Project SIMLANE

Software Architecture and Requirements Specification

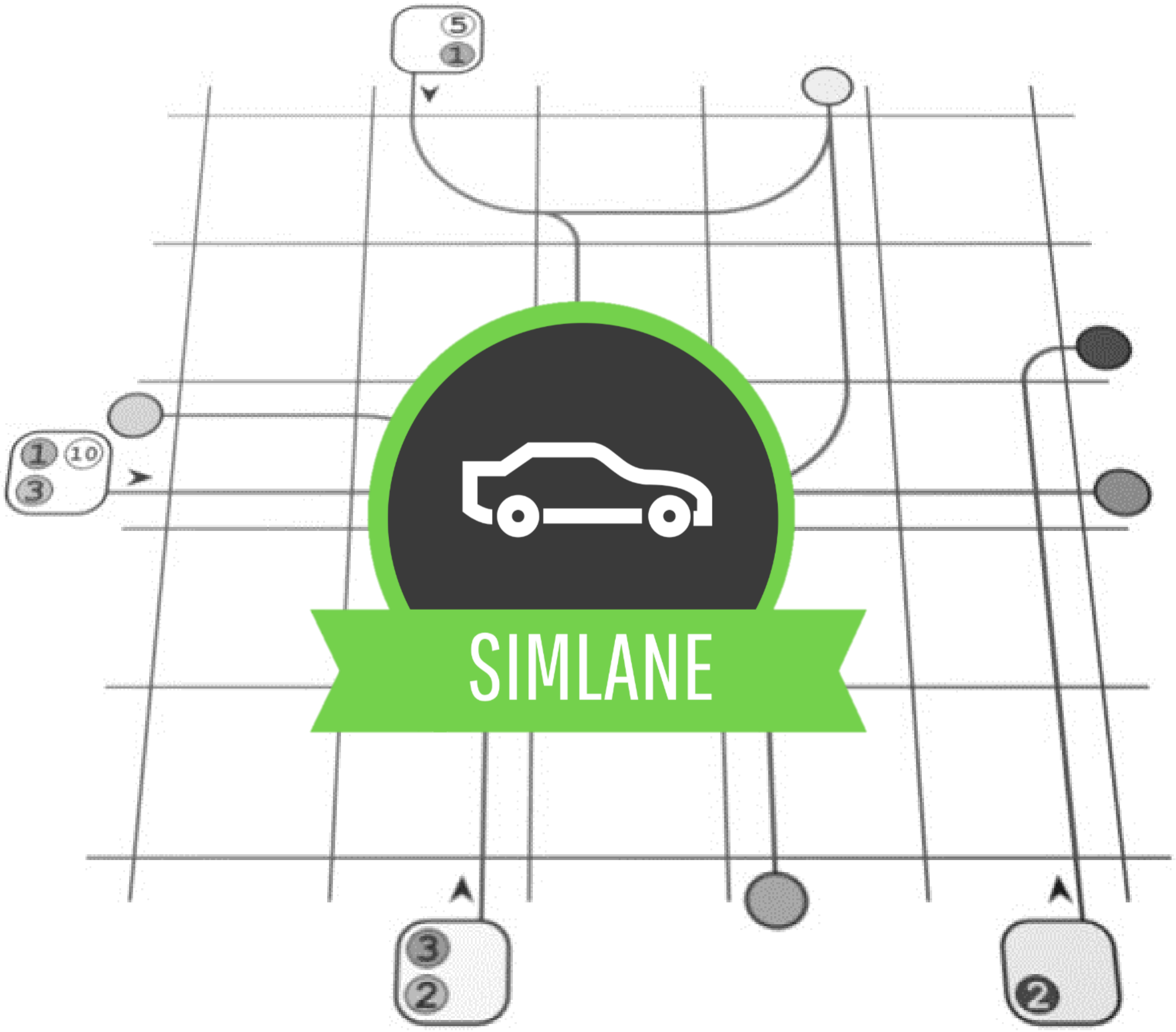


Table of Contents

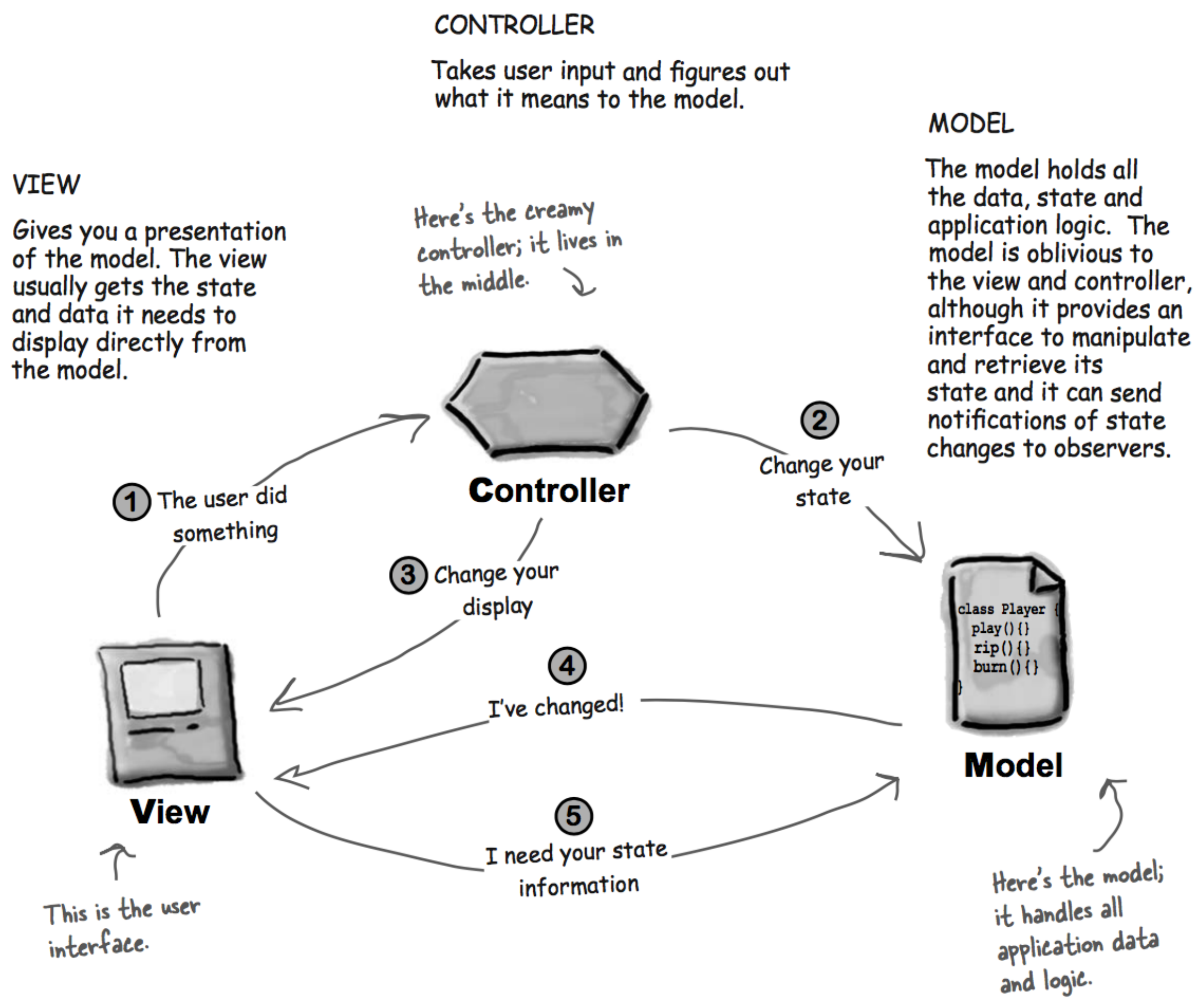
1. Software Architecture

The following chapter describes the overall structure of the SIMLANE project. The software is divided into multiple components that interact with each other in well-defined ways. The individual components as well as the interactions between them will be explained below.

1.1. Design Pattern

The SIMLANE software project mostly follows the **Mode View Controller (MVC)** design pattern. This design pattern is a very common way of structuring applications that offer a graphical user interface. The goal of applying the MVC design pattern is to separate the code used for displaying the graphical user interface from the code that makes up the actual business logic of the application.

The following image gives you an overview of the MVC architecture. The image shows the three main components of the MVC design pattern and it also illustrates the interactions between these components using arrows. These interactions are very important for the MVC pattern because they define a clear way of communication between the different parts of the software.



CONTROLLER
Takes user input and figures out what it means to the model.
MODEL
The model holds all the data, state and application logic. The model is oblivious to the view and controller, although it provides an interface to manipulate and retrieve its state and it can send notifications of state changes to observers.
VIEW
Gives you a presentation of the model. The view usually gets the state and data it needs to display directly from the model.
Here's the creamy controller; it lives in the middle.
Controller
1 The user did something
2 Change your state
3 Change your display
4 I've changed!
5 I need your state information
class Player { play() {} rip() {} burn() {} }
Model
View
This is the user interface.
Here's the model; it handles all application data and logic.

Picture 1: An illustration of the MVC design pattern
Source: https://stackoverflow.com/questions/5217611/the-mvc-pattern-and-swing

1.1.1. MVC Structure

In the MVC design pattern, there are the following three components:

- **Model:**

  The model contains the business logic of the application. It consists of classes that represent all the (real world) objects which are part of the application. These classes contain fields to hold the current state of the object that they represent, and they also contain public methods to modify this state.

  The model does not contain any information which is related to how the objects will be represented in the graphical user interface.

- **View:**

  The view contains all the code that is needed to create the graphical user interface. The view must represent the state of the model in the graphical user interface.

- **Controller:**

  The controller is responsible for the communication between the model and the view.

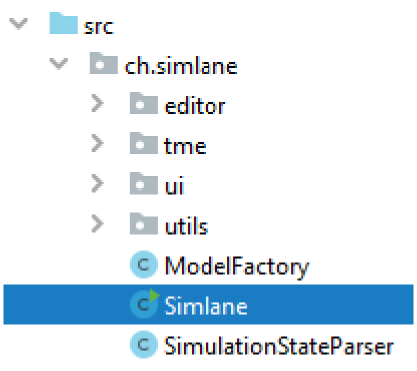
1.1.2. Interaction Between the Components in MVC

The image above illustrates five different interactions between the components. Each interaction is indicated using an arrow with a number. All these interactions (except number 3) are present in the SIMLANE project. They are explained in the following chapter.

1.2. The MVC Design Pattern in SIMLANE

The code of the SIMLANE project is structured into multiple packages. Each package contains classes that serve a similar purpose. The following chapter explains how these packages fit into the MVC design pattern.

1.2.1. Overview



SIMLANE is divided into the following packages:

- ch.simlane.editor
  This package contains the **Model** of the MVC design pattern.

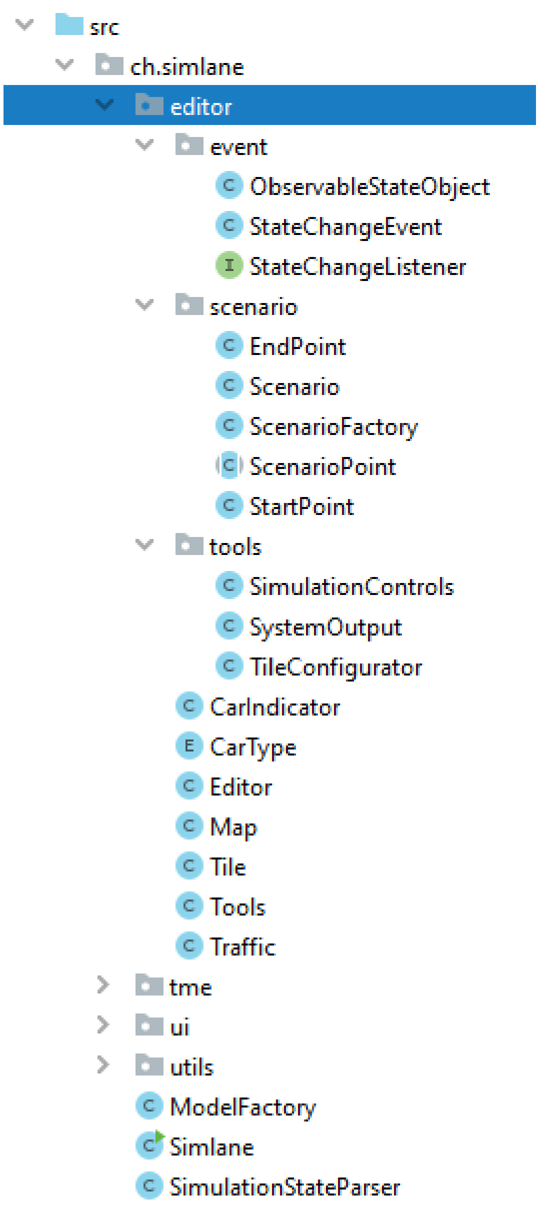
- ch.simlane.tme
  This package contains the code for the TME. It is not part of the overall MVC architecture.

- ch.simalne.ui
  This package makes up **View** of the MVC design pattern.

- Simlane.java
  This class represents the **Controller** in the MVC design pattern. It is also the main entry point to the application which creates the Model, the View and it instantiates the TME.

1.2.2. Package ch.simalne.editor (Model)

The editor package contains the **Model** of the MVC architecture.



The package is called **editor** because at the top level, **SIMALNE consists of an editor**, that allows the user to create a custom network of streets.

The Editor class is therefore the main class of the Model. It creates and contains all other parts of the Model.

The two most important classes in this package are the ones that represent the map (Map) and its individual tiles (Tile).

The Tile class represents the state of a single tile. The most important part of a tile's state is the information about the lanes that it contains. Each tile needs to know which of the 12 possible lanes have been drawn onto it by the user.

⚠ Lanes are not a separate class in the Model. They are part of the state of a tile.

The TME however does contain a class which represents lanes. This is because the TME does not know anything about tiles. Instead, the TME models lanes which are connected to each other in a graph. Those lanes have physical properties and they can contain cars.

The ch.simlane.editor.scenario sub-package contains classes that represent the current scenario. A scenario consists of the start and end points where cars enter respectively leave the user designed street network.

This package also contains a scenario factory which is responsible for creating the scenario object by reading the definition in the scenarios.json file.

Interaction

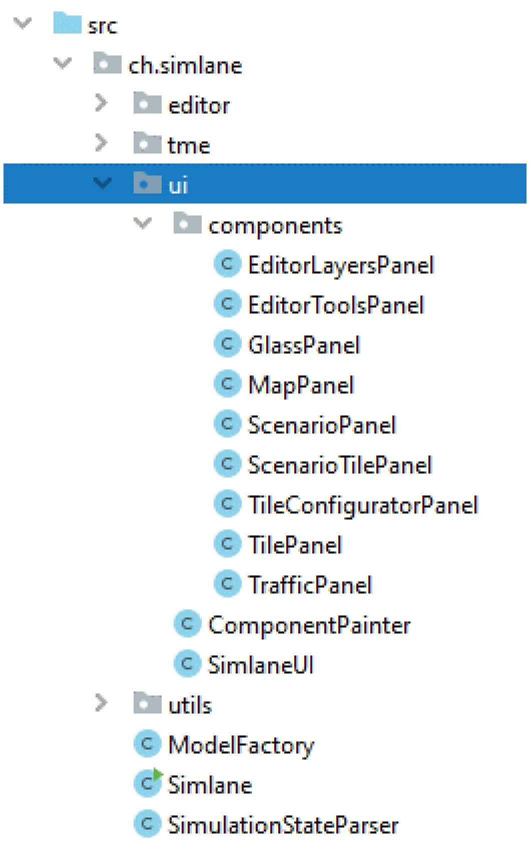
As illustrated by the arrow number 4 in picture 1, the Model does interact with the View. The Model needs to inform the View if it has changed its state. This is necessary to make the View react to any changes in the Model.

In SIMLANE, every component of the **Model** that can change its state, is a subclass of ObservableStateObject. This class provides a method which allows the **View** to register itself to a class of the Model as a listener. In order to do that, the view must simply implement the StateChangeListener interface and call the **addStateChangeListener()** method of the Model class.

When the model changes, every registered listener will be notified about the change by calling the listeners **stateChange()** method with an appropriate StateChangeEvent object.

1.2.3. Package ch.simlane.ui (View)

The ui package contains the **View** of the MVC architecture.



```
src
  ch.simlane
    editor
    tme
    ui
      components
        EditorLayersPanel
        EditorToolsPanel
        GlassPanel
        MapPanel
        ScenarioPanel
        ScenarioTilePanel
        TileConfiguratorPanel
        TilePanel
        TrafficPanel
      ComponentPainter
      SimlaneUI
    utils
    ModelFactory
    Simlane
    SimulationStateParser
```

The ui package is responsible for presenting the Model's state to the user by creating a frame with various panels.

This package contains classes that create the visual representation of the classes from the Model. The most important ones are the MapPanel and the TilePanel as well as the ScenarioPanel and the corresponding ScenarioTilePanel.

The SimlaneUI class is the main class for the user interface. The Controller creates an instance of this class and passes it a reference to the Model.

Interaction

The View can only exist with a reference to the Model. Because the View needs to ask the Model for its state. This is illustrated by arrow number 5 in picture 1.

In order to allow the Controller to react to user input such as pressing a button or clicking a tile, a listener is attached to the components of the View.

This is done by the Controller. Immediately after the Controller has created the View, it calls its **addSimlaneUIListener()** method and attaches a new SimlaneUIListener to the View. The SimlaneUIListener object gets notified about user interactions (illustrated by arrow number 1 in picture 1) and it can change the Model (illustrated by arrow number 2 in picture 1) based on what the user does.

1.2.4. Simlane.java (Controller)

The class Simlane.java makes up the **Controller** of the MVC architecture.

Simlane.java is the main entry point for the SIMLANE application. It contains the static main() method which creates an instance of the Simlane class.

The constructor of the Simlane class first creates the Model. Then it creates the View by invoking the SimlaneUI constructor, which takes a reference to the previously created Model as its only argument.

The Controller must also instantiate the **Engine** class. This class represents the **TME**.

2. Requirements Specification

The following chapter specifies the required functionality of the TME and its related software components. It also specifies how the different parts of the TME interact with each other and how the TME interacts with the rest of the SIMLANE application.

Definitions for the behavior of public methods are written using the perspective of someone who might uses the defined methods from outside the class. It is a good idea to divide complex public methods into multiple private sub methods.

2.1. Model Factory

The Model Factory is responsible for translating the map and the scenario (which are both part of the **Model of the editor**) into the **Model for the TME**. Before reading this chapter, I suggest reading the section about the term "Model" in chapter 4.

2.1.1. Implementation Overview

Class Name:	`ModelFactory`
Constructor:	`public ModelFactory()`
Methods:	`public Model getModel(Map map, Scenario scenario)`

2.1.2. Purpose

The **Model of the editor** contains a **Map** object and a **Scenario** object. The Map object contains the information about the custom street network that the user has designed, and the scenario object contains the information about the start and end points of cars, which surround the map.

The information from those two objects (Map and Scenario) is in a form that is suitable only for the Editor. The map keeps an array of individual tile objects that contain information about their lanes. The Tile class also contains methods so the controller can change the lanes that are present on the tile if a user changes his street network.

Once the user is done editing his street network, he wants to start the simulation of traffic. At this point, the information in the **Map** and **Scenario** objects needs to be translated into a new object, which is suitable for simulation. This new object is called the **Model for the TME**, because it's a representation of the street network and of the cars, which the TME can use for simulation.

This translation is performed by the **ModelFactory** class. The translation gets initiated by the **SimlaneUIListener** when a user clicks on the start simulation button. The **SimlaneUIListner** calls the getModel() method of the Model Factory. If the **Model for the TME** was successfully created and returned by the getModel() method, the **SimlaneUIListener** will pass the **Model for the TME** on to the **Engine** class of the TME using its loadModel() method.

2.1.3. Constructor

The Model Factory has an empty constructor with no arguments. This constructor gets called by the Simlane class (Controller) to create the Model Factory.

2.1.4. Methods

```
public Model getModel(Map map, Scenario scenario)
```

The most important task of the getModel() method is to create a graph that represents the custom street network. This graph is part of the Model object (the Model for the TME) which must be returned by this method. The structure of the graph is described in the chapter "Lane and Connector Graph". You should read that chapter now.

The graph gets created by walking through each tile of the map. The graph is expanded with more connectors and lanes as more tiles are visited while traversing the map. The graph creation process is described below.

While the graph gets created, the method ensures that no lanes and no start or end points are unconnected. If an unconnected lane, start or end point is encountered, the creation of the graph gets interrupted and an appropriate exception must be thrown.

I suggest creating a custom exception class, by creating a new class inside the ch.simlane package. This is necessary because the SimlaneUIListener must catch that exception and handle it in a way that the user gets notified about the error in his street network.

<u>**Lane and Connector Graph Creation Process**</u>

Once the complete **lane and connector graph** is created, the Model will only hold references to the graph's Connector objects that represent **startPoints**. However, during the creation process, a temporary HashMap will contain references to additional Connector objects of the graph.

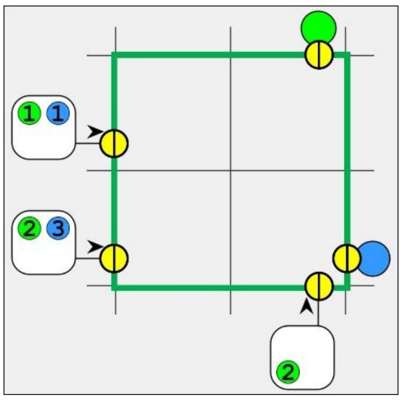
HashMap<ConnectorLocation, Connector>

This HashMap has the name **"unconnected"**, because it will contain all Connector objects that are not yet connected. A Connector object is called "unconnected" if either its **in** or **out** reference has not yet been set. You can think of an "unconnected" Connector as being only **one half of a Connector**. Either the **IN** or the **OUT** part of the Connector object is still missing.

An **unconnected** Connector does however always have properly set **refIn** and **refOut** values. This is important, in order to unambiguously identify all the existing Connector objects in the HashMap.

The HashMap maps ConnectorLocation objects onto Connector objects. This allows us to retrieve a Connector from the HashMap by only knowing its location. A ConnectorLocation object uses two tile references (**tileFrom**, **tileTo**) to unambiguously specify the location of a Connector. The same concept is used when setting the **refIn** and **refOut** values of a Connector object.

Creating the Connectors for the StartPoints and EndPoints



The very first part of the **lane and connector graph** that gets created consists of the Connectors for all the StartPoints and EndPoints of the Scenario.

The picture on the left shows an example for a scenario with various start and end points. The Connectors for those start and end points are all located somewhere on the green border shown in the picture on the left. The Connector objects are indicated using yellow circles.

To create those Connector objects, the Scenario object (which gets passed to this method as a parameter) must be used. The Scenario object holds the information about all the start and end points.

This information can be used to create a Connector object for every start and end point. Those Connector objects do not yet refer to any incoming or outgoing lanes. References to lanes will be added later, when traversing through the map. When creating the Connector objects for the start and end points, it is however important, that their location is properly set using the **refIn** and **refOut** parameters of the Connector constructor.

The location of those Connectors is at the **border of the map** (indicated using a green border in the picture above). This means, the Connectors are not located in between two actual map tiles. Instead, every Connector of a start or end point is located **on the side of just one map tile**. It is however necessary to specify two tiles for the location of every Connector (**refIn** and **refOut**).

In order to have two tiles, we simply create a "substitute" tile which has a location outside of the actual map. The location is specified using the **row** and **col** parameters when constructing the Tile object. The resulting Tile object will never contain any lanes, it will only be used for specifying the location of a Connector.

All Connector objects that are created for the start and end points must be added to the **"unconnected" HashMap** using a corresponding ConnectorLocation object as their key. When constructing the ConnectorLocation objects, we will again need to create "substitute" tiles. A ConnectorLocation object for a Connector of a start or end point will always refer to one actual map tile and to one "substitute" tile.

The Connectors for the start points must also be added to the List<Connector> **startPoints** filed of the Model. The order of the Connectors in this list is arbitrary.

When creating the cars, we will need references to the Connector objects of the start and end points as well. Those Connectors must therefore be added to additional HashMaps as described in the chapter "StartPoint and EndPoint HashMaps".

Traversing the Map

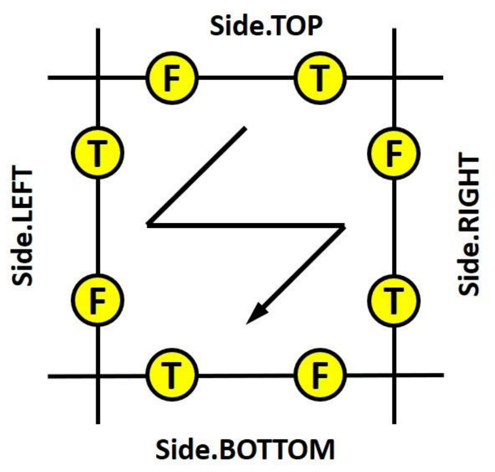
The next part of the **lane and connector graph** that will be created consists of the actual lanes and the connectors in between those lanes.

In order to create those lanes and connectors, the map will be traversed as described in the chapter "Map Traversal Order". The map object is passed to this method as a parameter.

As the map gets traversed, each tile of the map will be processed separately. For every tile, corresponding **Lane** and **Connector** objects will be created and then added to the **lane and connector graph** before proceeding to the next tile.

While a single tile is being processed, additional temporary references to Connector objects must be stored. Those references will be stored in two arrays of Connector objects. Each of those arrays must have a length of 4. The first array (called **"fromConnectors"**) will contain the 4 Connectors that are located on the right-hand side of each side of the current tile (labeled **F** in the picture below). The second array (called **"toConnectors"**) will contain the 4 Connectors that are located on the left-hand side of each side of the current tile (labeled **T** in the picture below).

The picture below illustrates to which array the 8 Connectors of the current tile belong. The picture does also indicate the order of those Connector objects in the two arrays. It is important that the order matches the ARRAY_INDEX_? constants defined in the class Editor (Top, Left, Right, Bottom).



Before the lanes of the current tile get added to the graph, the **fromConnectors** and **toConnectors** arrays must be initialized with appropriate Connector objects. Those Connector objects will not contain any references to lanes yet. As with the Connectors of the start and end points that were created earlier, it is important, that the location of those new Connector objects is properly set using the **refIn** and **refOut** parameters of the Connector constructor.

Note: While traversing the map, tiles that are located on the border of the map (right next to where a start or endpoint might be located) must be treated in a special way (refer to the chapter "Map Traversal" for details). For those tiles, some Connectors can only be created using "substitute" tiles for their **refIn** or **refOut** values. The same goes for the creation of ConnectorLocation objects.

Once the **fromConnectors** and **toConnectors** arrays have been initialized with new Connector objects, the lanes of the current tile will be added to the graph. In order to do this, we traverse through all the lanes of the current tile and for every lane that is **enabled**, we create a new **Lane** object.

The **start** and **end** fields (references to Connector objects) of the new **Lane** object must be set properly. The **start** field of the **Lane** will reference one of the Connectors found in the **fromConnectors** array. The **end** field of the **Lane** will reference one of the Connectors found in the **toConnectors** array.

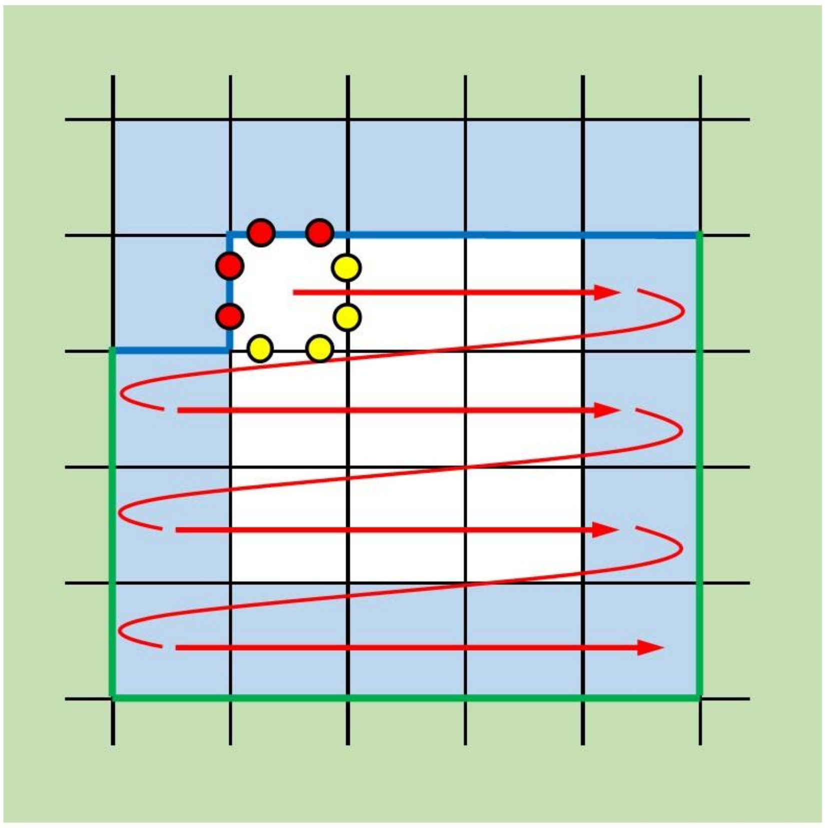
Lanes that start at the same side must be added to one **LaneGroup**. Even if there is only a single lane that starts on a specific side, it is important to create a **LaneGroup** for the lane. Because lanes that start at the same side must be added to the same **LaneGroup**, the lanes of the tile should be traversed in an appropriate order.

For every individual **Lane**, there is a corresponding Connector in the **toConnectors** array (the Connector matches the lanes **end** Connector). Every Lane must be added to the **in** List of its corresponding Connector.

For every **LaneGroup**, there is a corresponding Connector in the **fromConnectors** array (the Connector matches the common **start** Connector of the lanes inside the **LaneGroup**). It is necessary to set the **out** reference of the corresponding Connector so that it points to its **LaneGroup**.

If all lanes of the tile have been processed, it might be necessary to create an **Intersection** object for this tile. This is necessary if the tile contains lanes that intersect with each other. To check whether an **Intersection** must be created use the steps described in the chapter “Intersecting lanes check and Intersection creation”.

The next step is to the connect the Connectors in the temporary **fromConnectors** and **toConnectors** arrays to the rest of the **lane and connector graph**.



The picture above shows the complete map that is being traversed. The tile that is currently being processed is located in row 1 and column 1. For this tile the temporary **fromConnectors** and **toConnectors** are shown using circles.

When processing a tile, only the Connectors located at its top and left side (shown in red) are added to the **lane and connector graph**. The Connectors at the right and at the bottom side of the tile (shown in yellow) will be added to the graph at a later point. Those Connectors are not yet complete (they are still **“unconnected”**). See the remark on corner cases for more details.

In order to add the red Connectors to the **lane and connector graph**, they need to be processed in the following way:

Every red Connector must be joined (connected to) its corresponding **“unconnected”** Connector stored in the **“unconnected”** HashMap. The process of joining two **“unconnected”** Connectors will create one complete Connector object that is ready to be added to the **lane and connector graph**.

Before connecting two Connectors, a few validations must be made. Those validations ensure, that the map does not contain unconnected lanes.

Remark on corner cases

When processing a tile that is the last tile in the current row, the Connectors that are located on the right-hand side of the tile must also be added to the **lane and connector graph**.

When processing a tile that is located in the last row of the map, the Connectors that are located on the bottom side of the tile must also be added to the **lane and connector graph**.

Processing the top and left (red) Connectors

The following section describes how a Connector gets validated and then added to the **lane and connector graph**. This process must be executed for every Connector located at the top and left side of the tile (the red Connectors).

First, we need to check if the current red Connector was used for this tile. This means, we need to check if the tile contained a lane that started or ended at the current red Connector. This can be done by simply checking if the red Connectors **in** or **out** field does point to a **Lane** or **LaneGroup**. Depending on the result, we need to check the following:

- If the Connector was used for at least one lane (**out** != null || **in** != null):
  We need to ensure that a corresponding Connector (with the same location) is present in the **"unconnected"** HashMap.
  - If not, we interrupt the graph creation process because we have detected an unconnected lane. This causes the method to throw an exception as described above.
  - If such a corresponding Connector is found, we can join the red Connector with the Connector that we found in the HashMap. The process of joining two Connectors is described below.
- If the Connector was not used for any lanes (**out** == null && **in** == null):
  We need to ensure that the **"unconnected"** HashMap does not contain any corresponding Connector (with the same location).
  - If the HashMap does contain a corresponding Connector, we interrupt the graph creation process because we have detected an unconnected lane. This causes the method to throw an exception as described above.
  - If such a corresponding Connector is not found in the HashMap, we can ignore this unused red Connector and move on with the next red Connector.

Once all red Connectors have been processed, we can move on to processing the yellow Connectors.

Joining Connectors

When we join Connectors, we always start out with two **"unconnected"** connectors and end up with one complete Connector. The first one of those unconnected Connectors is the current **red Connector**. The second one is the corresponding unconnected Connector that we found in the HashMap (we will call it **corresponding Connector** from now on).

Those two Connectors are connected by copying the value of the **in** / **out** field of the **red Connector** to the **in** / **out** field of the **corresponding Connector** that we found in the HashMap. This turns the **corresponding Connector** from the HashMap into a complete Connector.

The **red Connector** is no longer used. It is important to update the **start** or **end** references of all Lanes that were previously connected to the temporary **red Connector** so that they point to the complete Connector and no longer to the **red Connector**.

The complete Connector is now part of the **lane and connector graph**. It is also no longer an unconnected Connector, which means it must be removed from the **"unconnected"** HashMap.

Adding the non-empty right and bottom (yellow) Connectors to the "unconnected" HashMap

The following section describes how the right and bottom (yellow) Connectors of the current tile must be processed. The steps in this section must be executed for every yellow Connector.

First, we need to check if the current yellow Connector is non-empty. This means we need to check if the Connector contains lanes in either it's **in** or **out** field. Depending on the result we need to do the following:

- If the Connector is empty, we can ignore it and move on the next yellow Connector.
- If the Connector is non-empty, we must add it to the **"unconnected"** HashMap.

If the Connector gets added to the **"unconnected"** HashMap, this means it was used to connect at least one lane from the current tile. The Connector will remain in an "unconnected" state until it is joined with (connected to) a corresponding red Connector at a later time.

After processing all the yellow Connectors, we can move on to processing the next tile of the map as described in the section "Traversing the Map" of this chapter.

If all tiles of the map have been successfully processed, the **"unconnected"** HashMap will be empty. And at this point, the complete **lane and connector graph** has been created.

The only part of the model that still needs to be created is the cars. Those will be created next, as described in the section below.

Creating the cars

When we created the Connectors for the start and end points in the section above, we stored them in an additional HashMaps called **startPointConnectors** and **endPointConnectors**. Those HashMaps will now be used.

The information that we need for creating the cars is stored in the **inboundTraffic** HashMap of the start points found in the Scenario. The same start points are also found in the key set of the **startPointConnectors** HashMap, which might be more convenient to access as we must iterate through all the start points.

When iterating through all the start points, we can create the required **Cars** for each individual start point as described below.

When creating a **Car,** it is important to set the following properties:

- The **ref** field of the Car. It must point to the appropriate **CarType** of the **Car**.
- The **startPoint** field. It must point to the corresponding start point **Connector**.
- The **endPoint** field. It must point to the corresponding end point **Connector**.

Each car that is created, must be added to the Models (the **Model for the TME**) **cars** list.

Returning the complete model

If all the above steps have been executed, the **Model for the TME** is successfully created. It must then be returned by the method.

2.2. Simulation State Parser

The Simulation State Parser is responsible for parsing a **SimulationState** object and changing the **Model of the editor** so that it reflects the state represented by the parsed **SimulationState** object.

2.2.1. Implementation Overview

Class Name:	`SimulationStateParser`
Constructor:	`public SimulationStateParser(Editor editor)`
Methods:	`public void parse(SimulationState simulationState)`

2.2.2. Purpose

During the simulation of traffic on a user designed street network, the position of the cars on the street network and the state of all lanes is continuously updated by the TME. The TME stores this state information for the cars and for the lanes in its model (the **Model for the TME**). This is an internal object of the TME, that cannot directly be accessed by the SIMLANE application.

The TME does however provide access to the state information inside its model (the **Model for the TME**) through **SimulationState** objects.

During the process of simulation, the TME continuously produces new **SimulationState** objects. Such a **SimulationState** object always represents the latest state of the entire simulation (it is a representation of the internal model (the **Model for the TME**). This means, a **SimulationState** object contains information about the current position of all cars and about the current state of each lane (a lane can be active or inactive) in the simulation.

A **SimulationState** object contains **LaneState** and **CarState** objects. The main purpose of the SimulationStateParser is to read the information stored in those **LaneState** and **CarState** objects and to adjust the **Model of the Editor** accordingly.

The state of the simulation changes continuously. The TME does therefore produce multiple new **SimulationState** objects per second. Whenever a new **SimulationState** object is created by the TME, an event gets fired. This event is processed by the **SimlaneTMEListener**. The **SimlaneTMEListener** invokes the parse() method of the **SimulationStateParser** passing it the new **SimulationState** object. This **SimulationState** object must then be parsed by the **SimulationStateParser**, in order to make the editor display the current simulation state. This allows the user to see the flow of traffic on his street network.

⚠ The parse() method of the **SimulationStateParser** is called multiple times per second (≈ every 200 ms). Every time, the parse() method is called, the given **SimulationState** object must be completely parsed.

This implies, that the parse() method must be fairly time efficient.

2.2.3. Constructor

The Constructor is invoked by the Simlane class (Controller). Its only parameter is a reference to an object of the type **Editor** (the **Model of the editor**). The reference is required, as this **Model of the editor** must be updated every time a new **SimulationState** object is parsed.

2.2.4. Methods

```
public void parse(SimulationState simulationState)
```

The parse() method has one parameter of the type **SimulationState**.

As the name suggests, the parse() method parses the given **SimulationState** object. This involves reading all the information that is stored in the **SimulationState** object and then changing the **Model of the editor** accordingly.

Only the following two parts of the **Model of the editor** will be changed during this process:

- The **activeLanes** array of all the **Tiles**.
  This array stores whether the individual lanes of the tile are currently active (green traffic signal) or inactive (red traffic signal).
- The **traffic** array of the **Traffic** object.
  This array contains **CarIndicator** objects that describe the location of each car on the map.

The task of the parse() method can be broken down into the following two sub tasks:

- Parsing all the **LaneState** objects of the **SimulationState** object and changing the **activeLanes** array of all the Tiles accordingly.
- Parsing all the **CarState** objects of the **SimulationState** object and changing the **traffic** array of the Traffic object accordingly.

The following sections describe those two sub tasks.

<u>**Parsing the LaneState objects**</u>

The **LaneState** objects are stored in a private List field called **laneStates**. Each **LaneState** object can be processed separately. The **LaneState** objects should however be sorted first and then processed on a per tile basis.

This means, that we should first create an empty list of **LaneState** objects for every tile on the map. For each **LaneState** object, we can then determine the tile that it belongs to and add it to the corresponding list. Once the **LaneStates** are sorted into lists, according to the Tile they belong to, we can continue by processing each of those lists.

For the sorting process, we need to determine the **Tile**, that a **LaneState** object belongs to. This can be done using the **ConnectorLocation** fields of the **LaneState** object. I suggest creating private sub methods for tasks like this.

Every list of **LaneState** objects can be processed as follows. Start by creating a temporary boolean array with a size of 4 x 4. This array will hold the new "**active**" values for every lane of the current tile.

We then process every **LaneState** object in the list sequentially. For the **LaneState** object that is currently being processed, we need to determine the **fromSide** and the **toSide** values of the lane. This can again be done using the **ConnectorLocation** fields of the **LaneState** object.

If the **fromSide** and **toSide** values have been found, they can be used to store the LaneState's **active** value at the proper location in the temporary array.

If all **LaneState** objects in the current list have been processed, the temporary array can be passed to the setActiveLanes() method of the current tile. This will update the **Model of the editor**.

If all **LaneState** objects have been processed as described above, we need to process the **CarState** objects next.

Parsing the CarState objects

The **CarState** objects are stored in a private List field called **carStates**. Each car state must be processed separately. Before we start processing the **CarState** objects, we must create an empty **CarIndicator** array. The length of the array must correspond to the length of the **carStates** list.

To process a single **CarState** object, we first need to determine the lane, that the car is located on. This can be done using the **ConnectorLocation** fields of the **CarState** object.

The next step is to calculate the **x** and **y** coordinates of the **Car**, in order to create a corresponding **CarIndicator** object. This process differs, depending on what kind of lane the **Car** is located on (refer to the chapters "CarIndicator class" and "Position Values for Cars on Lanes" for more details).

Once the **x** and **y** coordinates have been determined we can create the **CarIndicator** object and add it to the temporary **CarIndicator** array. If all **CarState** objects have been processed, the setTraffic() method of the **Traffic** class can be used to update the **Model of the editor**.

Parsing additional SimulationState information

The **SimulationState** object will contain additional information about the current state of the simulation. This data must be parsed as well, and the **Model of the editor** must be updated accordingly.

If all the data in the **SimulationState** object has been parsed and if the **Model of the editor** has been fully updated, the parse() method will return.

2.3. Engine

The **Engine** class is the main class for the TME. To use the TME, an instance of the **Engine** class must be created. The **Engine** class provides all the necessary methods that an application needs to interact with the TME.

The **Engine** class contains the following two important private fields:

- private Model **model**
  This field holds a reference to the model (the **Model for the TME**) that represents the street network and the cars for the purpose of simulation.
- private Simulation **simulation**
  This field holds a reference to the **Simulation** object. A **Simulation** object represents a simulation for a model (a **Model for the TME**).

The **Engine** object can be created without specifying any arguments. The fields mentioned above will initially have **null** values. A **model** can be loaded at a later time using the loadModel() method. The **simulation** field will be assigned a value once a simulation is started using the startSimulation() method.

An application that uses the TME usually wants to react to events that occur inside the TME. In order to listen to events from the TME, a **TMEListener** object must be attached to the TME by calling the addTMEListener() method of the **Engine** class.

2.3.1. Implementation Overview

Class Name: `Engine`

Constructor: `public Engine()`

Methods:

```
public void loadModel(Model model)
public boolean validateModel()
public void startSimulation()
public void pauseSimulation()
public void stopSimulation()
public void addTMEListener(TMEListener listener)
```

2.3.2. Methods

`public void loadModel(Model model)`

The loadModel() method simply assigns the given model to the private model field.

This method should however first assure that this Engine instance is not currently running a simulation. Trying to load a model while a simulation is running must produce an exception.

`public boolean validateModel()`

This method ensures that the currently loaded model can be simulated. Validating the model involves the following tasks:

- Checking that every car in the model can reach its corresponding end point.
- Defining the path for every car in the model.

Both tasks involve the use of graph algorithms on the **lane and connector graph**. The requirements for the path finding algorithm are described in the chapter “Path Finding Algorithm”.

If the model has been successfully validated, a boolean field inside the **Engine** class must be set to indicate this.

`public void startSimulation()`

The startSimulation() method is responsible for creating and initializing the **Simulation** object. Once the Simulation object has been initialized, this method will invoke the start() method of the **Simulation** object.

2.4. Simulation

The **Simulation** class is the part of the TME that controls the simulation. The TME creates an instance of this class, if the user requests a simulation.

The most important task of the **Simulation** object is, to keep track of the timing while the simulation is running. This means, that the **Simulation** object must regularly (multiple times per second) call the update() method of all the **Car** and **Intersection** objects inside the model.

Calling the update() method of all **Car** and **Intersection** objects, causes the **Cars** to move (and it causes the intersections to change their traffic lights).

⚠ Even though the **Simulation** object does call the update() methods of the **Car** and **Intersection** objects, the actual update of the internal state of a **Car** or **Intersection** object is done by the object itself. This means, that every **Car** object is responsible for calculating its own next position. The same applies to the **Intersection** objects.

2.4.1. Implementation Overview

Class Name: `Simulation`

Constructor: `public Simulation(Model model)`

Methods:

```
public void start()
public void pause()
public void stop()
public void resetModel()
public SimulationState getState()
public void addEventListener(SimulationEventListener l)
```

2.4.2. Methods

`public void start()`

This method must initialize a timer or a new thread, that regularly calls the update() method of every **Car** and **Intersection** object of the model.

`public SimulationState getState()`

This method must create and return a new **SimulationState** object, that represents the current state of the simulation.

This involves creating a **LaneState** object for every **Lane** of the model and a **CarState** object for every **Car** of the model.

2.5. Car

The **Car** class is the most important part of the Model for the TME. This class represents the **physical state and behavior** of a car for the purpose of simulation.

2.5.1. Implementation Overview

Class Name: `Car`

Constructor: `public Car(R ref)`

Methods:

```
public void update()
.
.
. (not all methods are listed here)
```

2.5.2. Methods

```
public void update()
```

The update() method is the most important part of the **Car** class. This method performs the necessary physical calculations to update all the state properties of the car.

This update process involves checking the lane segment that is ahead of the car for other cars or red traffic lights. The results of this check and the information about the current physical state of the car (location, velocity and acceleration) are then used to determine the next action (accelerate, hold current speed or decelerate). Based on this decision and using the current physical state of the car, the next physical state of the car is calculated.

Calculating the next physical state always requires the use of the current system time, to determine the time that has passed since the last call to update().

3. Algorithms

The following section describes problems that require the use of complex algorithms to solve them.

3.1. Intersection Controller Algorithm

This algorithm will be defined later.

3.2. Path Finding Algorithm

Every car of the model must be able to reach its corresponding end point.

Checking this condition requires an algorithm, that can traverse the **lane and connector graph**. This algorithm must be able to determine, if there is a path between a given start and end point (two **Connector** references).

The same algorithm can then be used to determine the actual path for every car (a sequence of **Connector** references, starting with a start point and ending with an end point).

The algorithm will be used during the validation of the model as part of the validateModel() method of the **Engine** class.

The algorithm does not have to be time efficient, as those calculations are only performed once before the start of the simulation. The algorithm should however return the shortest possible path between two nodes, in cases where multiple paths exist.

4. Further Specifications

4.1. Terminology

4.1.1. Definition of the term "Model"

In the context of the SIMLANE application, the term Model can refer to two different objects.

If the use of the term Model might be ambiguous, we use the two following terms to distinguish the different objects:

- Model of the editor
- Model for the TME

These two objects are **entirely different in a structural as well as a functional way**. The following two chapters describe each term.

The Model of the editor (part of the MVC design pattern)

The term Model can refer to the **Model of the editor**. This is part of the MVC design pattern.

The very first thing that the Controller does, is creating an instance of the **Editor** class and assigning that instance to the private variable called **editor**. This instance of the Editor class is the Model in the MVC design pattern. To avoid ambiguity, we call this instance of the Editor class the **"Model of the editor"**. There is only one such instance in the SIMLANE application. The **Model of the editor** contains the state information of the map and its individual tiles. If the user changes the street network by selecting tiles and then adjusting those tile's lanes, the **Model of the editor** will change to reflect the new street network.

It is important to note, that the **Model of the editor** is made up of a map with individual tiles. However, in the Model for the TME there is no concept of a map or tiles. This is the most important structural difference between the **Model of the editor** and the **Model for the TME**.

The Scenario is another important part of the **Model of the editor**. The map (and its tiles) together with the scenario make up the part of the Model of the editor that gets translated into the Model for the TME in order to simulate the traffic on the user's custom street network. This translation is performed by the Model Factory. The next chapter describes the Model for the TME.

The Model for the TME (representation of the Lanes and Cars for Simulation)

The term Model can also refer to the **Model for the TME**.

The **Model for the TME** gets created by the Model Factory based on the Map and the Scenario, which are both part of the Model of the editor. The **Model for the TME** that gets created by the Model Factory is an instance of the **Model** class. To avoid ambiguity, we call instances of the Model class the **"Model for the TME"**. The Model Factory will create such an instance of the Model class, every time the user clicks on the start simulation button.

The **Model for the TME** is used by the TME to run a simulation of the traffic on the user's street network. The **Model for the TME** contains a graph, that represents the user designed street network and it contains a list of cars which make up the traffic that must be simulated.

The **Model for the TME** does therefore represent the street network and the cars on this street network. You could think of the **Model for the TME** being the "**Model of the street network and cars**" to allow a direct comparison with the "**Model of the editor**".

4.2. Class Specifications

4.2.1. Scenario class

The **Scenario** class is part of the Model of the Editor. It is not known to the TME.

The **Scenario** class represents the start and end points that surround the map on which the user can create his custom street network.

4.2.2. Connector class

The **Connector** class is part of the TME. A Connector object has two purposes:

1. A Connector is used to **link together the different lane segments** in the **lane and connector graph**. The Connector does hold references to the lanes that lead into the Connector (**in**) and it also holds references to the lanes that lead away from it (**out**).
2. A Connector also creates the connection between the **Model for the TME** and the **Model of the Editor**. This means, the Connector object ensures that every lane in the **lane and connector graph** can be mapped onto its representation in the **Model of the Editor**.

Explanation of Purpose 1:

The **lane and connector graph** of the **Model for the TME** contains the lanes that form the street network. In the graph, those lanes are linked to each other using **Connector** objects.

Every **Connector** object holds references to the lanes that lead into it and to the lanes that lead away from it. Those references are stored in the private fields **in** respectively **out**.

The **in** field is of the type Lane[]. The lanes in this array are all from different LaneGroups. The **out** filed is of the type LaneGroup. This is possible, because all lanes that lead away from a Connector (the lanes referred to in the **out** field) belong to the same LaneGroup.

Explanation of Purpose 2:

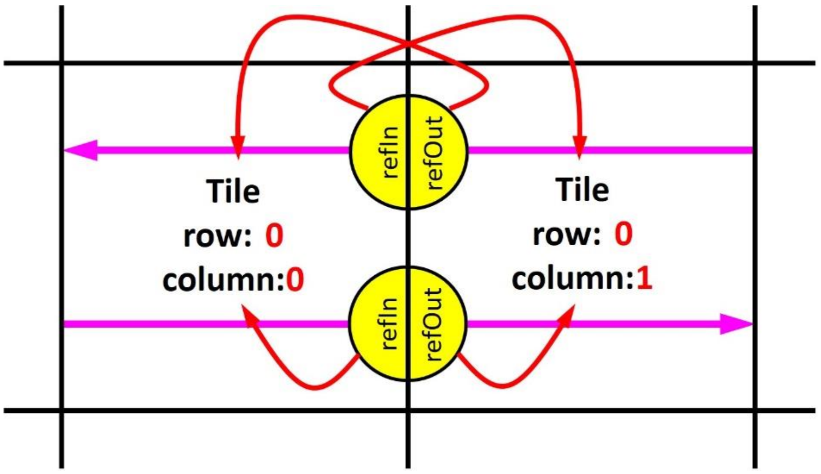
Connectors are initially created during the graph creation process (described in the chapter "Model Factory"), when the **Model of the Editor** is translated into the **Model for the TME**. During this translation process, the information about the map and its individual tiles is lost. The TME does not have a concept of maps or tiles. Instead, the **Model for the TME** only contains lanes that are connected to each other (using Connector objects).

Therefore, it is important to ensure, that at a later point every lane in the **lane and connector graph** can still be mapped onto the corresponding lane on a specific tile of the map.

This is achieved by passing two Tile references to the constructor of the **Connector** object. The first Tile reference points to the Tile where the incoming lanes of the Connector are located (**refIn**). The second reference points to the Tile where the outgoing lanes of the Connector are located (**refOut**).

The picture below illustrates two different Connectors and their values for the **refIn** and **refOut** fields.

Note that for both Connector objects, their location on the Map is unambiguously defined through the two Tile references.



As seen in the picture above, there are always two Connector objects located between two adjacent Tiles. Those Connector objects can still be distinguished because their references to the two Tiles are inverted (note the crossed arrows for the upper Connector).

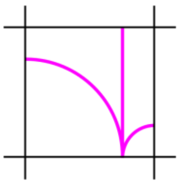
This is possible, because the traffic on a lane can only flow in one direction (cars always drive on the right-hand side of a tile). This is illustrated using the pink arrows in the picture above.

The type of the **refIn** and **refOut** fields is generic. The Connector class is designed that way, to decouple the TME from the rest of the SIMLANE application. If the TME is used in the SIMALNE application, this generic Type will always be **Tile**. Because the Connector objects must refer to tiles.

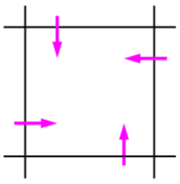
However, the TME might be used in a different software project that does not use tiles. In that case, the Connectors will refer to a different kind of object and the generic type for the **refIn** and **refOut** fields will be set accordingly.

4.2.3. LaneGroup class

The **LaneGroup** class is part of the TME. It is used to sort lanes on a tile into different groups. On every tile, lanes that start on the same side belong to the same LaneGroup. If two lanes start on the same side, then they do start at the exact same point. This is ensured by the restriction, that cars must always drive on the right-hand side of a tile.



The image on the left illustrates a LaneGroup. This LaneGroup contains all of the three possible lanes that start on the same side (on the bottom side of the tile).



For a single tile there can be at most four LaneGroups. One for each side of the tile where lanes might start. There can be fewer LaneGroups if the tile has one or more side from which no lane starts.

Lanes in a LaneGroup are in no particular order.

The grouping of lanes into LaneGroups is important for controlling the traffic lights at intersections. For this purpose, the LaneGroup class contains an additional private field, indicating whether the lanes of the LaneGroup are active (green traffic light) or not (red traffic light). Refer to the chapter about the Intersection class for more details.

4.2.4. Intersection class

The **Intersection** class is part of the TME. It is used to control the traffic with traffic lights, when lanes intersect with each other.

The **Intersection** object contains an array to hold all **LaneGroups** that are part of the intersection. It is important that the **LaneGroups** are added to the array at the proper index (according to the SIDE_INDEX_? constants specified in the Editor class). The side of a **LaneGroup** is defined by the common start side of all lanes in the **LaneGroup** (a **LaneGroup** always contains lanes that start on the same side).

Every Intersection object also contains a boolean array to indicate which **LaneGroups** intersect with each other. The Intersection object must not store the information about which individual lanes intersect with each other. For the control of the traffic on the Intersection it is sufficient to know which **LaneGroups** intersect with each other.

Only lanes from different **LaneGroups** can intersect with each other. Because lanes in the same **LaneGroup** always start on the same side. Lanes that start on the same side can never intersect with each other.

4.2.5. SimulationState class

The **SimulationState** class is part of the TME. This class is used by the TME to represent the current state of a running traffic simulation.

SimulationState objects are parsed by the SimulationStateParser to update the **Model of the Editor** according to the current state of the traffic simulation.

Every **SimulationState** object contains the following fields:

- An array of **LaneState** objects.
  The **LaneState** objects are used to represent the current state (active or inactive) of each lane in the model.
- An array of **CarState** objects.
  The **CarState** objects are used to represent the current state (exact position on a lane) of each car in the model.
- Some additional fields that contain information about the progress of the simulation. This should include at least a time value, to indicate how long the simulation has been running.

4.2.6. LaneState class

The **LaneState** class is part of the TME. This class is used to represent the current state of a single **Lane** in a **SimulationState** object.

The **LaneState** class contains the following fields:

- The ConnectorLocation of the start Connector of the corresponding Lane.
- The ConnectorLocation of the end Connector of the corresponding Lane.
- A boolean field, to indicate whether the lane is active or not.

4.2.7. CarState class

The **CarState** class is part of the TME. This class is used to represent the current state of a single **Car** in a **SimulationState** object.

The **CarState** class contains the following fields:

- The ConnectorLocation of the start Connector of the lane that this car is currently located on.
- The ConnectorLocation of the end Connector of the lane that this car is currently located on.
- A double field to indicate the exact position of the car on its lane.

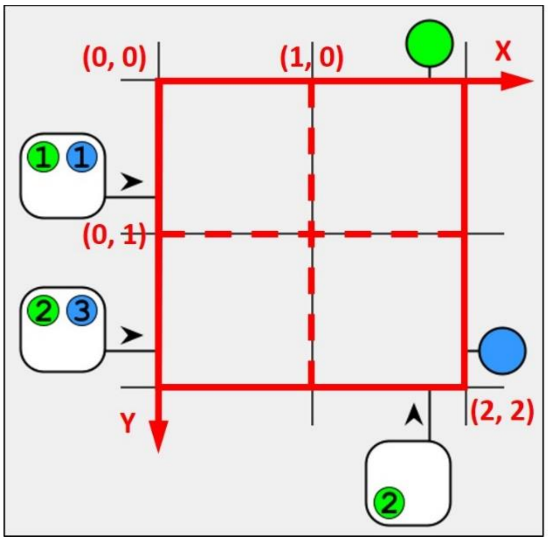
4.2.8. CarIndicator class

The **CarIndicator** class is part of the SIMLANE application. It is not known to the TME. A **CarIndicator** object represents the location of a **Car** on the map. It only contains the following three fields:

- double **x** (the x coordinate of the Car on the Map)
- double **y** (the y coordinate of the Car on the Map)
- CarType **type** (the CarType of the Car, this defines its color)

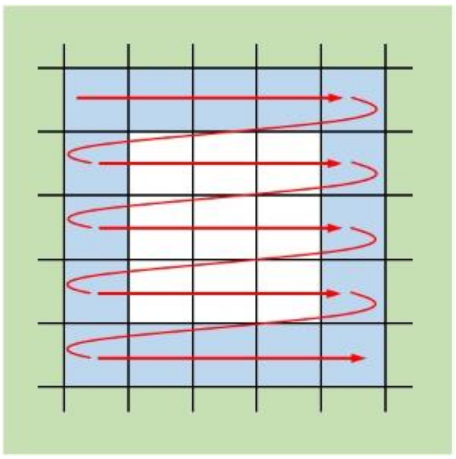
It is not necessary for a **CarIndicator** to unambiguously identify a **Car**. It is sufficient to know the **CarType** of the Car, as the user is unable to distinguish cars that have the same color.

The image below illustrates the coordinate space that is used for the values of **x** and **y**.



4.3. Definition of SIMLANE Specific Concepts

4.3.1. Map Traversal



The tiles of the map should usually be traversed in the order which is indicated by the image on the left.

You should take into consideration that the map may have any rectangular shape, it does not have to be square.

The dimensions of the map (the number of rows and columns) can be obtained using appropriate methods on the Map or Scenario objects.

It is important to note that only the tiles which are shaded **blue** or **white** are **actual map tiles** that **contain lanes**.

The tiles which are shaded **green** in the image on the left are the **tiles that make up the scenario** (scenario tiles). Those tiles cannot contain lanes. Instead they might contain start or end points where cars will enter or leave the custom street network.

Because **the blue map tiles are located on the border of the map**, next to tiles that might contain a start or end point (called scenario tiles, shaded in green), they must often be treated in a special way when traversing the map.

4.3.2. Lane and Connector Graph

The **lane and connector graph** is the main part of the **Model for the TME**. The **Model for the TME** "contains" this graph. The **Model for the TME** is simply an object of the type Model (an instance of this class). It "contains" the **lane and connector graph** by holding references to some of the graph's nodes. Those references are stored in a private field of the **Model for the TME**. This private field is called **startPoints**. It is a list of references to **all the start points** in the **lane and connector graph**. The startPoints are **Connector** objects which have only an outgoing lane (but no incoming lanes). Therefore, the **startPoints** field is of the type **List<Connector>**.

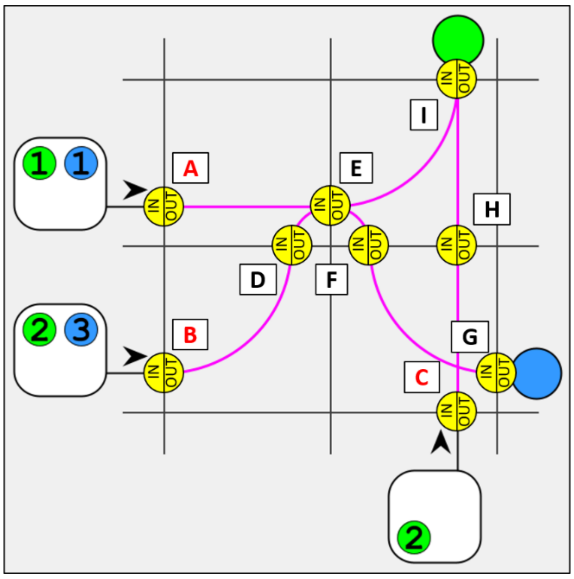
The following section illustrates how the Model "contains" the **lane and connector graph** by holding references to the startPoints of this graph.

If an object must contain a graph, it usually needs just one reference to a single node of this graph (the root). All other nodes of the graph can then be reached by travelling through the graph. This does however require the graph to be "connected", meaning every node must be reachable from every other node through at least one path in the graph.

The **lane and connector graph** is not necessarily connected. Therefore, the Model contains references to **all startPoints** in the graph. This ensures that all nodes of the **lane and connector graph** can be reached by travelling through the graph, starting at one of the startPoints (see the remark on unreachable graph sections below for more details).

As the name suggests, the **lane and connector graph** consists of **Lane** objects and **Connector** objects that hold references to each other, in order to form a graph. In the **lane and connector graph**, every **Lane** is located between two **Connector** objects. The first **Connector** is located at the beginning and the second **Connector** is located at the end of the **Lane**.

The following picture shows a simple user designed street network. On the picture you can also see the **Connector** objects (indicated by yellow dots). Those **Connector** objects will be created when the street network is translated into a **lane and connector graph** by the process described in the “Model Factory” chapter.

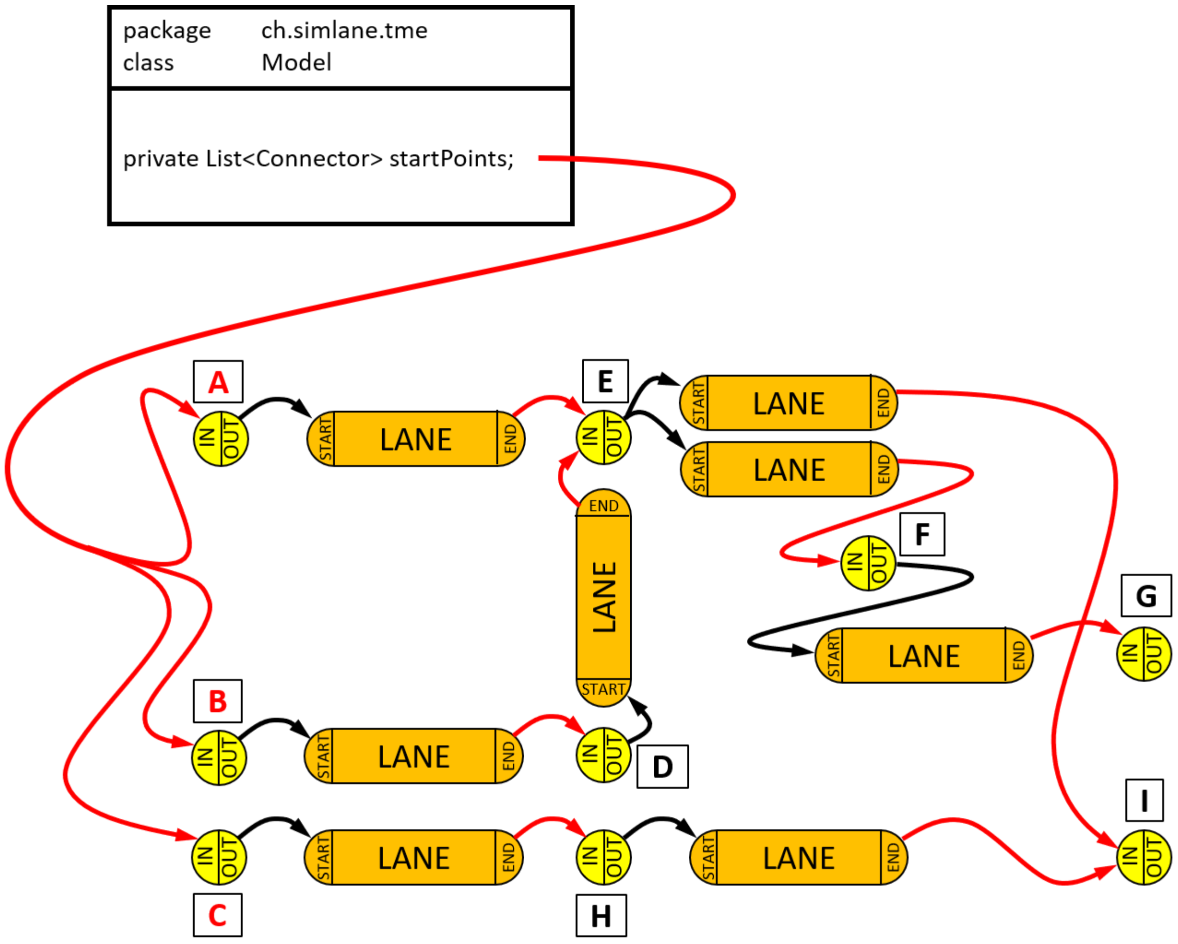


The picture shows, that the lane and connector graph consists of alternating **Connector** and **Lane** objects. Every **Connector** is followed by a **Lane** and vice versa.

The **startPoints** (labeled **A**, **B**, **C**), are **Connector** objects, that only reference an outgoing lane.

Consequently, the **endPoints** (labeled **G**, **I**) are **Connector** objects, that only reference an incoming lane. The **endPoints** however, do not play a significant role in the graph creation process. They are also not directly referenced by the Model, as they are reachable from one of the **startPoints** by travelling through the graph.

The following image shows the structure of the **lane and connector graph** that results from translating the above street network using the process described in the chapter “Model Factory”. Please note the explanations below.



```
package     ch.simlane.tme
class       Model

private List<Connector> startPoints;
```

The yellow circles represent Connector objects. The orange shapes represent Lane objects.

It is important to note the following:

- Red arrows indicate references to Connector objects.
  The image above shows the red arrows pointing to the **in** field of the Connectors. Those references do however point to the Connector objects themselves and not to one of its fields.
- Black arrows indicate references to LaneGroup objects.
  The image above shows the black arrows pointing to Lane objects instead of LaneGroup objects. The LaneGroup objects are not shown.

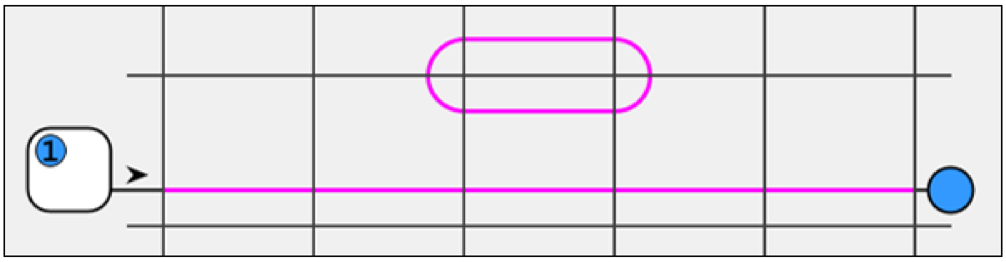
The image above is a simplification of the actual graph and does not show all objects and references. The following aspects of the **lane and connector graph** are not shown in the image:

- LaneGroup objects are not shown. Instead, the picture shows the Lane objects that are contained by the LaneGroup objects.
- In addition to the references shown on the picture, the following references exist:
  - Every Connector object holds references that point back to the lanes that lead into it (not only to the lanes that lead away from it).
  - Every Lane object holds a reference that points back to the Connector that it started on.

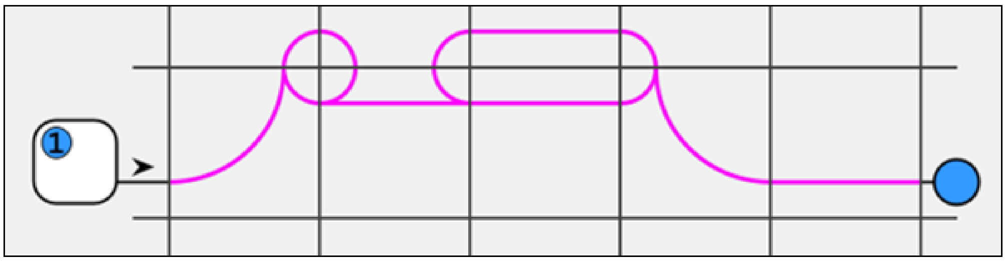
Remark on unreachable graph sections

A user might design a street network, that has unreachable lanes. Such a street network is valid, as it has no unconnected lanes. However, during the graph creation process described in the “Model Factory” chapter, those unreachable lanes will be lost. This behavior is intended, as the unreachable sections of the map are not relevant for the simulation.

The following picture illustrates a valid street network with an unreachable section of lanes:



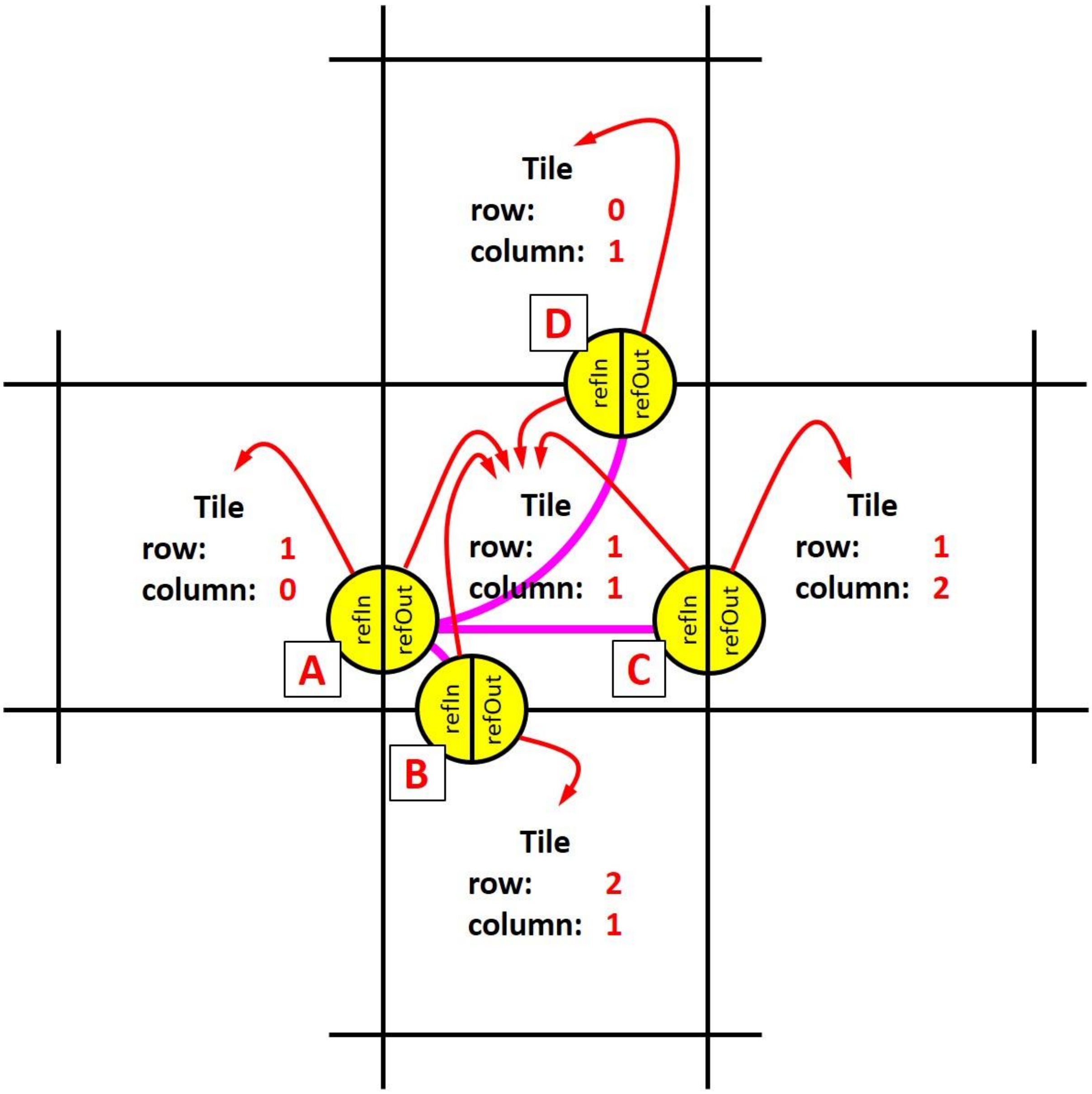
On the map, lanes of an unconnected section can still appear connected (illustrated below). And it is also possible that an **endPoint** of the map is not reachable (illustrated below), which will be detected during model validation at a later point. Both cases do not have to be considered in a special way. They are mentioned here only to help understand the different situations that can occur during the graph creation process.



4.3.3. Connector, LaneGroup and Lane references

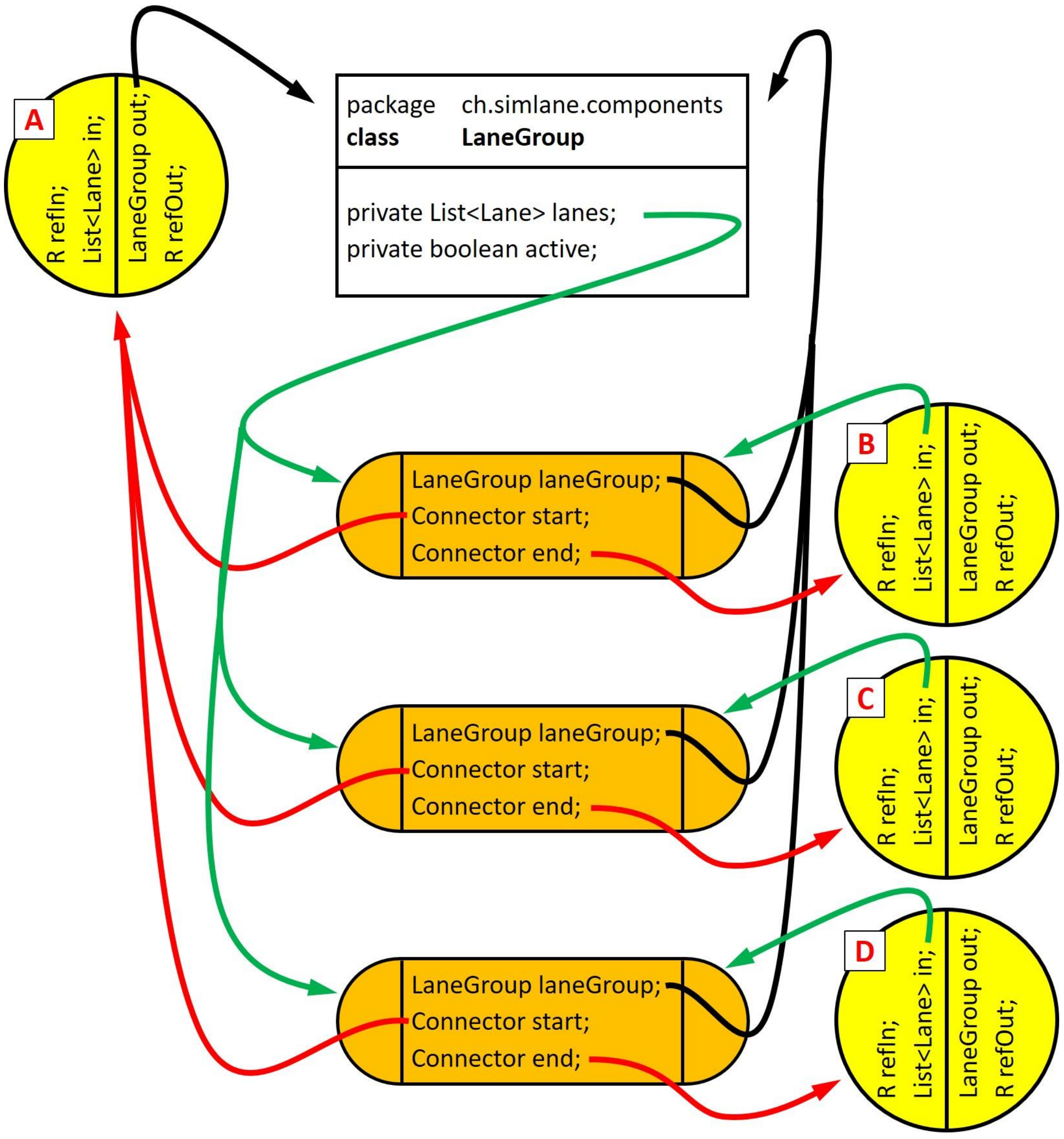
In the **lane and connector graph**, there are various references between the following three types of objects: **Connector**, **LaneGroup** and **Lane**. Each **Connector** objects holds additional references to two tiles (**refIn** and **refOut**). This chapter explains all those references and it illustrate the connections between the three objects mentioned above.

The following picture shows a tile containing three lanes. The three lanes all start at the same point and lead to a different side of the tile. In the picture you can also see four different **Connector** objects (labeled **A**, **B**, **C**, **D**). For each **Connector** object, the values of its **refIn** and **refOut** fields are indicated using red arrows that point to the referenced tile. Note how the location of each connector is unambiguously defined using its **refIn** and **refOut** fields.



The picture on the next page shows the same **Connectors** (labeled **A**, **B**, **C**, **D**) again, but with more details. In addition to the **Connectors**, the picture below also shows the **LaneGroup** and the **Lane** objects for the situation that is illustrated above.

The references between the different objects shown below are indicated using arrows of different colors. Red arrows represent references to **Connector** objects, black arrows represent references to **LaneGroup** objects and green arrows represent references to **Lane** objects.



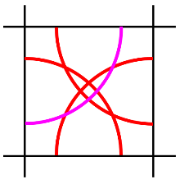
In the image above, **Lane** objects are illustrated using orange shapes. The yellow circles indicate **Connector** objects.

4.3.4. lanesIntersect() method

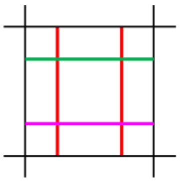
The lanesIntersect() method of the class Tile must meet the following specifications:

The static methods in the Editor class can be used to determine the following: Is the first lane a turn or a straight lane? If it is a turn, determine if it’s a left (big) or a right (small) turn.

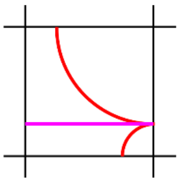
Depending on the result, return true if the second lane intersects with the first lane in one of the following ways:



If the first lane is a left (big) turn, it intersects with the second lane, if the second lane is also a left (big) turn.



If the first lane is a straight lane, it intersects with the second lane, if the second lane is a straight lane other than the one that goes in the opposite direction.



The method must also return true if the first lane has the same end point as the second lane. This is because two lanes always intersect with each other if they have the same end point.

The lanesIntersect() method must return false, if none of the above conditions were met. In this case, the two lanes do not intersect with each other.

An exception must be thrown if the two given lanes are identical.

4.3.5. Intersecting lanes check and Intersection creation

If two lanes on a single tile intersect with each other, an **Intersection** object must be created during the creation of the **Model for the TME**. The **Intersection** object ensures that the traffic on a tile with intersecting lanes can be controlled using traffic lights during the simulation.

In order to check, if an **Intersection** object must be created, the method lanesIntersect() of the class Tile can be used. This method can be called with two lanes (specified using side constants) to determine if the given lanes intersect with each other.

Use the **laneState** array of the current tile and check every lane against every other lane to find intersecting lanes (this should be made more efficient by avoiding unnecessary checks). If two lanes on the tile do intersect with each other, an **Intersection** object must be created.

After creating the **Intersection** object, all **LaneGroups** must be added to it as described in the chapter “Intersection class”.

The **Intersection** object was only created because two intersecting lanes have been found. The lane groups of those two intersecting lanes must now be marked in the **intersectingLaneGroups** array of the **Intersection** object.

After marking the **LaneGroups** in the array, the search for intersecting lanes on the tile (as described above) must be continued. If the search has finished and if all intersecting **LaneGroups** have been marked in the **intersectingLaneGroups** array, the Intersection object is fully initialized.

The next step is to add this **Intersection** object to the model (the **Model for the TME**) by adding it to the list of Intersections.

4.3.6. StartPoint and EndPoint HashMaps

In order to retrieve the Connectors for the start and end points when creating the cars, we need to add them to HashMaps with appropriate keys. The two required HashMaps are described below.

The Connectors for the start points must be added to a HashMap of the type HashMap<StartPoint, Connector>, we call it **startPointConnectors**. For each Connector, the corresponding StartPoint object must be used as a key.

The Connectors for the end points must also be added to an additional HashMap. It is of the type HashMap<CarType, Connector> and we call it **endPointConnectors**. As the type of the HashMap indicates, when adding a Connector, we must use the corresponding end point's **carType** as a key. This allows us to retrieve the Connector of an end point based on the end point's CarType.

The **startPointConnectors** and **endPointConnectors** HashMaps must both be created and filled with their Connectors, when the Connectors are first created. The HashMaps will then be used in a later step, for the creation of the cars.

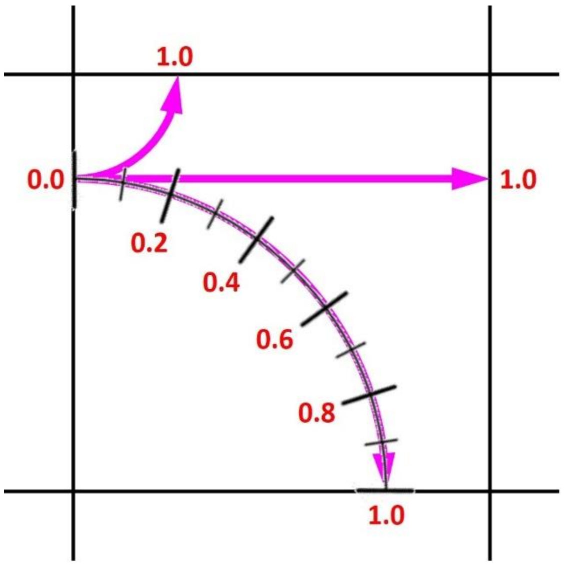
4.3.7. Position Values for Cars on Lanes

The current position of a car is described using two values:

- The lane that the car is currently located on.
- A double value called **pos**. This value describes the position of the car on its lane.

The **pos** value is always in the range from 0 to 1. A **pos** value of 0 indicates, that the car is located at the beginning of the lane. A **pos** value of 1 indicates, that the car is located at the end of the lane. Using values between 0 and 1, every possible location of the car on the lane can be described. For example, a **pos** value of 0.5 means that the car is in the middle of the lane.

If the lane is not a straight lane, additional calculations must be performed to determine the actual position of the car. This is illustrated in the picture below.



Those additional calculations will be performed by the parse() method of the **SimulationStateParser** class.