simulationSpeed)  
*Set the Simulation Speed object.*

### 4.16.1 Detailed Description

Class used for time management in the simulation. Includes a clock that runs with regard to system time. One second in real life equals one minute in simulation time.

### 4.16.2 Member Function Documentation

#### 4.16.2.1 GetDayNumber()

```
int SimulationClock::GetDayNumber ( )
```

Returns the day number of the simulation.

##### Returns

An integer value that represents the day number

#### 4.16.2.2 GetElapsedTime()

```
double SimulationClock::GetElapsedTime ( )
```

Returns the elapsed time in simulation.

##### Returns

A double value that represents the elapsed time in seconds

#### 4.16.2.3 GetSimulationTime()

```
std::string SimulationClock::GetSimulationTime ( )
```

Returns the simulation's time (1 second in real life = 1 minute in simulation time)

##### Returns

A string value that contains the time in "xx:xx" form

#### 4.16.2.4 SetSimulationSpeed()

```
void SimulationClock::SetSimulationSpeed (
    int simulationSpeed ) [inline]
```

Set the Simulation Speed object.

##### Parameters

<i>simulationSpeed</i>	
------------------------	--

The documentation for this class was generated from the following files:

- src/simulationClock.hpp
- src/simulationClock.cpp

## 4.17 Simulator Class Reference

Class that handles the crucial parts of simulation. This class includes the main threads that the user uses to control the simulation. The simulation is initialized/finished in this class.

```
#include <simulator.hpp>
```

### Public Member Functions

- [Simulator](#) ()  
*Construct a new [Simulator](#) object.*
- [~Simulator](#) ()  
*Destroy the [Simulator](#) object.*
- void [StartSimulation](#) ()  
*Starts and initializes the simulation.*
- void [SimulatorThread](#) ()  
*The simulator's main thread that is used to run the simulation.*
- void [UpdateSimulation](#) (float deltaTime, float currentTime)  
*Updates the simulation with regard to time changes.*

- void [DrawSimulation](#) ([Visualization](#) \*gui)  
*Draws the simulation if the GUI is enabled.*
- [City](#) \* [LoadCity](#) ()  
*Loads the city from a .json file.*
- void [InputThread](#) (std::shared\_future< void > exitFuture)  
*Thread that is used for user input via terminal.*
- void [SetCity](#) ([City](#) \*c)  
*Set the [City](#) object.*

### 4.17.1 Detailed Description

Class that handles the crucial parts of simulation. This class includes the main threads that the user uses to control the simulation. The simulation is initialized/finished in this class.

### 4.17.2 Member Function Documentation

#### 4.17.2.1 DrawSimulation()

```
void Simulator::DrawSimulation (
    Visualization * gui )
```

Draws the simulation if the GUI is enabled.

##### Parameters

<i>gui</i>	graphical user interface used in visualization
------------	--

#### 4.17.2.2 InputThread()

```
void Simulator::InputThread (
    std::shared_future< void > exitFuture )
```

Thread that is used for user input via terminal.

##### Parameters

<i>exitFuture</i>	information about the thread status (exit or not)
-------------------	---



#### 4.17.2.3 LoadCity()

```
City * Simulator::LoadCity ( )
```

Loads the city from a .json file.

##### Returns

City\*

#### 4.17.2.4 SetCity()

```
void Simulator::SetCity (
    City * c ) [inline]
```

Set the City object.

##### Parameters

<i>c</i>	city that is set
----------	------------------

#### 4.17.2.5 UpdateSimulation()

```
void Simulator::UpdateSimulation (
    float deltaTime,
    float currentTime )
```

Updates the simulation with regard to time changes.

##### Parameters

<i>deltaTime</i>	tells how much time has passed in the main loop
<i>currentTime</i>	tells what is the current system time

The documentation for this class was generated from the following files:

- src/simulator.hpp
- src/simulator.cpp

## 4.18 TrafficLight Class Reference

A class for representing the traffic lights in the simulated city.

```
#include <trafficlight.hpp>
```

## Public Member Functions

- [TrafficLight](#) (std::pair< int, int > location, int redAndGreenDuration, int yellowDuration)  
*Construct a new Traffic Light object.*
- std::pair< int, int > [GetLocation](#) () const  
*Get the coordinates of the traffic light's location.*
- int [GetRedDuration](#) () const  
*Get the red light duration of the traffic light.*
- int [GetYellowDuration](#) () const  
*Get the yellow light duration of the traffic light.*
- int [GetGreenDuration](#) () const  
*Get the green light duration of the traffic light.*
- void [TurnRed](#) ()  
*Switches the state of the traffic light to red.*
- void [TurnYellow](#) ()  
*Switches the state of the traffic light to yellow.*
- void [TurnGreen](#) ()  
*Switches the state of the traffic light to green.*
- std::string [GetStatus](#) () const  
*Get the current color of the traffic light.*

### 4.18.1 Detailed Description

A class for representing the traffic lights in the simulated city.

### 4.18.2 Constructor & Destructor Documentation

#### 4.18.2.1 TrafficLight()

```
TrafficLight::TrafficLight (
    std::pair< int, int > location,
    int redAndGreenDuration,
    int yellowDuration ) [inline]
```

Construct a new Traffic Light object.

#### Parameters

<i>location</i>	The coordinates of the traffic light
<i>redDuration</i>	The duration of the red light
<i>yellowDuration</i>	The duration of the yellow light
<i>greenDuration</i>	The duration of the green light

### 4.18.3 Member Function Documentation

#### 4.18.3.1 GetGreenDuration()

```
int TrafficLight::GetGreenDuration ( ) const [inline]
```

Get the green light duration of the traffic light.

##### Returns

Green light duration

#### 4.18.3.2 GetLocation()

```
std::pair<int, int> TrafficLight::GetLocation ( ) const [inline]
```

Get the coordinates of the traffic light's location.

##### Returns

The coordinates of the traffic light as `std::pair<int, int>`

#### 4.18.3.3 GetRedDuration()

```
int TrafficLight::GetRedDuration ( ) const [inline]
```

Get the red light duration of the traffic light.

##### Returns

Red light duration

#### 4.18.3.4 GetStatus()

```
std::string TrafficLight::GetStatus ( ) const [inline]
```

Get the current color of the traffic light.

##### Returns

A string representing the current state of the traffic light ("red", "yellow", or "green")

#### 4.18.3.5 GetYellowDuration()

```
int TrafficLight::GetYellowDuration ( ) const [inline]
```

Get the yellow light duration of the traffic light.

##### Returns

Yellow light duration

The documentation for this class was generated from the following file:

- src/trafficlight.hpp

## 4.19 Visualization Class Reference

Class used for visualizing the traffic simulator in a separate GUI.

```
#include <visualization.hpp>
```

### Public Member Functions

- [Visualization](#) (int cellSize, [Grid](#) \*g)  
*Construct a new [Visualization](#) object.*
- sf::RenderWindow & [GetWindow](#) ()  
*Used for returning the Window object.*
- void [PrintGrid](#) ([Grid](#) \*grid)  
*Prints out the current grid.*
- void [PrintCars](#) (std::vector< [Car](#) \* > cars)  
*Prints out the current cars.*

#### 4.19.1 Detailed Description

Class used for visualizing the traffic simulator in a separate GUI.

#### 4.19.2 Constructor & Destructor Documentation

##### 4.19.2.1 Visualization()

```
Visualization::Visualization (
    int cellSize,
    Grid * g )
```

Construct a new [Visualization](#) object.

## Parameters

<i>cellSize</i>	cell size in the GUI
<i>g</i>	pointer to a grid

### 4.19.3 Member Function Documentation

#### 4.19.3.1 GetWindow()

```
sf::RenderWindow & Visualization::GetWindow ( )
```

Used for returning the Window object.

## Returns

sf::RenderWindow&

#### 4.19.3.2 PrintCars()

```
void Visualization::PrintCars (
    std::vector< Car * > cars )
```

Prints out the current cars.

## Parameters

<i>cars</i>	
-------------	--

#### 4.19.3.3 PrintGrid()

```
void Visualization::PrintGrid (
    Grid * grid )
```

Prints out the current grid.

## Parameters

<i>grid</i>	
-------------	--

The documentation for this class was generated from the following files:

- `src/visualization.hpp`
- `src/visualization.cpp`

# Index

- AddBuilding
  - City, [23](#)
- AddClock
  - City, [23](#)
- AddConnection
  - Node, [44](#)
- AddEvent
  - Car, [13](#)
  - City, [24](#)
  - Person, [46](#)
- AddIntersection
  - City, [24](#)
- AddPersonAndCar
  - City, [24](#)
- AddRoad
  - City, [25](#)
- AddTrafficLight
  - City, [25](#)
  - Intersection, [40](#)
- AllowHorizontal
  - Intersection, [40](#)
- AllowVertical
  - Intersection, [40](#)
- Analysis, [7](#)
  - Analysis, [7](#)
  - ExportToCSV, [8](#)
  - GenerateHourlyHistogram, [8](#)
  - GetData, [8](#)
  - SpecifyRoad, [8](#)
- AtDestination
  - Car, [13](#)
- Building, [9](#)
  - Building, [10](#)
  - Draw, [10](#)
  - GetLocation, [11](#)
  - GetName, [11](#)
  - GetType, [11](#)
- BuyCar
  - Person, [47](#)
- Car, [12](#)
  - AddEvent, [13](#)
  - AtDestination, [13](#)
  - Car, [13](#)
  - CarInFront, [14](#)
  - CheckIntersection, [14](#)
  - Dijkstra, [14](#)
  - Draw, [15](#)
  - GetDirection, [15](#)

- GetIntersection, [15](#)
- InitializeSchedule, [16](#)
- LabelsFree, [16](#)
- SetColor, [16](#)
- SetDestination, [17](#)
- SetDirection, [17](#)
- SetSpeedLimit, [17](#)
- Update, [17](#)
- YieldRight, [18](#)
- CarInFront
  - Car, [14](#)
- Cell, [18](#)
  - Cell, [19](#)
  - Draw, [19](#)
  - GetType, [20](#)
  - GetX, [20](#)
  - GetY, [20](#)
  - IsOccupied, [20](#)
  - Occupy, [21](#)
- CheckIntersection
  - Car, [14](#)
- ChooseRoad
  - City, [25](#)
- City, [21](#)
  - AddBuilding, [23](#)
  - AddClock, [23](#)
  - AddEvent, [24](#)
  - AddIntersection, [24](#)
  - AddPersonAndCar, [24](#)
  - AddRoad, [25](#)
  - AddTrafficLight, [25](#)
  - ChooseRoad, [25](#)
  - City, [23](#)
  - DrawBuildings, [26](#)
  - DrawCars, [26](#)
  - DrawIntersections, [26](#)
  - GetBuildingNodes, [26](#)
  - GetCars, [27](#)
  - GetGrid, [27](#)
  - GetIntersection, [27](#)
  - GetNode, [28](#)
  - GetRoads, [28](#)
  - IsBusy, [28](#)
  - IsValidBuilding, [29](#)
  - IsValidIntersection, [29](#)
  - IsValidPerson, [29](#)
  - IsValidRoad, [30](#)
  - PrintCity, [30](#)
  - TimeUntilNextEvent, [30](#)

- UpdateCars, 31
- UpdateIntersections, 31
- Commercial, 31
  - Commercial, 32
  - Draw, 33
- CreateSchedule
  - Event, 35
- Dijkstra
  - Car, 14
- Draw
  - Building, 10
  - Car, 15
  - Cell, 19
  - Commercial, 33
  - Industrial, 39
  - Intersection, 41
  - Residential, 51
- DrawBuildings
  - City, 26
- DrawCars
  - City, 26
- DrawIntersections
  - City, 26
- DrawSimulation
  - Simulator, 56
- Event, 33
  - CreateSchedule, 35
  - Event, 33
- ExportToCSV
  - Analysis, 8
- GenerateHourlyHistogram
  - Analysis, 8
- GetBuildingNodes
  - City, 26
- GetCar
  - Person, 47
- GetCars
  - City, 27
- GetCell
  - Grid, 36
- GetConnections
  - Node, 44
- GetData
  - Analysis, 8
- GetDayNumber
  - SimulationClock, 54
- GetDirection
  - Car, 15
- GetElapsedTime
  - SimulationClock, 54
- GetEnd
  - Road, 52
- GetError
  - InvalidCityException, 43
- GetGreenDuration
  - TrafficLight, 59
- GetGrid
  - City, 27
- GetIntersection
  - Car, 15
  - City, 27
- GetLocation
  - Building, 11
  - Intersection, 41
  - Node, 44
  - Person, 47
  - TrafficLight, 59
- GetName
  - Building, 11
  - Person, 47
- GetNeighborCells
  - Grid, 36
- GetNode
  - City, 28
- GetPersonType
  - Person, 48
- GetRedDuration
  - TrafficLight, 59
- GetResidence
  - Person, 48
- GetRoads
  - City, 28
- GetSchedule
  - Person, 48
- GetSimulationTime
  - SimulationClock, 54
- GetSizeX
  - Grid, 37
- GetSizeY
  - Grid, 37
- GetSpeedLimit
  - Road, 52
- GetStart
  - Road, 52
- GetStatus
  - TrafficLight, 59
- GetType
  - Building, 11
  - Cell, 20
  - Node, 45
- GetWindow
  - Visualization, 61
- GetWorkplace
  - Person, 48
- GetX
  - Cell, 20
- GetY
  - Cell, 20
- GetYellowDuration
  - TrafficLight, 59
- Grid, 35
  - GetCell, 36
  - GetNeighborCells, 36
  - GetSizeX, 37



- GetSizeY, [37](#)
- Grid, [36](#)
- Industrial, [38](#)
  - Draw, [39](#)
  - Industrial, [38](#)
- InitializeSchedule
  - Car, [16](#)
  - Person, [49](#)
- InputThread
  - Simulator, [56](#)
- Intersection, [39](#)
  - AddTrafficLight, [40](#)
  - AllowHorizontal, [40](#)
  - AllowVertical, [40](#)
  - Draw, [41](#)
  - GetLocation, [41](#)
  - Intersection, [40](#)
  - Update, [41](#)
- InvalidCityException, [42](#)
  - GetError, [43](#)
  - InvalidCityException, [42](#)
- isAtHome
  - Person, [49](#)
- IsBusy
  - City, [28](#)
- IsHorizontal
  - Road, [53](#)
- IsOccupied
  - Cell, [20](#)
- IsValidBuilding
  - City, [29](#)
- IsValidIntersection
  - City, [29](#)
- IsValidPerson
  - City, [29](#)
- IsValidRoad
  - City, [30](#)
- IsVertical
  - Road, [53](#)
- LanelIsFree
  - Car, [16](#)
- LoadCity
  - Simulator, [56](#)
- Node, [43](#)
  - AddConnection, [44](#)
  - GetConnections, [44](#)
  - GetLocation, [44](#)
  - GetType, [45](#)
  - Node, [44](#)
- Occupy
  - Cell, [21](#)
- Person, [45](#)
  - AddEvent, [46](#)
  - BuyCar, [47](#)
  - GetCar, [47](#)
  - GetLocation, [47](#)
  - GetName, [47](#)
  - GetPersonType, [48](#)
  - GetResidence, [48](#)
  - GetSchedule, [48](#)
  - GetWorkplace, [48](#)
  - InitializeSchedule, [49](#)
  - isAtHome, [49](#)
  - Person, [46](#)
  - UpdateLocationFromCar, [49](#)
- PrintCars
  - Visualization, [61](#)
- PrintCity
  - City, [30](#)
- PrintGrid
  - Visualization, [61](#)
- Residential, [50](#)
  - Draw, [51](#)
  - Residential, [50](#)
- Road, [51](#)
  - GetEnd, [52](#)
  - GetSpeedLimit, [52](#)
  - GetStart, [52](#)
  - IsHorizontal, [53](#)
  - IsVertical, [53](#)
  - Road, [52](#)
- SetCity
  - Simulator, [57](#)
- SetColor
  - Car, [16](#)
- SetDestination
  - Car, [17](#)
- SetDirection
  - Car, [17](#)
- SetSimulationSpeed
  - SimulationClock, [55](#)
- SetSpeedLimit
  - Car, [17](#)
- SimulationClock, [53](#)
  - GetDayNumber, [54](#)
  - GetElapsedTime, [54](#)
  - GetSimulationTime, [54](#)
  - SetSimulationSpeed, [55](#)
- Simulator, [55](#)
  - DrawSimulation, [56](#)
  - InputThread, [56](#)
  - LoadCity, [56](#)
  - SetCity, [57](#)
  - UpdateSimulation, [57](#)
- SpecifyRoad
  - Analysis, [8](#)
- TimeUntilNextEvent
  - City, [30](#)
- TrafficLight, [57](#)
  - GetGreenDuration, [59](#)

- GetLocation, [59](#)
- GetRedDuration, [59](#)
- GetStatus, [59](#)
- GetYellowDuration, [59](#)
- TrafficLight, [58](#)

Update

- Car, [17](#)
- Intersection, [41](#)

UpdateCars

- City, [31](#)

UpdateIntersections

- City, [31](#)

UpdateLocationFromCar

- Person, [49](#)

UpdateSimulation

- Simulator, [57](#)

Visualization, [60](#)

- GetWindow, [61](#)
- PrintCars, [61](#)
- PrintGrid, [61](#)
- Visualization, [60](#)

YieldRight

- Car, [18](#)