Självständigt arbete på grundnivå

Independent degree project – first cycle

Datateknik
Computer engineering

Line Graph Utility
A software module for routing

Jonas Bergman

Mittuniversitetet
MID SWEDEN UNIVERSITY

Campus Härnösand Universitetsbacken 1, SE-871 88. **Campus Sundsvall** Holmgatan 10, SE-851 70 Sundsvall.
Campus Östersund Kunskapens väg 8, SE-831 25 Östersund.
Phone: +46 (0)771 97 50 00, Fax: +46 (0)771 97 50 01.

MID SWEDEN UNIVERSITY
DSV

Examiner: Ulf Jennehag, ulf.jennehag@miun.se
Supervisor: Börje Hansson, borje.hansson@miun.se
Author: Jonas Bergman, jobe0900@student.miun.se
Degree programme: Programvaruteknik, 180 credits
Main field of study: Software technology
Semester, year: Fall, 2015

Abstract

This project was about building a line graph utility, a software module that should read map data from a PostGIS database and transform that information into a line graph (edge based graph) that the calling software could use to perform routing decisions. This outer calling application is part of a project (by an anonymized company) for flexible public transportation, that is meant to manage and direct a fleet of vehicles to where the customers actually are, instead of idling at bus stops. The software module should take different kinds of restrictions and conditions into account when building the line graph, to reflect the actual traffic situation. That can be turn restrictions, traffic signs, inclination, or conditions such as temporary hindrances, time of day. Some are static, but others vary dynamically and the state is to be found in the database.

This study has found a set of tools that aids in the transformation of OpenStreetMap data into a PostGIS database; for building the topology of the map; querying the database; and data structures for representing the graph and line graph.

The result of the project is a piece of working software that can return a line graph as a Boost graph with some restrictions taken into account, but it has not yet implemented them all, and more specifically, it does not handle conditional restrictions yet. There remains a good deal of work to implement all that complex logic.

Contents

1 Introduction

The work presented in this thesis is about flexible routing of public transportation. The result of the work is a software module that loads map data and converts it into in-memory data structures that can be used for routing decisions by exposing an *API*, (Application Programming Interface). This module is part of a bigger transportation optimization system that is meant to enable flexible public transportation solutions.

The module will be used for finding efficient routes in a dynamic traffic environment, i.e. the complete solution must take turn restriction, traffic lights and road signs into account. The outline of how to do this is by loading map data from *OpenStreetMap*[1] into a database (*PostgreSQL*[2] extended with *PostGIS*[3]). Upon a request directed to the API, the module should build (with soft real-time requirements) a data structure suitable for passing to a routing algorithm.

1.1 Background and problem motivation

The company (*anonymized*) aims at developing a solution for managing *flexible* public transportation, meaning no more buses standing idle and empty at bus stops, waiting just in case another bus fills up. The buses can be directed to where they are needed, and part of the solution is finding the best routes and give directions to the drivers where they should go. The public does not need to wait at bus stops, but can ask for pick-up via a mobile app.

There can obviously be huge benefits from such a transportation system. Less vehicles are needed, and better utilization of the vehicles, which should be good both for the environment and the finances of the operation. The public should also benefit from having access to public transportation where needed, and not from fixed locations.

Central for such a system is efficient routing of the vehicles, with almost instant updates on restrictions made available to the drivers needing directions. This project is a small piece in that puzzle.

1.2 Overall aim

This project should result in a working software module, fulfilling the requirements set by the company. There is needed some preliminary studying of graph theory, data structures, and research into what theories and solutions that already might exist, and if so, if they can be adapted and used in this project.

1.3 Scope

The scope of this project is to create the routing data structures representing the map data, not the routing algorithms, although they might affect one another, such that the choice of algorithm might affect what data structures are suitable.

1.4 Detailed problem statement

The software in this project is a module, exposing a function. When the function is called, it should load map data from a database, which has previously been loaded with OpenStreetMap-data, and build a connected graph to be used for routing decisions, and the data structure is returned to the caller so it can be used for routing. The building of the graph should happen in *soft real-time* so that it reflects all known restrictions in the database. For example if one

[1] http://www.openstreetmap.org
[2] http://www.postgresql.org
[3] http://postgis.net

road gets temporarily closed it should be marked as such, and that should be represented in the graph.

The requirements from the company states that the graph should be represented as a *line graph*, which is a basic technique for representing available turns at junctions. The software module shall be implemented in C++, using the *Boost Graph Library*[4] for the data structures. The software should be developed using *Behavior Driven Development* (BDD) or *Test Driven Development* (TDD) as methodologies, and otherwise adhere to the company's coding standards.

1.5 Outline

- Chapter 2 will present some background on graph theory, and research in map routing, regarding both theoretical foundations and some available implementations.
- Chapter 3 shows the methods and tools used.
- Chapter 4 is about the design and implementation of the software module.
- Chapter 5 presents the results from testing the implementation.
- Chapter 6 will include some discussion and conclusions made during this project.

1.6 Contributions

The work presented in this report is the sole work of the author.

[4] http://www.boost.org/doc/libs/1_54_0/libs/graph/doc/index.html

2 Related work

One of the first applications of *graph theory* was when *Leonhard Euler* considered the *Königsberg bridge problem*: Is there a way to walk over the seven bridges of Königsberg only once?

It is trivial to see that there is a close correlation between graphs and maps, as you can see in figure 2.1, with the roads and junctions in the map being lines and dots in the graph. A line is mostly called *edge* or *arc* and a connecting dot is called *vertex*, *node* or *point*; in this report it will mostly be *vertex* and *edge*, but to differentiate, another type of graphs called *line graph* will use the names *node* and *line* to distinguish. As one delves deeper into the theoretical material, one will find that there is good to know some *graph theory* and be familiar with some definitions.

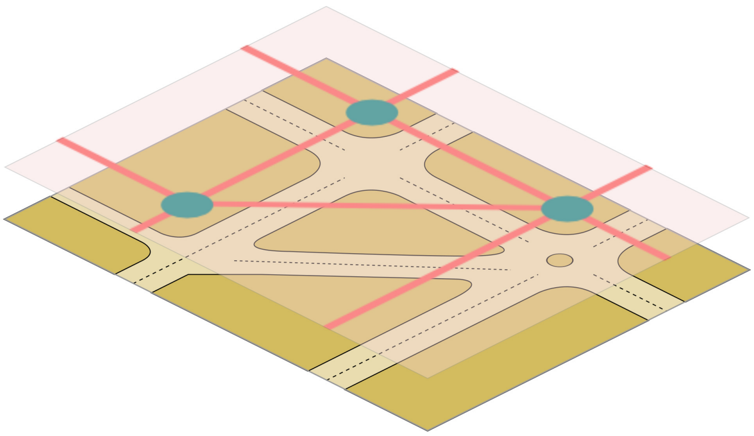
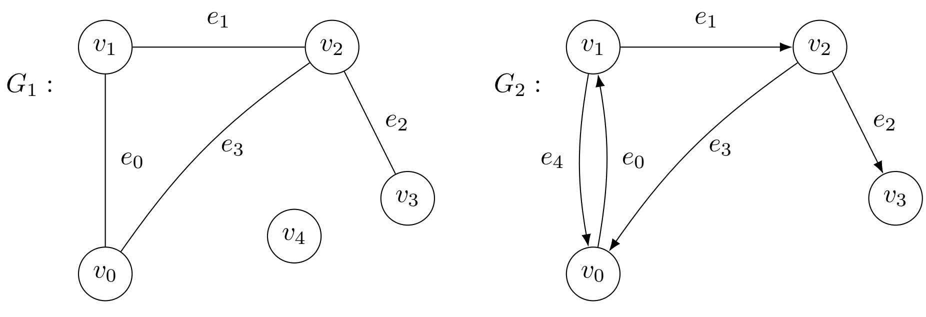


Figure 2.1: Graphs and road maps are a natural match.

2.1 Graph theory

There are good lecture notes such those from *Tampere University of Technology* [1] and *Univerity of Turku* [2] (both happen to be Finnish) and a good text book by *Reinhard Diestel* [3] are available to get into this subject. One does not need to understand all the concepts, but be familiar with some basic definitions and notations.

A *graph* is made up by *vertices* and *edges*, see figure 2.2.



(a) Undirected (b) Directed

Figure 2.2: A *graph* with *vertices* and *edges*.

So a graph G is a pair of sets, $G = (V, E)$ where $V = \{v_0, ...\}$ is the set of vertices and $E = \{e_0, ...\}$ is the set of edges. The edges can have their own labels as in the figure, or they can be denoted by the pair of vertices they connect: e_0 could be also named as (v_0, v_1) or v_0v_1. A graph can be *undirected* (figure 2.2a) if the edges have no sense of direction, or it can be *directed* (figure 2.2b) if the direction of travel along an edge matters; e_0 is distinct from e_4 because the have different directions although they connect the same nodes v_0 and v_1. A directed graph can also be called a *digraph*.

To decide how "big" a graph is, one can count the number of vertices, $|V|$, to get the *order* or *cardinality* of the graph. If one counts the number of edges, $|E|$ one gets the *size* of the graph. In figure 2.2, G_1 has order 5, and size 4; and G_2 has order 4 and size 5.

Edges are *adjacent* if they share a common vertex, and vertices are adjacent if they are connected by an edge, one can also say that v is *incident* with e. In figure 2.2a, v_0 and v_1 are adjacent but not v_1 and v_3, and e_0 and e_1 are adjacent but not e_0 and e_2.

The number of edges connecting to a vertex is called the *degree* of the vertex, $d(v)$. In figure 2.2a, $d(v_2) = 3$, $d(v_3) = 1$ and $d(v_4) = 0$. A vertex of degree 1 is called a *pendant* vertex, or *leaf*, and a vertex of degree 0 is called *isolated*. If all *components* of a graph are *connected*, then the graph is a connected graph. In figure 2.2 graph G_2 is connected, but graph G_1 is not because it has an isolated vertex as a component.

A graph is *planar* if it is possible to draw without edges crossing each other. It is *Eulerian* if one can travel over every edge in the graph only once (as in the *Königsberg bridge problem*). A graph is called *Hamiltonian* if one can visit every vertex in the graph only once (as in the *Travelling salesman problem*).

Travels in graphs can be called different names. Ruohonen [1] has the most general name *walk* for travel from vertex to vertex along edges. A walk is *open* if it ends on a different vertex than it started, or *closed* if it ends on the same vertex. If an edge is traversed only once, the walk is called a *trail*. If any vertex is visited only once then the trail is a *path*. If the walk is a path but with the start and ending vertices being the same, then the walk is a *circuit*.

One can partition a graph into *subgraphs* if on places a cut in a vertex (*cut vertex*) or over a set of edges (*cut set*). In figure 2.2a a cut vertex could be v_2 and a cut set could be $\{e_1, e_3\}$.

2.1.1 Graph representation

There are different ways of representing graphs. We have so far used

- Graph diagram
- Set definitions, $V(G) = \{v_0, v_1, v_2, v_3, ...\}, \quad E(G) = \{e_0, e_1, e_2, e_3, ...\}$

One can also use

- *Adjacency matrix*
- *Incidence matrix*
- *Adjacency list*

An *adjacency matrix* is a matrix that shows if vertices are adjacent or not. A value of 0 indicates that the vertices are not adjacent. For an *unweighted* graph, adjacency can be indicated with a 1, or if it is a *weighted graph*, it can be the value of the weight (e.g. edge length or cost). From figure 2.2a:

$$D = \begin{array}{c} \\ v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{ccccc} v_0 & v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \tag{2.1}$$

An *incidence matrix* describes which vertices that are incident with which edges. In a directed graph it is *positive* if it is the *start* vertex of the edge, or *negative* if it is the *ending* vertex of the edge. From figure 2.2b:

$$A = \begin{array}{c c} & \begin{array}{ccccc} e_0 & e_1 & e_2 & e_3 & e_4 \end{array} \\ \begin{array}{c} v_0 \\ v_1 \\ v_2 \\ v_3 \end{array} & \begin{pmatrix} 1 & 0 & 0 & -1 & -1 \\ -1 & 1 & 0 & 0 & 1 \\ 0 & -1 & 1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{pmatrix} \end{array} \tag{2.2}$$

A *dense* graph has almost all vertices connected to each other, i.e. there are few 0s in the adjacency matrix. In a *sparse* graph there are a lot fewer edges than there could be, so the adjacency matrix has a lot of 0s. To be space efficient, especially in computing, it can therefore be better to represent a graph as an *adjacency list*, which simply lists for each vertex which other vertices it is adjacent to. No 0s needs to be included. From figure 2.2a:

$$v_0 : (v_1, v_2), \quad v_1 : (v_0, v_2), \quad v_2 : (v_0, v_1, v_3), \quad v_3 : (v_2) \tag{2.3}$$

2.2 Map routing

For graphs as those described above, there exists basic algorithms such as *Dijkstra* and *bidirectional search*, or more goal directed such as *A**, that tries to find the shortest path from vertex s (source) to vertex t (target). To do that, each edge needs to be associated with a length. That is, the *metric* is *distance*.

However, when it comes to map routing there can be other metrics that are more important than the shortest path. For example *time* (we want the shortest driving time); *road category* or *land use* (we don't want to route through a residential area with low speed limits, or avoid having to go by ferry); *turn cost* (turning slows driving down so prefer straight routes); *multimodal* (when going by public transport we want to minimize waiting and the number of exchanges); *via* (we want to travel via a specific road or city); and so on.

A really basic ingredient in map routing is of course also the fact that roads are directed, i.e. there can be one-way roads. It is also important to take into account that there can be turn restrictions, so that a turn is not allowed at a junction, although it looks like it on the map (and the graph). Even more complicating is the fact that different restrictions on roads might be permanent, or just temporary due to road work, accidents, etc, so there is a difference between *static routing*, where the metric costs are static, and *dynamic* routing where the costs fluctuate over time.

2.2.1 Overview

In an overview of route planning techniques from 2009 [4], it is stated that the starting point for a "horse race" in developing speed-up techniques started in 2005 (p.124), when continental sized road networks of Europe and USA were made publicly available. Before that, large map data had been proprietary and it was hard to compare different approaches. The last decade since then has seen a quick development in the area, so a new overview in a tech report in 2014 from researchers in German universities and Microsoft [5] stated that the previous report was now outdated. This last report is a great overview of route planning techniques from the basic *Dijkstra*, continuing to different families of techniques: *goal directed*, *separator based*, *hierarchical*, *bounded hop*. The report also describes combinations of different techniques and notes on *path retrieval* (getting a description of the shortest path, no just the cost), *dynamic networks* and *time dependence*.

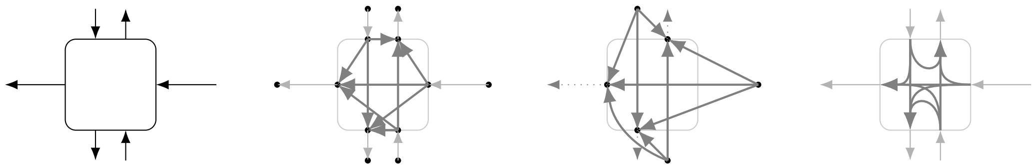
The motivation for the speed-up techniques is to enable "instant" route planning in large networks. The Dijkstra algorithm might need some seconds to complete a query, while one with some preprocessing might be able to perform a query in milli- or even microseconds. This is done by dividing the work into two distinct phases: the *preprocessing* phase, and the *query* phase. The preprocessing phase takes the original graph and performs transformations and builds new data structures. This is a process that can take a lot of time, from seconds to hours and even days depending on algorithm, and the data the size of the data structures might multiply several times. The gain is that the query phase executes almost instantly.

A lot of the research has been conducted on simple models without turn restrictions, so it is easy to compare the speed gain to Dijkstra's algorithm, and one have thought that adding turn costs or restriction on top will not be so hard. However, it turns out that most algorithms with large gains in speed are quite inflexible and have trouble to incorporate changing restrictions and metrics without the need for running the preprocessing phase again [6, p.2]. A more flexible way would be to have a separation of *topology*, i.e. how the graph "looks" with vertices and edges, from the *metrics*, i.e. the cost for travel in the graph.

Those techniques with preprocessing can be characterized as *offline* techniques, while techniques that perform all processing in the query phase can be called *online*. As said before, a lot of the research has been done on continental scale maps. But if one restricts one self to a metropolitan map with a graph of a smaller size, then perhaps the queries perform fast enough without preprocessing, or the preprocessing phase is so fast that can be run online?

2.2.2 Map representation

To have a real-world application that performs route planning, it also needs to seriously take turn restrictions into account, and to be more useful also be able to handle *turn costs*. There exists several techniques for that, see figure 2.3. The most straight-forward technique might be to introduce some new vertices so that edges have a *head* and a *tail* vertex, and turns are modeled by connecting head and tail vertices; this is called a *full-blown* representation (figure 2.3b). One of the most used representations is by converting the original *vertex-based graph* to an *edge-based graph* (also called *arc-based graph* or *line graph*). It can be viewed as connecting tails to tails, see figure 2.3c. These techniques introduces several new edges and vertices, inflating the space needed for the data structures. A more compact representation is keeping a table for each junction with the associated turn and turn costs, see figure 2.3d.



(a) None (b) Full (c) Edge-based (d) Turn table

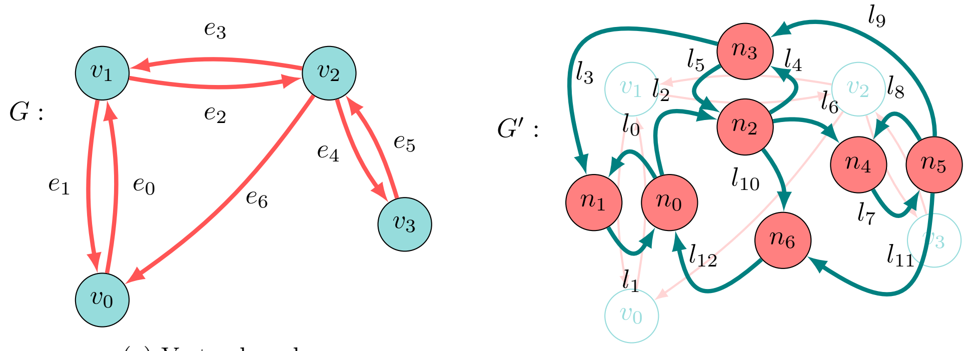
Figure 2.3: A closer look at a junction with two bidirectional and two unidirectional roads with different turn representations. *(After [6, p.8].)*

Edge-based graph

An *edge-based/arc-based/line* graph is a pretty straight forward transformation, where the edges of the original graph is turned into vertices in the transformed graph, and two vertices in the transformed graph is connected by an edge if a turn is allowed in the original graph. To make it simpler to distinguish beteween the vertex-based original graph and the new edge-based graph, we can call the new vertices *nodes*, and the new edges *lines*, i.e. "$road = node$", with nodes connected by lines, if a travel is allowed. This gives us a graph $G' = G_{edge-based} = (N, L)$ where N is the set of *nodes*, $N = \{n_0, n_1, ...\}, N = E$ and L is the set of *lines*, $L = \{l_0, l_1, ...\}$ connecting the nodes, see figure 2.4.

As one can see, the complexity and size of the graph grows in the transformation, what was $|V| = 4$ vertices and $|E| = 7$ edges became $|N| = 7$ nodes and $|L| = 13$ lines. The increase in size of the data structures is one drawback with this simple transformation, but on the positive side is the fact that one can apply ordinary algorithms such as Dijkstra to the edge-based graph just as easily as on the original graph. Another disadvantage might be that it lets the topology represent metrics, i.e. a turn restriction is hard coded into the topology, so how does one handle temporary restrictions?

Volker [7] has written a study on "Route Planning with Turn Costs" and uses edge-based



(a) Vertex-based

(b) Edge-based

Figure 2.4: Transformation of a graph G to an *edge-based* graph G'.

graphs as the foundation. He also introduces an *interface graph* as a link between a vertex-based and an edge-based graph. This new graph builds on an elaboration on what a turn and a junction actually is, with an incoming and an outgoing edge being adjacent on the junction vertex. One can thus see a turn T as either $T = (e_{in}, e_{out})$, i.e. an incoming and an outgoing edge, or as $T = (s, t, u)$ as going from vertex s to vertex u, *via* vertex t. This kind of finer look at what a turn is, is also used by others to build a more compact representation, see below, section 2.2.2.

Turn tables

One way of dealing with a more compact representation of turns and restrictions and costs associated with them is presented in “Efficient Routing in Road Networks with Turn Costs” [8], where they use a standard vertex-based graph (where roads are edges and junctions are vertices) on which they use speed-up techniques such as *Contraction Hierarchies*, but they associate each junction with a table that describes the costs of turns there, essentially describing all possible turns and restrictions. It turns out that a lot of junctions in a road map share the same characteristics and therefore can share the same *turn table*, so that on a map over Europe on average 18 vertices could share the same turn table. Thereby they managed to reduce preprocessing time by a factor 3.4 and space by a factor 2.4 with the same query times.

Another solution which uses the vertex-based graph with a turn table at the junctions is described in “Customizable Route Planning in Road Networks” [6, p.6]. Their solution uses a *separator-based* speed-up technique, which has been viewed as slower than hierarchical techniques, but they argue that this is the most flexible solution with a clean separation of *topology* and *metrics*, with two preprocessing stages before the query stage; one slow *metric-independent* that works on the topology, and a faster *metric customization* stage that can be run for each metric (takes about a second). This solution also uses the fact that many junctions share the same characteristics and therefore turn tables can be shared.

Bidirectional edges

So far we have thought of the directed original graph as having an edge for each direction of travel between two vertices, as with edges e_0 and e_1 in figure 2.4a. But this can be more compactly represented with one edge having a couple of flags indicating directions, meaning that for most roads we need not have two complete edge structures, but only one with two extra bits indicating the direction. This is however not something that can be done in an edge-based graph as the *lines* in it are not bidirectional.

2.3 Map data

One needs to have good map data as the source for building these data structures and apply smart algorithms on. With poor data it does not matter if one has smart algorithms. A said earlier, a race in route planning started with the public release of previously proprietary data. About the same time *OpenStreetMap* also began, which is an *crowd-sourced* project, meaning that anyone interested can be a cartographer and contribute with map data. Over the years the project has grown to an impressive size, and is used as the base for many applications. However, it turns out that the map data actually lacks a lot of turn restrictions. They might be hard to enter, and they are impossible to spot when comparing aerial photos with maps. Efentakis et al. states that for Athens with 277 thousand vertices only 214 restrictions were entered. They propose an automated remedy by comparing GPS-traces to the maps and deducing that turns seldom made are actually banned and could be marked as restricted in the map data [9].

All the same, a lot of high quality applications exists built on *OSM* (OpenStreetMap) data and this project aims at that to.

2.3.1 Projections

Generally speaking, the globe is spherical, but a map is flat. That means that one somehow needs to *project* the spherical data on a flat surface. There exists a lot of different *projections* that tries to do it best. To keep track of which projection one is working in, one can identify it by its *SRID – Spatial Reference System Identifier* that uniquely identifies which projection and which kind of coordinate system one works with.

2.3.2 Topology

The *topology* is about the relationship between objects in a map [10]. If one only thinks of a map as a collection of lines, it is hard to make something out of that information. It becomes useful when we understand the topology, that "this line is connected to that line at this point". The we have a relationship between the lines and can understand how to travel on the map.

Analyzing the topology also makes it possible to correct errors made while adding items to the map data, such as if two lines don't actually meet. Then there is a gap and there is no connection. When analyzing, one can opt to connect lines that are within a small distance of one another, thereby correcting mapping errors.

2.4 Available applications

Some of the research described earlier in this chapter is used actual working applications. For example *CRP (Customizable Route Planning)* [6] is used in Bing Maps, and *CH (Contraction Hierarchies)* are used in for example GraphHopper[1] and OSRM[2], and they are open source routing applications. So the source code is available so one can study how the data structures are implemented and how the algorithms work. There exists a lot of other solutions built on OSM as well[3], using Contraction Hierarchies or other speed-up techniques.

2.5 Memory or database

All research referred to so far has been about building data structures to be held *in memory* so algorithms can operate on them. But as we speak of *queries*, on might think that databases and query languages might be useful as well. There is some research, and a technique called *HLDB* is interesting [11], [12]. It is fast enough, and very flexible, permitting to query for *alternative routes* and *points of interest*.

[1] https://graphhopper.com/
[2] http://project-osrm.org/
[3] http://wiki.openstreetmap.org/wiki/Applications_of_OpenStreetMap

pgRouting is an open source database extension to *PostgreSQL*, often used for holding OpenStreetMap-data. It has a function called `pgr_trsp`[4] that looks for the shortest path with turn restrictions, so obviously standard relational databases can be part of a solution.

[4] http://docs.pgrouting.org/2.0/en/src/trsp/doc/index.html

3 Methodology

The overall methodology for the development is a combination a of *Behaviour Driven Development, BDD* and *Test Driven Development, TDD*, meaning all features of the module have either a *scenario* (BDD) or a *test case* (TDD) written.

As for tools, some are given in the specifications (see Appendix A), while others have been chosen during a selection process. Below is presented the tools chosen, and in some cases what alternatives that were also considered and tested. The categories are:

- Behaviour and Test Driven Development.
- Database.
  - Database and extensions.
  - Loading OpenStreetMap into database.
  - Build topology.
  - Examining map data.
  - Connecting to database from application.
- Reading configurations from `json`-file.
- Building graph.

3.1 Behaviour and Test Driven Development

Behaviour Driven Development tests usually have the structure: *Scenario* → *Given* → *When* → *Then*, written with words to describe the steps. An example in the *Gherkin* language is shown in listing 3.1.

```
Scenario: vectors can be sized and resized
       Given: A vector with some items
       When:  the size is increased
       Then:  the size and capacity change
```

Listing 3.1: Example of a BDD scenario in *Gherkin*.

So when developing BDD style one has to think through different scenarios and write them down, which can be helpful when thinking about what one tries to accomplish.

3.1.1 Tools, installation and usage

The testing library for this project is Catch[1], which is a small library for both BDD and TDD, where the BDD *"scenario"* corresponds to a TDD *"test-case"*, and *"given", "when", "then"* corresponds to *"section"*, meaning one can choose the development style one wishes. *Catch* was chosen because it is header only, and there is no need for complicated building of libraries and setting up paths; one can just include the header in the project and go.

Simply download the file `catch.hpp` and put it either in your project tree or in your path for includes.

Include the header in the source for your test, and get Catch to provide a `main`-method. See listing 3.2 for an example of how to implement the above stated "feature".

[1] `http://www.catch-lib.net`

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include <vector>

SCENARIO ("Vectors can be sized and resized", "[vector]") {
    GIVEN ("A vector with some items") {
        std::vector<int> v(5);

        REQUIRE (v.size() == 5);
        REQUIRE (v.capacity() >= 5);

        WHEN ("the size is increased") {
            v.resize(10);

            THEN ("the size and capacity change") {
                REQUIRE (v.size() == 10);
                REQUIRE (v.capacity() >= 10);
            }
        }
    }
}
```

Listing 3.2: A basic BDD scenario with Catch

3.1.2 Alternatives

The BDD style of developing seems not to have caught on in c++ so much. There are a few libraries. Cucumber-Cpp[2] was investigated as it is an implementation for c++ of the Cucumber tool, which is widespread in many programming languages, so one could write the test for features in the ordinary `.feature`-files in the *Gherkin* language, that are common for writing features for tests. But I could not get Cucumber-Cpp to build correctly with CMake and the dependencies.

3.1.3 Remarks

It should not be a very difficult task to write a script that reads a `.feature`-file and outputs a template in c++, using the Catch syntax.

If one were to *not* go for BDD-style of testing, then one could go for TDD testing using Boost Test, if one would want to keep using Boost for most parts of the project.

3.2 Database

The database of choice, and in the requirements of the project, is PostgreSQL[3], with the extension PostGIS[4] which gives the database *spatial* and *geographic* capabilities, which are needed to simplify working with maps and such, for example when needing to measure distances in different projections. How to set up the database with users and passwords and such are not given in this report, but it is not so hard. When setting up databases one can interact via either the commandline or a *graphical user interface, GUI* such as *pgAdmin3*.

3.2.1 Tools, installation and usage

The tool set was given in the requirements, as mentioned before. On my Debian/Ubuntu system they can be installed as shown in listing 3.3.

[2] `https://github.com/cucumber/cucumber-cpp`
[3] `http://www.postgresql.org/`
[4] `http://postgis.net/`

```
$ sudo apt-get install postgresql postgresql-contrib-9.3 postgis postgresql-9.3-postgis-2.1 pgadmin3
↪   osm2pgsql
```

Listing 3.3: Installation of database tools

Listing 3.4 shows how to create a new database called `mikh_db` with a user "jonas" (that is already set up as a user with rights to create databases), and enabling the needed spatial extensions to work with map data.

```
$ createdb mikh_db -U jonas
$ psql -U jonas -d mikh_db -c "CREATE extension postgis;"
$ psql -U jonas -d mikh_db -c "CREATE extension postgis_topology;"
$ psql -U jonas -d mikh_db -c "CREATE extension hstore;"
$ psql -U jonas -d mikh_db -c "SET search_path=topology, public;"
```

Listing 3.4: Create database and enable spatial extensions.

3.2.2 Loading map data

To get the `.osm`-file, which is actually in `xml`, into the database one needs a conversion tool to parse the file and populate some tables with data.

Tools, installation and usage

There exists several tools for importing `OSM` data into a database. It was hard to know which one to pick and different options were tried, but the chosen tool is osm2pgsql[5]. It was installed in listing 3.3.

```
$ osm2pgsql -U jonas -d mikh_db -k -s mikhailovsk.osm
```

Listing 3.5: Usage of `osm2pgsql`.

Listing 3.5 reads an `.osm`-file in the current directory and populates the database `mikh_db`. The flags `-k` tells to use "hstore" for tags and, `-s` to make a "slim" conversion. Two different `.osm`-files have been provided for testing, "mikhailovsk.osm" and "partille.osm", hence the usage of "mikhailovsk.osm".

One might specify other flags as well. Among the options is to chose a different projection than the default `900913`. It is also possible to specify a `.style`-file which is a configuration over which tags to import. It is possible to use this file to decide which tags to import into the database and which tags to discard.

Alternatives

There exists a bunch of other tools that can convert OpenStreetMap files into database tables, such as *Osmosis, Imposm, osm2po, osm2pgrouting*, and others; all with different strengths and weaknesses, such as being good and free, but not open source.

3.2.3 Building topology

With the data in the database, it is time to build a topology of the map data, saying how the vertices and edges are connected, to make it possible to build a routable graph. A lot of the "nodes" in the `osm`-data are only useful for describing the geometry, while what is interesting when routing are the nodes that connects edges; that is the junctions at which roads meet. Therefore it is essential to analyze the data an build tables that contain information about the topology.

[5] `http://wiki.openstreetmap.org/wiki/Osm2pgsql`

One can have different thoughts of when to do this. It would be possible to do this at the preliminary step when loading the data into the database. That would be good if one was certain of that the topology is stable. If the network is more volatile, it would be better to build the topology on every query, to be certain that one always has the most up-to-date information. On the other hand; the topology for a road network should be stable, and temporary closures and other changing conditions will be better reflected in tags that can be queried when calculating costs for routing. That is the path taken in this project.

Tools, installation and usage

The choice for this project is PostGIS' topology extension. It is a part of PostGIS, which is already installed.

The osm-data from `osm2pgsql` has a table for all the lines in the map, called `planet_osm_line`, but in addition to roads it contains lines for railways, waterways, borders, buildings etc. So to build routing data we need to extract the lines only representing the roads, and put it in a new table. Listing 3.6 shows that.

```
$ psql -U jonas -d mikh_db -c "CREATE TABLE roads AS SELECT * FROM planet_osm_line WHERE highway IS
↪   NOT NULL;"
```

Listing 3.6: Creating a table with only roads in it.

Then one can build a topology of the roads, as shown in listing 3.7. The first line creates a new schema called `roads_topo` which will hold the topology data in the projection `900913` (the projection used when loading the database). The second line adds a column called `topo_geom` to the table `roads` in the public schema. The third line connects that column with the newly built corresponding topology in the `roads_topo` schema. The topology is built with a tolerance of 1.0 units. The unit for this projection is meters, so it means that if there are several nodes within 1.0 meters or a node within 1.0 meters from a road, they are joined. This can be essential to building a routable network. When running the validation tool in JOSM on the `mikhailovsk.osm`-file, it reported 16 suspect cases with nodes close but not connected, see figure 3.1.

```
$ psql -U jonas -d mikh_db -c "SELECT topology.CreateTopology('roads_topo', 900913);"
$ psql -U jonas -d mikh_db -c "SELECT topology.AddTopoGeometryColumn('roads_topo', 'public',
↪   'roads', 'topo_geom', 'LINESTRING');"
$ psql -U jonas -d mikh_db -c "UPDATE roads SET topo_geom = topology.toTopoGeom(way, 'roads_topo',
↪   1, 1.0);"
```

Listing 3.7: Building a topology with PostGIS.

Alternatives

One might load the database, and build a topology with osm2pgrouting[6], and the PostgreSQL extension pgRouting[7]. That solution is pretty smooth, and might heal the topology with a tolerance, but it seems it only builds the topology and does not give access to tags and other information usable when calculating costs.

Another attempt, was to run a topology building SQL function (as in `http://blog.loudhush.ro/2011/10/using-pgrouting-on-osm-database.html`), and then run another function to remove all nodes without topological meaning. But that lead to the problem shown in figure 3.1, as there had been no "healing" of nodes first. One solution could of course to write antoher function for that, or to fix the `.osm`-file manually in JOSM before loading it into the database. But the solution with the PostGIS topology seems like a better way to go.

[6] `http://pgrouting.org/docs/tools/osm2pgrouting.html`
[7] `http://pgrouting.org/index.html`

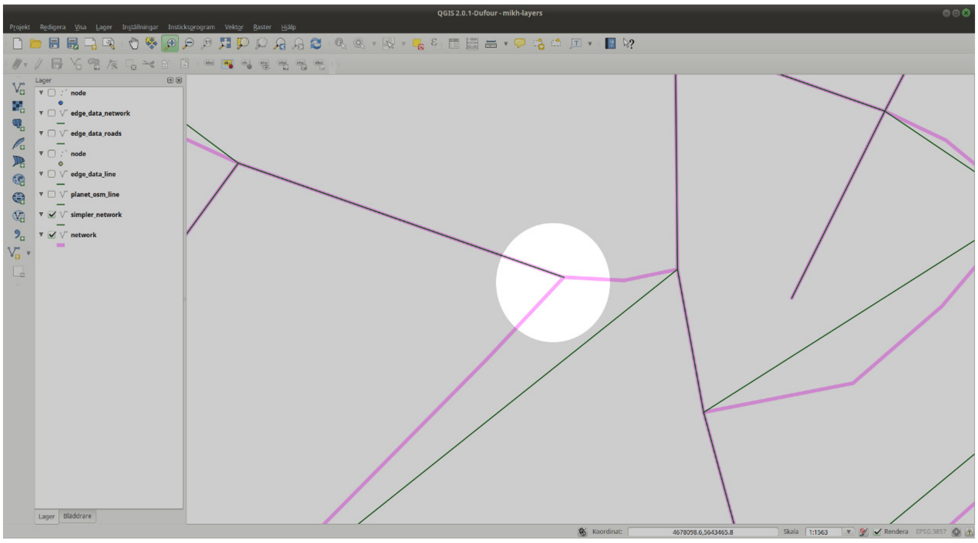


Figure 3.1: Error building topology with a node close but not connected.

3.2.4 Examining map data

Map data lends itself to visualization. And it is also useful to build a mental model of what one is working on, and to see the results.

JOSM

JOSM[8] is an editor for OpenStreetMap. It can open `.osm`-files and display them, inspect elements of the map, and it has tools for editing and validation, meaning one might be able to fix files that has problems. See figure 3.2.

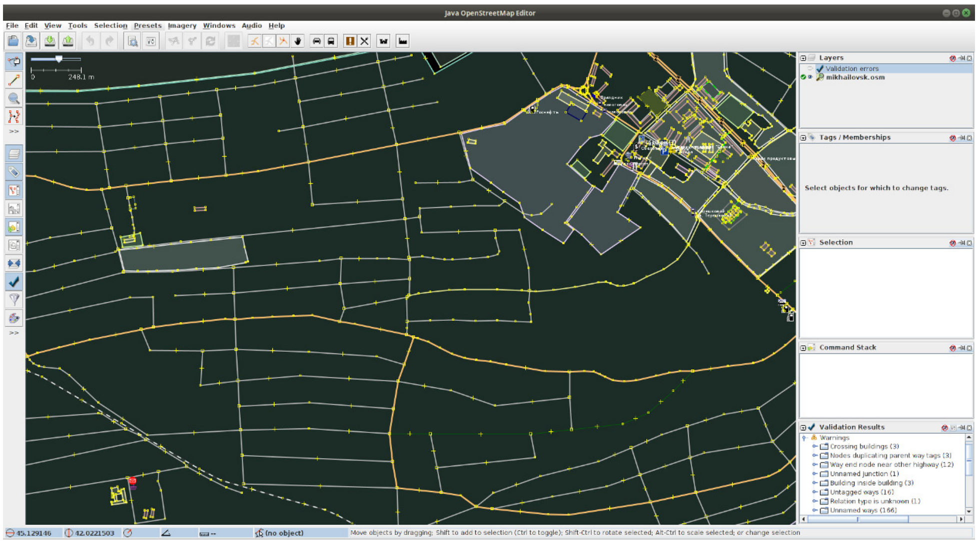


Figure 3.2: JOSM editor with Mikhailovsk map.

[8] `https://josm.openstreetmap.de/`

QGIS

QGIS[9] is a tool that can load spatial data from databases and display, as well as load for example `.osm`-files. It makes it good to visualize for example query results or transformations you have made in the database. See figure 3.3 for an example with layers of PostGIS-data of “Mikhailovsk” on top of each other.

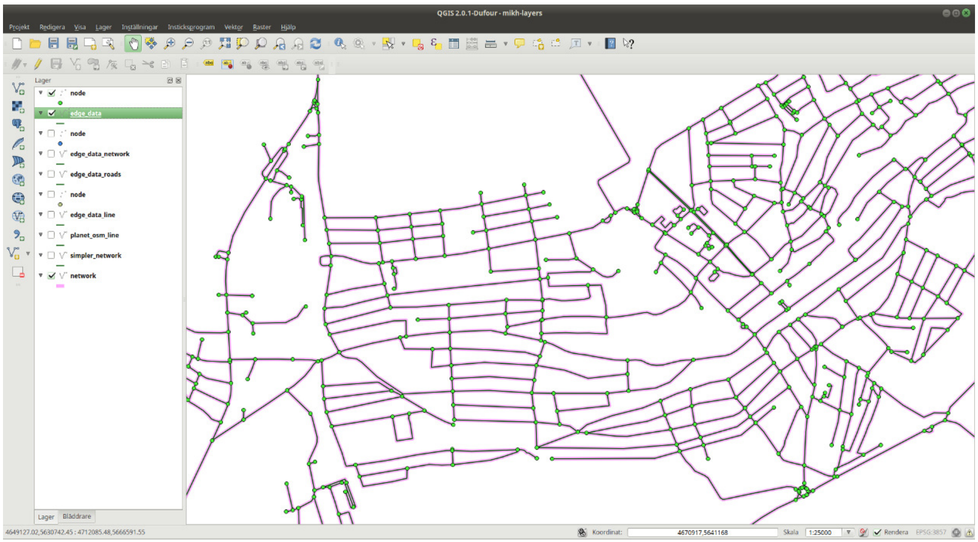


Figure 3.3: QGIS editor with Mikhailovsk map from PostGIS tables.

When loading data from PostGIS one might have to specify which projection to use for display. The default projection when loading `.osm`-files into the database using `osm2pgsql` is `SRID 900913`, and to display that correctly in QGIS one needs to use the projection `EPSG:3857`.

3.2.5 Connecting to database

After the module has read in the configuration, the next step is to connect to the database and perform some work on the map data before extracting relevant information.

The connection to the PostgreSQL database is handled by the library libpqxx[10], and while there exists a few alternatives, it is natural to go for the official alternative.

Tool, installation and usage

Installation of `libpqxx` is shown in listing 3.8.

```
$ sudo apt-get install libpqxx-4.0
```

Listing 3.8: Installing `libpqxx` on a Debian/Ubuntu system.

It is pretty straightforward to use: include the header, make connections and transactions. A snippet is shown in listing 3.9.

```
#include <pqxx/pqxx>
...
pqxx::connection conn("dbname=testdb user=tester password=tester hostaddr=127.0.0.1 port=5432");
```

Listing 3.9: Include header and make a connection to the database.

When compiling, one must link with the libraries `pqxx` and `pq`, as shown in listing 3.10.

[9] http://www.qgis.org/
[10] http://pqxx.org/development/libpqxx/

```
$ g++ mytest.cpp -lpqxx -lpq -o mytest
```

Listing 3.10: Linking `libpqxx` at compile time.

3.3 Configuration

The module should be configured by a settings file, written as *json*. Settings can be related to the database such as host address, table names etc; or it can be configuration of costs for the routing such as speed limits, traffic lights, turn restrictions.

3.3.1 Tools, installation and usage

There is no meaning in writing a json-parser for this module as there exists lots of good libraries. The one chosen is Boost Property Tree[11], as the project uses other Boost libraries, and it simple enough to get started with.

As several Boost packages will be used in this project, it is just as good installing all of them (for a Debian/Ubuntu based system), see listing 3.11.

```
$ sudo apt-get install libboost-all-dev
```

Listing 3.11: Installation of Boost libraries.

An example to see how simple it is to parse a json-file is shown in listing 3.12.

```cpp
#include <string>
#include <iostream>
#include <boost/property_tree/ptree.hpp>
#include <boost/property_tree/json_parser.hpp>

void readJsonFile(const std::string& filename) {
    boost::property_tree::ptree pt;
    boost::property_tree::read_json(filename, pt);
    std::string host = pt.get<std::string>("host");
    int port = pt.get<int>("port");
    std::cout << "Host: " << host << ", port: " << port << std::endl;
}
```

Listing 3.12: Parsing a json-file.

3.3.2 Alternatives

One could go for a header-only solution here as well, such as jsoncons[12], which was also tested, but *Boost Property Tree* seemed nice and easy to get working if one already has the Boost libraries installed.

3.4 Build Graph

The requirements said that the *“Boost Graph Library (BGL)”* should be used for representing the graph and for returning the *line graph* structure for routing back to the calling application.

As discussed in section 2.1.1, the most space efficient way of representing a sparse graph is an *adjacency list*, and the *BGL* has such a data structure. Using template arguments one can configure what kind of data structures to use for the lists of *edges* and *vertices*, and the data structures to use for *edges* and *vertices*, and if the graph is *directed* or *undirected*.

If one has some properties of the edges and vertices that one wishes to keep in the graph (like the “cost” or some identifier of an edge), it is possible in several ways, either as *“interior”*

[11] http://www.boost.org/doc/libs/1_54_0/doc/html/property_tree.html
[12] https://github.com/danielaparker/jsoncons

or *"exterior"* properties, and `adjacency_lists` can use *interior* properties either as *"bundled properties"* or as *"property lists"*.

The *property lists* are external structures for some property that gets mapped to e.g. an edge in the graph.

The *bundled properties* are more intuitive, by using data structures as the *descriptors* of the *edges* and *vertices*, and with the properties as fields.

An example from the documentation for *bundled properties* [13] shows the difference clearly, in terms of how easy or hard it is to read or understand the code in the different approaches. See listing 3.13 showing the *bundled* approach and listing 3.14 showing the *property list* way.

```
// Vertices = Cities
struct City
{
  string name;
  int population;
  vector<int> zipcodes;
};

// Edges = Highways
struct Highway
{
  string name;
  double miles;
  int speed_limit;
  int lanes;
  bool divided;
};

// Map using `City` as vertex descriptor and `Highway` as edge descriptor.
typedef boost::adjacency_list<
    boost::listS, boost::vecS, boost::bidirectionalS,
    City, Highway>
  Map;
```

Listing 3.13: Bundled properties in a graph.

```
typedef boost::adjacency_list<
    boost::listS, boost::vecS, boost::bidirectionalS,
    // Vertex properties
    boost::property<boost::vertex_name_t, std::string,
    boost::property<population_t, int,
    boost::property<zipcodes_t, std::vector<int> > > >,
    // Edge properties
    boost::property<boost::edge_name_t, std::string,
    boost::property<boost::edge_weight_t, double,
    boost::property<edge_speed_limit_t, int,
    boost::property<edge_lanes_t, int,
    boost::property<edge_divided, bool> > > > > >
  Map;
```

Listing 3.14: Property lists in a graph.

In this project, the *bundled properties* were chosen for their ease of understanding and reading.

4 Implementation

As stated at the start of chapter 3, the software module called the *"Line Graph Utility", (LGU)*, that should be the outcome of this project, is sequential in nature. The complete specification is available in appendix A, but here is an outline of the main use case.

- The using application calls the LGU's `get_directed_line_graph()`.
- LGU queries the PostGIS database and builds a graph from the road network.
- LGU builds a directed line graph from the previous step (i.e. it converts nodes to edges and assigns weight to those edges based on road signs and other elements which are present in the nodes).
- `get_directed_line_graph()` returns a directed line graph structure which is based on a C Boost graph structure to the function caller.

This can be expanded to a series of steps. First comes a preliminary step, not actually part of the module, but essential during development and testing:

- Loading the map data into the database and build a topology.

The following steps are performed during development and usage of the tool:

- Load configurations from *json*-file.
- Get the relevant edges and vertices from the database; store the topology.
- Apply restrictions and costs on the topology.
- Build a graph structure from the topology, using *Boost Graph Library.*
- Transform the structure into a *line graph (edge-based/arc-based graph).*
- Return the line graph.

4.1 Design

The sequential nature of the module, with a few easily identifiable objects, lead to no big design process was deemed necessary. Taking an object oriented approach, it is easy from the above list to identify *configuration (and configuration reader); edges; vertices; database; topology; restrictions; costs; graph (and graph builder and transformer); line graph.* All can be packaged up in a *Line Graph Utility.* The design therefore evolved gradually without a master plan more specific than this.

Another reason for this, was that this project was a discovery into not really well understood territory, despite some introductory research. It was necessary to learn the tools and concepts as the project proceeded, so the design and implementation grew incrementally. The incremental goals set during development, was to be able to build a graph from the map data, later extended to being able to build a line graph from that, to finally being able to apply restrictions and costs to the graphs.

A decision that was made early on, was to try not to pass pointers around, but instead use references, to reduce the complexity of memory handling. That means that a lot of functions gets passed in a reference to an object to fill in, rather than return a pointer to a newly constructed object. All the same, some pointers could not be avoided and raw pointers were used in those cases.

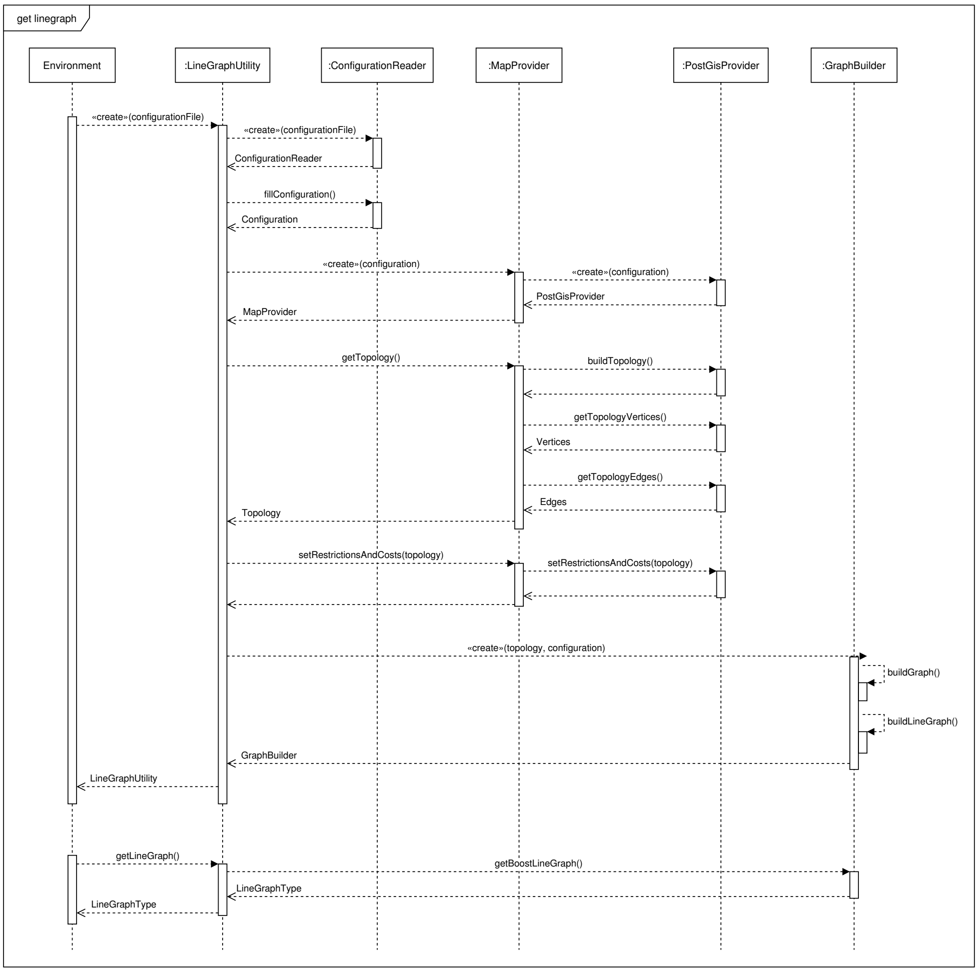


Figure 4.1: Sequence diagram of main use case to get a line graph.

4.1.1 Dynamic design

The sequence presented above has been refined into a design that can be shown in a sequence diagram, see figure 4.1 and appendix B.1.

The calling application *"Environment"* instantiates a `LineGraphUtility` object with the file name to a configuration file. The `LineGraphUtility` instantiates a `ConfigurationReader` that can be asked to fill in a `Configuration` object. The configuration contains among other things, a setting for which `MapProvider` to use. The idea is that one can read the *OpenStreetMap* data in several ways; for example parse the `.osm`-file, or use different databases or different tools to import `.osm`-files into the database. Hence the flexibility by using an abstract *map provider*. The only implementation in this project so far is the `PostGisProvider`, but others could be developed if it turns out there are better ways to access the map data.

So the actual work on retrieving the map data is performed by the `PostGisProvider`, that is fetching the `Topology` and applying *restrictions* and *costs* on the topology. The idea behind this separation is that the topology should be reasonably stable and constant, and that dynamic changes in the traffic, such as blocked roads, should be handled as restrictions and costs that are applied to the static topology. But it is also possible to perform an update on the topology if needed, for example if there has been built a new road. See figure 4.2 for a diagram of updating the restrictions and costs, and figure 4.3 for a diagram on updating the topology, (also in appendix B.2 and appendix B.3).

Back to figure 4.1, after having a restricted topology we instantiate a `GraphBuilder` object with the topology och configurations. This `GraphBuilder` builds a directed graph, and converts it to a *line graph*. If all went well the `LineGraphUtility` now is ready to serve the calling application a *line graph* any time it gets called.

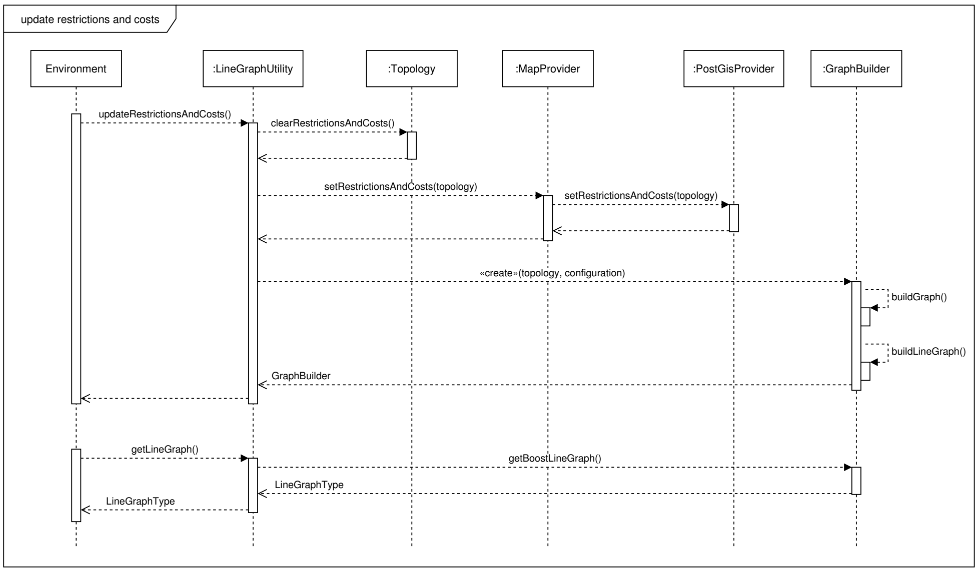


Figure 4.2: Sequence diagram of updating costs and restrictions on a topology.

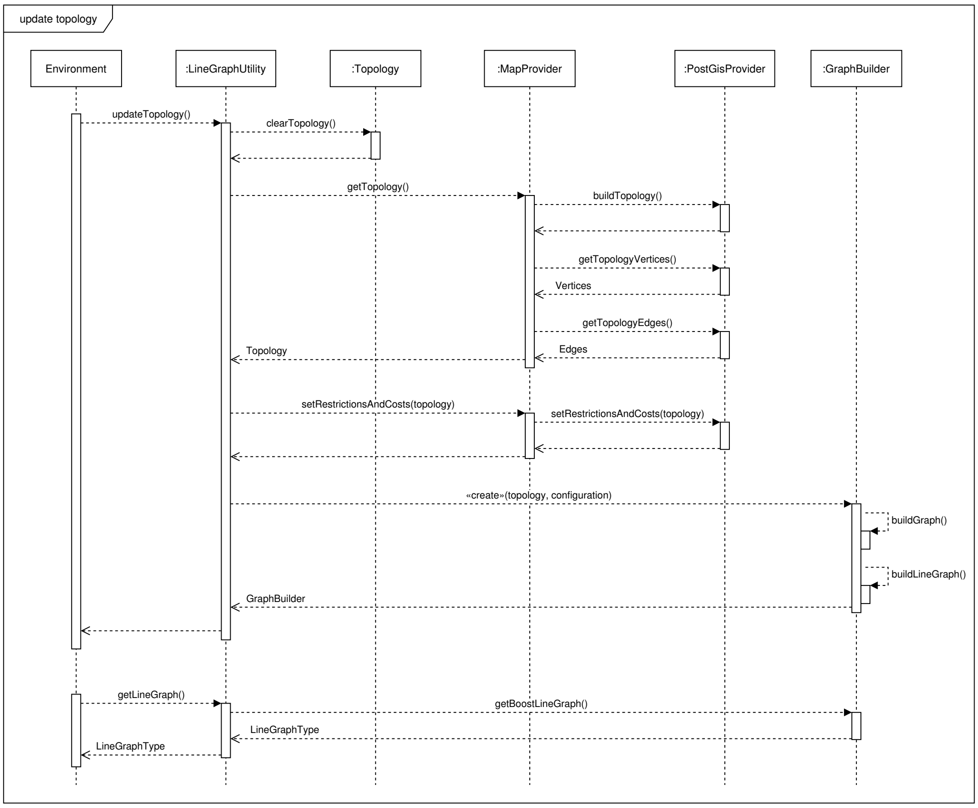


Figure 4.3: Sequence diagram of updating the topology.

4.1.2 Static design

A few classes were introduced in the sequence diagrams above, and a more complete view of the classes can be seen in the class diagram in figure 4.4 and in appendix B.4.

As can be seen, the application is divided in a few *"packages"*. The division is more to help when navigating code, it is not enforced by namespaces:

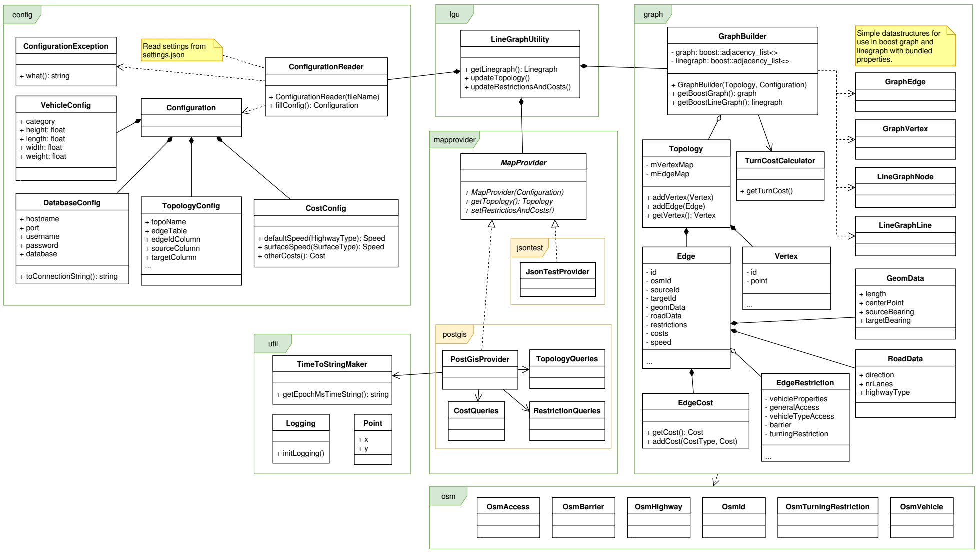


Figure 4.4: Class diagram of the *line graph utility*.

- `lgu`: The entry point to the `LineGraphUtility`.
- `graph`: Classes related to graphs like `GraphBuilder`, `Topology`, `Edge`, `Vertex`.
- `mapprovider`: Classes related to providing map data.
- `config`: For handling configurations.
- `osm`: Helper classes for constants and concepts related to *OpenStreetMap* data.
- `util`: A few general helper classes.

This is an attempt to modularize the development process; to keep related classes in a specific area. It also makes navigating the code easier. Another attempt to make the packages coherent is that each package should have its specific *exception* class, that is the only public exception that gets thrown by the classes in the package. Other exception classes might be used internally but not exposed publicly.

4.2 Project structure

Apart from the packages above, there are directories in the project to support testing, setting up and documentation. This gives the project a basic directory structure as shown in listing 4.1.

```
./
├── catchtest/
├── config/
├── doc/
├── graph/
├── lgu/
├── mapprovider/
├── osm/
├── preparation/
├── util/
└── README.md
```

Listing 4.1: First level directory structure of the project

Each directory has a `README.md` file, a textfile in `markdown` mark up, that explains the purpose of the directory. Each directory with code also should have a `catchtest` directory, where there are tests for the code in the directory/package.

4.2.1 catchtest

Most of the code developed in this project has been part of a test, and all *"packages"* have their own set of tests. *Catch* is a header-only framework, and that header resides in this root `catchtest` directory, and there is a source file that calls that header and functions as the entry point when testing, see appendix listing C.2. It also contains a sub-directory for configuration settings used during development.

In section 3.1.1 the *Catch* testing framework is introduced, and in listing 3.2 there is a small example of how to write a scenario. There one can see that the scenario is tagged with `"[vector]"`, and those tags can be used to determine which tests to run. If no tags are specified all tests are run, but if one specifies a tag, it only runs tests that matches that tag. One can also specify which tests *not* to run by prepending a tilde ('~') to the tag. See listing 4.2. If one wants to see the results of all tests and not only failed ones, one can add the flag `-s`. If running tests from inside and *integrated development environment, IDE* one can specify the arguments in a *"run configuration"* instead.

```
$ testapp ~[timing] -s
```

Listing 4.2: Running tests except those tagged with `[timing]`, showing all results.

4.2.2 config

See appendix listing C.3 for the contents of this package, whose purpose is handling configurations. The central part is a data structure `Configuration`, made up of data structures for *cost, database, topology* and *vehicle*. The configurations are filled in by the `ConfigurationReader` class which reads from a specified settings file.

The `CostConfig` are mainly concerned with keeping track of speeds for different categories of roads and surfaces. The types are specified in `OsmHighway` in the `osm` package, see section 4.2.7.

The `DataBaseConfig` is about connecting to the database.

The `TopologyConfig` is about which tables and columns in the database to use when getting the topology.

The `VehicleConfig` keeps characteristics about the vehicle we are routing through the map, such as weight, height, category (as specified in `OsmVehicle`, see section 4.2.7).

4.2.3 doc

Listing C.4 shows the contents of this package, that contains the documentation for the project. It has a directory for this report, and a directory for the UML diagrams.

The diagrams are not meant to be exact documentation, but rather give an idea of the concepts and the big picture, and therefore method names might be missing or spelled differently than in the actual code.

4.2.4 graph

The `graph` package (see appendix listing C.5) is really the central package, where the `Edge, Vertex, Topology` classes are, and the `GraphBuilder` resides. In addition there are the classes for restrictions and costs for edges (`EdgeCost, EdgeRestriction`); a helper class for calculating costs for turns (`TurnCostCalculator`); and a couple of essential simple types, `Cost` and `Speed` who are simply `typedefs`.

Edge

(See source code in appendix D.7.2 (header), D.7.3 (implementation)).
The `Edge` class represents an edge in the topology. So it keeps track of the *source* and *target* vertices, the original *OSM id* and of course its *id* in the topology. It also keeps track of properties of the underlying *road* (number of lanes; one-way; the road category), and its geometrical properties (length; centre point and *bearing* of the edge at the vertices to calculate turning angles).

Vertex

(See source code in appendix D.7.4 (header), D.7.5 (implementation)).
The `Vertex` class simply keeps the *id* it has in the topology and the *coordinates* of it, it does not keep track of an *OSM id* as it might not correspond to nodes in the *OSM* data, since the vertices were calculated when building the topology.

Topology

(See source code in appendix D.7.6 (header), D.7.7 (implementation)).
The `Topology` class is a collection of maps. One maps `{edgeId ↦ edge}`, that is it makes it possible to get to the topology's `Edge` object when one only has an *id* for it in the topology. The corresponding map exists for vertices, `{vertexId ↦ vertex}`. In addition there is a *multi-map* `{osmId ↦ edgeId}` that has an `osmId` as the key, that maps to several `edgeId`s. The reason for this is that an original road in the *OpenStreetMap* might be split into several edges when building the topology.

GraphBuilder

(See source code in appendix D.7.8 (header), D.7.9 (implementation)).
The `GraphBuilder` keeps a `Topology` and a `Configuration` as the base for building a *graph* and a *line graph*. The `GraphBuilder` header begins with defining a bunch of types, such as the data structures to be used for *edges* and *vertices* in the *Boost graph* (`GraphEdge`, `GraphVertex`). They keep track of the *id in the topology* and the corresponding *id in the graph*, so one can move from the one to the other. The `GraphBuilder` keeps a *map* for *vertex* such as `{topoVertexId ↦ graphVertexDescriptor}`, and a *multi-map* for *edges* such as `{topoEdgeId ↦ graphEdgeDescriptor}`. These maps make it possible to access the data in the data structures underlying the *descriptor*, e.g to get to the fields in the `GraphEdge` through the descriptor. The reason to use a *multi-map* for the edges are that the edge in the topology is undirected, but in the graph the edges are directed, so in the graph there should be a directed edge for each lane of the road, and thus for most roads there will be several *graph edges* for each *topology edge*.

The `GraphBuilder` also has the data structures that makes up the *nodes* and *lines* in the *Boost line graph*, and a map `{edgeId ↦ lineGraphNodeDescriptor}`, since that is the definition of a *line graph*: an edge turns into a node in the transformed graph.

There are also `typedefs` to make the code easier to work with, see listing 4.3.

```
typedef boost::adjacency_list
   < boost::listS, boost::vecS, boost::directedS,
    LineGraphNode, LineGraphLine >                            LineGraphType;
```

Listing 4.3: `typedef` a *line graph* to make the code more readable.

The operation when building the *graph* is that first all *vertices* in the *topology* gets added to the *Boost graph*, and then each *topology edge* gets examined to see if there are any restrictions that apply. If not, the correct number of edges (corresponding to the number of lanes) in each direction gets added to the graph.

When building the *line graph*, all the edges in the graph are added as *nodes* in the *Boost line graph*. To find which other nodes to connect to (= which travels are allowed), one has to look at all *out-going* edges from the *end vertex* of the edge, functioning as a *via-vertex*. That *turn* (or travel) is really in three parts: *source edge* → *via vertex* → *target edge*. The *OpenStreetMap* data also gives the option to specify turns as travel via another edge instead of via a vertex, but that is complicated, see discussion in 4.2.7. When the *adjacent* edges have been identified, they are one by one checked if they are part of any *turning restriction*. If the edge is not part of such a restriction, then a *line* (that is a *line graph edge*) is constructed from the *source node* to this *target edge/node* and added to the *line graph*.

EdgeCost

(See source code in appendix D.7.12 (header), D.7.13 (implementation)).
The `EdgeCost` is a class for keeping track of different costs for edges. It has tree types of costs: *travel time; barriers; other*. The *travel time* cost represents the time it takes to travel the edge, and is thus dependent on the *length, speed limit or road category* and *surface*. The *barrier type* is for costs that comes from slowdowns imposed by barriers such as *speed bumps, gates* and such. The *other* cost are for slowing down for *signs, traffic signals, zebra crossings* and the likes.

A note on stop signs: they are associated with a road. But it is generally only applicable in one direction. For example the stop sign only affects the incoming edge in a junction, not the reverse direction of the same road going out of the junction. Therefore one needs to look at the position of the stop signs and find out which junction it really belongs to, and then only apply the cost to the affected edge. This is not implemented yet, so at the moment edges in both directions of roads with signs have costs added, which is faulty behavior.

TurnCostCalculator

(See source code in appendix D.7.16 (header), D.7.17 (implementation)).
When calculating the *costs* or *weights* for a *line* in the *line graph* it is the cost for the *source node/edge* plus the cost for the *turn*. This `TurnCostCalculator` helps with that. The calculations for this has been re-factored out to its own class as one can imagine wanting to include different properties when calculating the cost. Thus it would make sense to make this an interface and add different implementing classes, but this project just has this one implementing class for now, and therefore skipped the interface.

The inspiration for the calculations made by this calculator comes from [7], but not all factors in that paper are included here.

It is obvious that it is more costly to make a sharp turn, as one needs to decelerate coming in to the turn, and accelerate going out of the turn. The sharper the turn, the slower one needs to go. The deceleration and acceleration characteristics are properties of the routed vehicle. Also if one is coming from a lower ranking road category and is turning into a higher category, one needs to give way, which is also a cost.

EdgeRestriction

(See source code in appendix D.7.14 (header), D.7.15 (implementation)).
Restrictions for edges/roads can be somewhat complicated. Some regulates *general access*[1] with values such as *yes, no*, but in addition much more arbitrary values such as *permissive; designated; discouraged; customer*. Then other restricts access depending on the *vehicle type* such as banning *cars* but allowing *buses*. Then again, the restriction can depend on the *vehicle properties* such as *weight* or *width*. In some cases, such as *sump buster barrier*[2], it can ban access for a car, but only impose a cost on a bus. A road can also be tagged as *disused*, which is clear, but it is not so clear what to do with a road marked as *no-exit*.

An edge might no have a restriction by itself, but be part of a *turning restriction relationship*, so that one can not turn from one edge to another, although traffic is allowed on both edges. In addition, the specifications (see appendix A) said that *conditional restrictions*

[1] `http://wiki.openstreetmap.org/wiki/Key:access`
[2] `http://wiki.openstreetmap.org/wiki/Tag:barrier%3Dsump_buster`

should be respected, that is restrictions that only apply for example at a certain time, a certain day of the week, for vehicles with certain properties or of a certain category, see figure 4.5.

The *conditional restrictions* has not been implemented yet, and the whole class is marked by being developed incrementally while discovering how many separate and complex parts of *OpenStreetMap* represents some kinds of restrictions.

Figure 4.5: Conditional restrictions. [14]

4.2.5 `lgu`

The `lgu` package (see appendix listing C.6) is the *entry point* into the whole software module. The specification (appendix A) said that the module should be called from a function `get_directed_line_graph()`. This has not been written yet, so the entry point is by instantiating a `LineGraphUtility` object and call `get_line_graph()` on it, but it would be simple to write a wrapper to actually provide the specified function if needed.

The package is really only one class, `LineGraphUtility`, and how it works has been described in section 4.1.1.

4.2.6 `mapprovider`

This package (see appendix listing C.7), should contain sub-packages, as the `mapprovider` directory otherwise only contains an interface, `MapProvider`, and an exception class. The interface is the way to get map data from a source (such as a database) into the classes of the application.

There are two sub-packages in the project. One is `jsontest`, which in the initial phases of the project was used to load a small set of edges and vertices from a `json` file. It has been abandoned after loading from database was developed, but still hangs around.

The other sub-package is `postgis`, which is a map provider that uses a *PostGIS* database with the `postigs_topology` extension as the source for map data. This is where a lot of development has taken place during this project.

`postgis`

The `postgis` package uses the `libpqxx` to work with the *PostGIS* database. The `PostGisProvider` class gets passed in a `Topology` object to modify when asked for a *topology* or to set *restric-*

tions and costs. It also knows how to persist the *lines* and *nodes* of a *line graph* back to the database, which was desired functionality in the specification (see appendix A).

All the logic to work with the database and how to fill in the topology exists in this package. To make it more manageable, the `PostGisProvider` has four helper classes to actually perform the queries and handle the results from the database. They have names that describes their area of work: `CostQueries`; `LineGraphSaveQueries`; `RestrictionQueries`; `TopologyQueries`. They are all *static* classes and cannot be instantiated, one can only call the methods statically.

Some remarks about those classes:

The `TopologyQueries` simply fetches the relevant data for vertices and edges. For the latter case, it also performs some calculations in the *SQL* query to calculate the geometric data.

The `LineGraphSaveQueries` creates a new schema and table and inserts some basic information about the *nodes* and *lines*.

The `RestrictionQueries` has to extract all the different information for *edge restrictions* (see section 4.2.4). It uses an inner class for *turning restrictions* to work with those queries and to extract `OsmTurningRestriction` data (see section 4.2.7) so those restrictions can be resolved. Turning restrictions are not really attributes of edges, but *relations* in the *OpenStreetMap*, and the `osm2pgsql` tool for importing *osm* data into a *PostGis* database does not really handle relations so they can be used straightforwardly[3]. Therefore some workarounds have been made: In the process of initializing the database on creation a `turning_restrictions` is created and a couple of custom *sql* functions are installed (see appendix D.13.3 and D.13.2), that extract *turning restrictions* relations from the table `planet_osm_rels`, and parses what kind of restriction it is and the *osm ids* of the members (i.e. the edges and vertex involved). With those *ids* the involved *topology edges* are identified and stored as a string as the that is easier to make use of in the program than an array. The result are stored in the `turning_restrictions` table, and when running the `RestrictionQueries` for turning restrictions the topology ids are parsed and operation can continue.

4.2.7 `osm`

This package (see appendix listing C.8) deals with handling concepts and constants in *OpenStreetMap* data, such as enumerating the different categories of *accesses*[4], *barriers*[5], *highways*[6] and *vehicles*[7].

OsmTurningRestriction

In addition to those classes above, there is a class for dealing with the concept of *turning restrictions*, which are *relations* between *edges* and *vertices* in an *OpenStreetmap*. This class is an attempt to keep track of that information. In *OSM* a turning restriction is a relation of *(from → via → to)*. The 'via' part can be either a vertex (at a junction) or other edges, saying "travel from Here to There via roads This and That are not allowed". That kind of relationship is a lot trickier to represent, especially for this software module that only should build a *line graph* of the allowed turns, but has no routing information and thus cannot decide if a "via way" relation is allowed or not. It has therefore been disregarded in this project, and a routing application needs to decide that information some other way. The class `OsmTurningRestriction` has a field telling if it is a *via way* or a *via vertex* restriction.

4.2.8 `preparation`

Before anything else can be done, one needs to prepare the database. That means installing needed extensions to handle geometric and geographic data, and set up some tables and functions needed. Then one can add the map data to the database.

[3] http://wiki.openstreetmap.org/wiki/Osm2pgsql/schema
[4] http://wiki.openstreetmap.org/wiki/Key:access
[5] http://wiki.openstreetmap.org/wiki/Key:barrier
[6] http://wiki.openstreetmap.org/wiki/Key:highway
[7] http://wiki.openstreetmap.org/wiki/Key:vehicle

Appendix listing C.9 show the contents of this package. There is an `.sql` file (see listing D.13.3) for initializing extensions `postgis; postgis_topology; hstore` and installing functions for finding *turning restrictions*. And there is a `.sql` file to use when building the topology in advance. Then there is a file `LGU.style` which tells `osm2pgsql` which tags to create columns for in the tables, and which tags to ignore. Then there is also the original `.osm` files with map data for *Mikhailovsk* and *Partille* (they are not included in the appendix, but there are instructions how to download them there, see appendix listing D.1 and D.2).

The way to prepare the database is shown in listing 4.4, which sets up a new database `mikh_db` for the *Mikhailovsk* map data.

```
$ # 1. Create database
$ createdb mikh_db -U tester

$ # 2. Install extensions and functions
$ psql -U tester -d mikh_db -f init_osm2pgsql_postgis_topology.sql

$ # 3. Import OSM data
$ # Flags:  -s    Slim mode (add data to db, do not build all in memory)
$ #         -k    Keep tags in `hstore` if not in own column
$ #         -S    Style-file to use
$ osm2pgsql -U tester -d mikh_db -s -k -S LGU.style mikhailovsk.osm

$ # 4. Building topology (optional)
$ psql -U tester -d mikh_db -f build_postgis_topology.sql
```

Listing 4.4: Preparing a database with map data.

4.2.9 `util`

This package (see listing in appendix C.10) contains a few utility classes: one for *logging* (using *Boost logging*) to be used where needed in the application; one for a *coordinate point* and one for *producing strings from current timestamp* which is used when building temporary topologies.

4.3 Development environment

Development of the project and the coding has taken place in *Eclipse Luna 4.4.2*. The build system is the default in Eclipse on Linux, generating *makefiles*.

- Compiler flags:
  - `std=c++11`
  - `DBOOST_LOG_DYN_LINK`
  - `O0`
  - `g3`
  - `Wall`
  - `c`
- Linker flags:
  - `lboost_log`
  - `lboost_log_setup`
  - `lboost_thread`
  - `lboost_system`
  - `lpthread`
  - `lpqxx`
  - `lpq`

The coding was to follow a *coding standard* (see appendix A.7) which regulates the naming scheme and the layout of the files.

As for working with the database the main tool has been *pgAdmin3*.

5 Results

The software module developed in this project does not fulfill all requirements (see appendix A), in that it does not handle *conditional restrictions* at all, and not all implemented restrictions are handled correctly, see section 5.1 below.

But the software does build a *line graph* that can be fetched and stored in database for inspection with visual tools, see section 5.2 below.

The project has so far been about implementing things and have not had any focus on performance, but some performance tests have been run, see section 5.3.

5.1 Specification fulfillment

Table 5.1 shows how much of the specification that has been fulfilled.

Table 5.1: Fulfillment of specification.

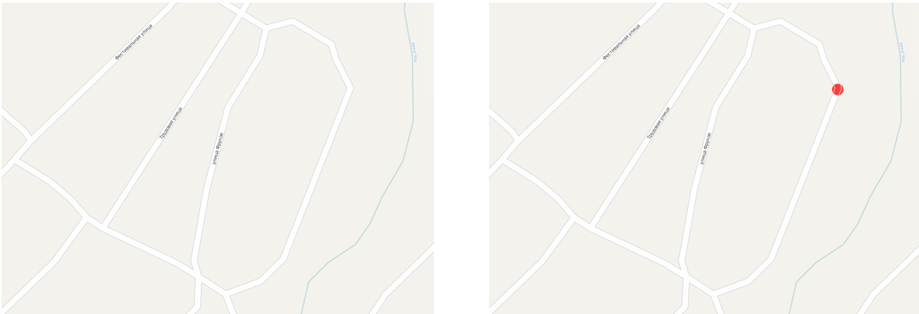
Section	Fulfills	Comment
1.2 Main use case		
1.2.1	X	As call to `LineGraphUtility::getLineGraph()`.
1.2.2	X	
1.2.3	X	
1.2.4	X	
1.3 Optional use case		
1.3.1	X	
1.3.2	X	
1.4 Functional requirements		
1.4.1		Lots of work remains to implement all restrictions.
1.4.2	X	
1.4.3	X	
1.4.4	X	Some small parts are hard coded.
1.5 Non-functional requirements		
1.5.1	X	Written in C++.
1.5.2	X	Did not find pgRouting really useable.
1.6 Testing requirements		
1.6	X	
1.7 Coding standard		
1.7	X	

5.2 Visual examination

Maps are easy to visualize, and a great number of tools exist to work with map data. Figure 5.1 shows a piece of a map over Mikhailovsk. In order to test if the handling of restrictions work, modified maps have been created. *JOSM*[1] is a tool for manipulating *OSM* data. In figure 5.1b it is indicated where a *bollard barrier* has been added in the middle of a road,

[1] https://josm.openstreetmap.de/

just to see if the restrictions work, and the new map is saved in its own `.osm` file, and a new database built for it.



(a) Original. (b) Modified with added barrier.

Figure 5.1: Map over part of Mikhailovsk. [15]

Using another tool, *QGIS*[2] can be used to load map data from for example a *PostGIS* database and viewed. In figure 5.2 the vertices and edges from the topology for that map has been layered on top of the image (with a slight misalignment). The topology is the same for both maps, it does not change with added barriers.

Figure 5.2: Edges (green) and vertices (blue) for the topology.

The interesting part is to see if the restriction has had any impact on the built *line graph*, see figure 5.3 for the original line graph, where the road is included in the line graph. It has a *node* in the middle and *lines* connecting to the adjacent *edges/nodes*.

Figure 5.4 shows the line graph after the restricting barrier has been added to the map. There one can see that the *edge* (road) has not been added as a *node* to the line graph, while all the other *lines* and *nodes* remain the same. This practically disables routing along that road.

[2] http://www.qgis.org

Figure 5.3: Original *line graph*. *Lines* in purple.

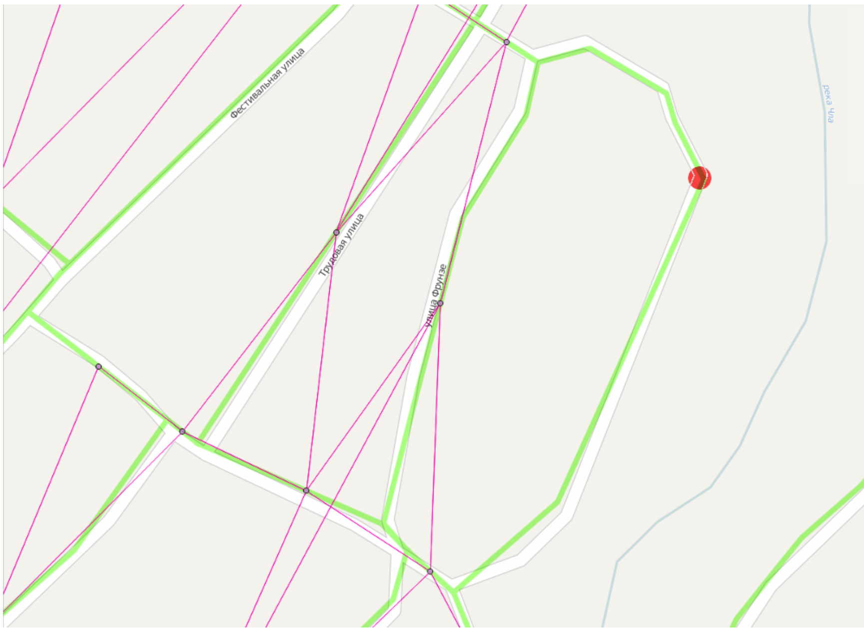


Figure 5.4: *Line graph* after added barrier. *Lines* in magenta.

5.3 Performance

There were *soft real time* requirements in the specification, but they were not specified more than that. But it is interesting so find out how long it takes to fetch a *line graph*, built on demand by the software module.

A few test cases were written in `LineGraphUtility_test.cc` that averages the number of *microseconds* it takes to instantiate a `LineGraphUtility` and fetch a *line graph*, over a given number of rounds.

The test runs on both a configuration with a pre-built topology, and a configuration that builds a temporary topology.

See table 5.2 for test results.

Table 5.2: Time in μs to fetch a *line graph*, with pre-built versus temporary topologies.

	Test #	1	2	3	4	Sum
	# of rounds	10	10	10	70	100
topology		avg (μs)	avg (μs)	avg (μs)	avg (μs)	avg (μs)
Mikhailovsk	normal	147859	149092	141782	133950	143171
	temporary	5026626	4924245	4917319	4875838	4936007
Partille	normal	180340	188405	179883	179978	182152
	temporary	10683194	10342420	10683873	10521535	10557756

The *size* (number of edges) and *order* (number of vertices) of the graphs are shown in table 5.3.

Table 5.3: Sizes of tested graphs

	Graph		Line graph	
	vertices	edges	nodes	lines
Mikhailovsk	654	1618	1618	4758
Partille	1645	2265	2265	5577

The results shows that, in order to meet *soft real time requirements* it is not possible to build temporary topologies at every instantiation of a `LineGraphUtility`, as the time increases dramatically. In the case of *Mikhailovsk*, the increase is from 0.14 s to 4.93 s, about 34 times as much. In the case of *Partille*, the increase is from 0.18 s to 10.55 s, nearly 58 times.

That fetching a line graph with a pre-built topology takes 0.15-0.2 s might fall within the requirements.

The test were conducted on a computer with 8 GB ram, processor Intel i7-4702MQ, running Linux Mint 17.1 with Linux kernel version 3.13.0-37-generic. The comiler flags were the same as for the rest of the project, i.e. no optimization.

6 Discussion

Presented below is my personal views of parts of the project and the outcome of it.

6.1 Research

6.1.1 Graph theory

Starting out on this project, I thought that one of the main obstacles would be no prior knowledge of *graph theory*, so I set out to allow for some time initially to get into the field. I am glad to have gained some fundamental knowledge of the area, but the time spent here could have been less.

6.1.2 Map routing

Reading about theory regarding map routing and graphs was really interesting, and a lot of research has been done in this area in later years. It initially gave me some ideas I thought I would like to try out, but once development got going, those theories vanished in favor of finding working solutions quickly.

6.1.3 Map data

OpenStreetMap is the source of map data for this project, and a lot of high quality projects. That puzzles me somewhat, because I have found it kind of messy. It is an *XML* application, but it has no official *schema*. That is, there is an informal consensus on which tags are good, but one can also make up ones own tags[1]. Another example of the messiness is the `maxspeed` tag, which can have the values *60; 50 mph; 10 knots*. That is, the default case is a unit of *km/h* and one can read the value as numeric. But one cannot be sure of that, because other units are allowed, and in that case one needs to parse the value as a string to find out which unit is used. It would surely have been better to let the unit be an attribute of the value, so one did not need to parse every value. In this project I decided to skip parsing, and just assume all values are *km/h*.

But, as said, a lot of good applications using *OSM* exists, see 2.4, so it is possible to work with. And it might also be unfair to say that *OpenStreetmap* is messy; it might be the case that it simply reflects the complex and difficult reality in the traffic, with lots of different rules and restrictions dependent on context and conditions.

6.1.4 Available applications

The fact that a lot of applications already existed, and some of them being *open source* and using *OpenStreetMap* as the the source for map data, made the direction of this project a little difficult. I proposed to the company that there are some good solutions out there that might just need some adaption to work, but they wanted their own thing. So the question for me was if I was to look at and copy features and concepts of those existing solutions anyway or just blindly go down my own path. In order to steer clear of issues with plagiarism I chose the latter, and that has surely impacted the project negatively. It would have been wiser to build upon the experience of others, developed through years.

6.2 Methodology

The main methodology for the project was supposed to be test driven (either BDD or TDD), but to be honest, most tests were written after the implementation of a feature, functioning

[1] http://wiki.openstreetmap.org/wiki/Map_Features

more as unit test, than driving the development. I don't think that has affected the outcome of the project negatively, it is more a matter which workflow feels best.

6.3 Design

Previously I have bee more into heavy design and modeling before starting coding, but in the last year I have tried to become more "agile", and start testing things out and be prepared to refactor and remodel when needed.

In this project perhaps it could have been good to design more, to have really thought through how the restrictions should work. On the other hand, a lot of the difficulties was discovered only when working on them, so it would be hard to have the full picture before. It is a balance in getting going and learning, and modeling before. What is clear, is that parts of the software as it stands now, should be re-modeled, specifically the *restrictions*.

6.4 Development

6.4.1 Coding standard

This was the first time I had to code to a standard. It was kind of awkward and unintuitive at first as it differs from my personal style, especially since having started to trying to practice *"Clean Code"*. and have less comments and visual dividers in the file. But after a short time the style became pretty easy to use. I don't think I have followed the standard completely, but it was too lng to read and get into before beginning to code.

6.4.2 Memory management

I tried to avoid pointers and only use references, but that turned out to be clumsy, so at times I reverted to using raw pointers. Eventually, I found out that it would have been a lot better to use the smart pointers from C++11 (or even Boost), but I did not want to spend the time needed for learning how to use them and redo the memory management completely.

6.4.3 Tools

OSM conversion

I tested and looked at a number of tools for converting *OpenStreetMap* data to a *PostGIS* database, and the choice fell on *osm2pgsql*. I am not certain that it was the right choice, as it has is shortcomings when working with restrictions. Fortunately, the developed software module is flexible so one can write a new *MapProvider* if one decides to work with another tool, that uses a different approach.

Database

The *pqxx* library was easy and straight-forward to work with.

Boost

This was the first time for me to use *Boost*. I have only used small parts of the library: obviously the *graph* package, the *property_tree* for parsing *json* and the *logging* package. There are some tricky concepts, but also a lot of useful stuff. Getting into all the long names and templates takes some getting used to, but it was OK.

Catch test

I really enjoyed the *Catch* testing library; small and easy to use. It didn't play so nicely with *Eclipse CDT*, marking errors throughout in the editor, but good enough.

6.4.4 OpenStreetMap

Restrictions

Turning restrictions are *relations*, and *osm2pgsql* does not really handle relations, so a lot of parsing was needed. And in the case of *conditional restrictions* I have not found out how to work with them. *osm2pgsql* can be instructed to put tags into separate columns in the database, but with conditional restrictions the tags changes 'looks' and the only way to find them is by parsing the *hstore* column.

Also, the *restriction* class in the application is kind of messy. It could do with some remodeling, partly to clean up, and partly to make it more extensible to incorporate *conditions*. The *OSM* syntax for *conditional restrictions*[2] is shown in listing 6.1, and could work as a model for developing a more generic restrictions class.

```
<restriction-type>[:<transportation mode>][:<direction>]:conditional
  = <restriction-value> @ <condition>[;<restriction-value> @ <condition>]
```

Listing 6.1: Syntax of conditional restrictions in OpenStreetMap.

6.5 Documentation

A lot of the time of the project has also been devoted to documentation and writing this report. I took the opportunity to learn how to write a report in LaTeX, using the excellent web service www.sharelatex.com. It has been a pleasure, and it feels really good not to depend on the shaky features with cross-referencing in word processors.

6.6 Results

As the project does not fulfill all requirements and did not finish on time, it was not all that successful. The reason for not meeting the specification is that I ran out of time, partly due to the specification was supplied more than two weeks late, and partly due to the complexities with handling restrictions.

It would be possible to continue development, most on finding good ways to handle conditional restrictions. From my horizon, I still think that the best solution would be to adapt an existing solution that has been developed and refined by many people through many years. Perhaps using *OSRM* together with a *PostGIS* database as demonstrated here: https://www.mapbox.com/blog/osrm-using-external-data/. But I do not have any overview of the greater project, and what it is trying to accomplish.

This project shows that there exists really good products, and that rolling ones own is not trivial. What first seemed like a straightforward sequential piece of software turned out to be tangled in complex handling of restrictions.

[2] http://wiki.openstreetmap.org/wiki/Conditional_restrictions

Bibliography

[1] Keijo Ruohonen. *Graph Theory*. Lecture notes. Tampere University of Technology, 2013. URL: http://math.tut.fi/~ruohonen/GT_English.pdf (visited on 2015-03-24).

[2] Tero Harju. *Graph Theory*. Lecture notes. University of Turku, Departement of Mathematics, 2012. URL: http://users.utu.fi/harju/graphtheory/graphtheory.pdf (visited on 2015-04-09).

[3] Reinhard Diestel. *Graph Theory*. 4th Electronic Edition 2010, Corrected Reprint 2012. 2012. URL: http://diestel-graph-theory.com/ (visited on 2015-03-25).

[4] Daniel Delling et al. "Engineering Route Planning Algorithms". English. In: *Algorithmics of Large and Complex Networks*. Ed. by Jürgen Lerner, Dorothea Wagner, and KatharinaA. Zweig. Vol. 5515. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 117–139. ISBN: 978-3-642-02093-3. DOI: 10.1007/978-3-642-02094-0_7. URL: http://i11www.iti.uni-karlsruhe.de/extra/publications/dssw-erpa-09.pdf (visited on 2015-04-12).

[5] Hannah Bast et al. *Route Planning in Transportation Networks*. Tech. rep. MSR-TR-2014-4. Jan. 2014. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=207102 (visited on 2015-04-11).

[6] Daniel Delling et al. "Customizable Route Planning in Road Networks". working paper, submitted for publication. 2013. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=198358 (visited on 2015-04-05).

[7] Lars Volker. *Route Planning in Road Networks with Turn Costs*. Student research project, Universität Karlsruhe (TH), supervised by P. Sanders and D. Schultes. Universität Karlsruhe, 2008. URL: http://algo2.iti.kit.edu/documents/routeplanning/volker_sa.pdf (visited on 2015-04-03).

[8] Robert Geisberger and Christian Vetter. "Efficient Routing in Road Networks with Turn Costs". In: *Proc. 23rd International Symposium on Distributed Computing (DISC, Elche, Spain, September 2009)*. Vol. 5805. Lecture Notes in Computer Science. Springer. URL: http://algo2.iti.kit.edu/download/turn_ch.pdf (visited on 2015-04-13).

[9] Alexandros Efentakis et al. "Crowdsourcing turning restrictions for OpenStreetMap." In: *EDBT/ICDT Workshops*. 2014, pp. 355–362. URL: http://ceur-ws.org/Vol-1133/paper-56.pdf (visited on 2015-04-14).

[10] QGIS. *Topology*. URL: https://docs.qgis.org/2.2/en/docs/gentle_gis_introduction/topology.html (visited on 2015-10-05).

[11] Ittai Abraham et al. "HLDB: Location-based services in databases". In: *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*. ACM. 2012, pp. 339–348. URL: http://research-srv.microsoft.com/pubs/172429/hldb_GIS12.pdf (visited on 2015-04-14).

[12] Ittai Abraham et al. *HLDB: Location-Based Services in Databases*. Tech. rep. MSR-TR-2012-59. June 2012. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=166857 (visited on 2015-04-14).

[13] Doug Gregor. *Bundled Properties*. URL: http://www.boost.org/doc/libs/1_54_0/libs/graph/doc/bundles.html (visited on 2015-10-08).

[14] "Achadwik". *File:Length and time restriction 2.jpg*. [License: Creative Commons Attribution-ShareAlike 2.0]. 2011. URL: http://wiki.openstreetmap.org/wiki/File:Length_and_time_restriction_2.jpg (visited on 2015-10-15).

[15] OpenStreetMap contributors ©. *OpenStreetMap*. [License: Creative Commons Attribution-ShareAlike 2.0]. 2015. URL: https://www.openstreetmap.org/#map=18/45.13903/42.03324 (visited on 2015-10-22).

A Specification

The complete specifications from the company.

A.1 General

Line Graph Utility, LGU is a software utility which can poll a PostGIS database for a road network and builds a directed line graph from that. The directed line graph is stored in memory and the call to `get_directed_line_graph()` returns a directed line graph stored in a C++ *Boost* graph structure. The directed line graph is built based on the time of the day, road signs, traffic lights and other conditions.

A.2 Main use case

A.2.1 Call `get_directed_line_graph()` from C++ code.

A.2.2 LGU queries the PostGIS database and builds a graph from the road network.

A.2.3 LGU builds a directed line graph from the previous step (i.e. it converts nodes to edges and assigns weight to those edges based on road signs and other elements which are present in the nodes).

A.2.4 `get_directed_line_graph()` returns a directed line graph structure which is based on a C Boost graph structure to the function caller.

A.3 Optional use case

A.3.1 All main use case steps.

A.3.2 Write the resulting directed line graph to a separate heterogeneous table in the PostGIS database so that the graph can be viewed in QGis.

A.4 Functional requirements

A.4.1 LGU should take into account the following elements when building a directed line graph and calculating a weight for each edge: road signs (including time scheduling for those), traffic lights, road type (OSM road types), time of the day, road marking (i.e. separate lanes should be treated as separate edges), crossing and roundabouts slowdown, slopes and downhills, one way streets, road conditions, 'closed road' attribute.

A.4.2 LGU should take into account restricted turns in the road network when building a directed line graph; i.e. it should not create edges between newly created nodes in a line graph.

A.4.3 LGU should only take road signs and other conditions which are already present in the PostGIS database, the database is the only source of data for LGU.

A.4.4 LGU should store all its settings in a `settings.json` file.

A.5 Non-functional requirements

A.5.1 LGU should be written in C/C++; or, Boost.Python can be used

A.5.2 LGU can re-use architecture and code from the pgRouting software, which has a very similar structure. Namely it can re-use the steps 1 and 2 of the pgRouting's source code:

- A C module that uses a query is passed in Postgresql in order to build a line graph.
- C++ modules that convert it into a boost graph, and launch the routing.
- Return a result into psql server (this step is not required)

A.6 Testing requirements

LGU should be tested with a road network map built from 2 `.osm` files, `partille.osm` and `mikhailovsk.osm`.

A.7 Coding standard

Not actually written down in this document, but noted in an earlier conversation was that the company uses a *coding standard* [1] that must be followed.

[1] http://www.possibility.com/Cpp/CppCodingStandard.html

B UML Diagrams

Sequence and class diagrams of the software module.

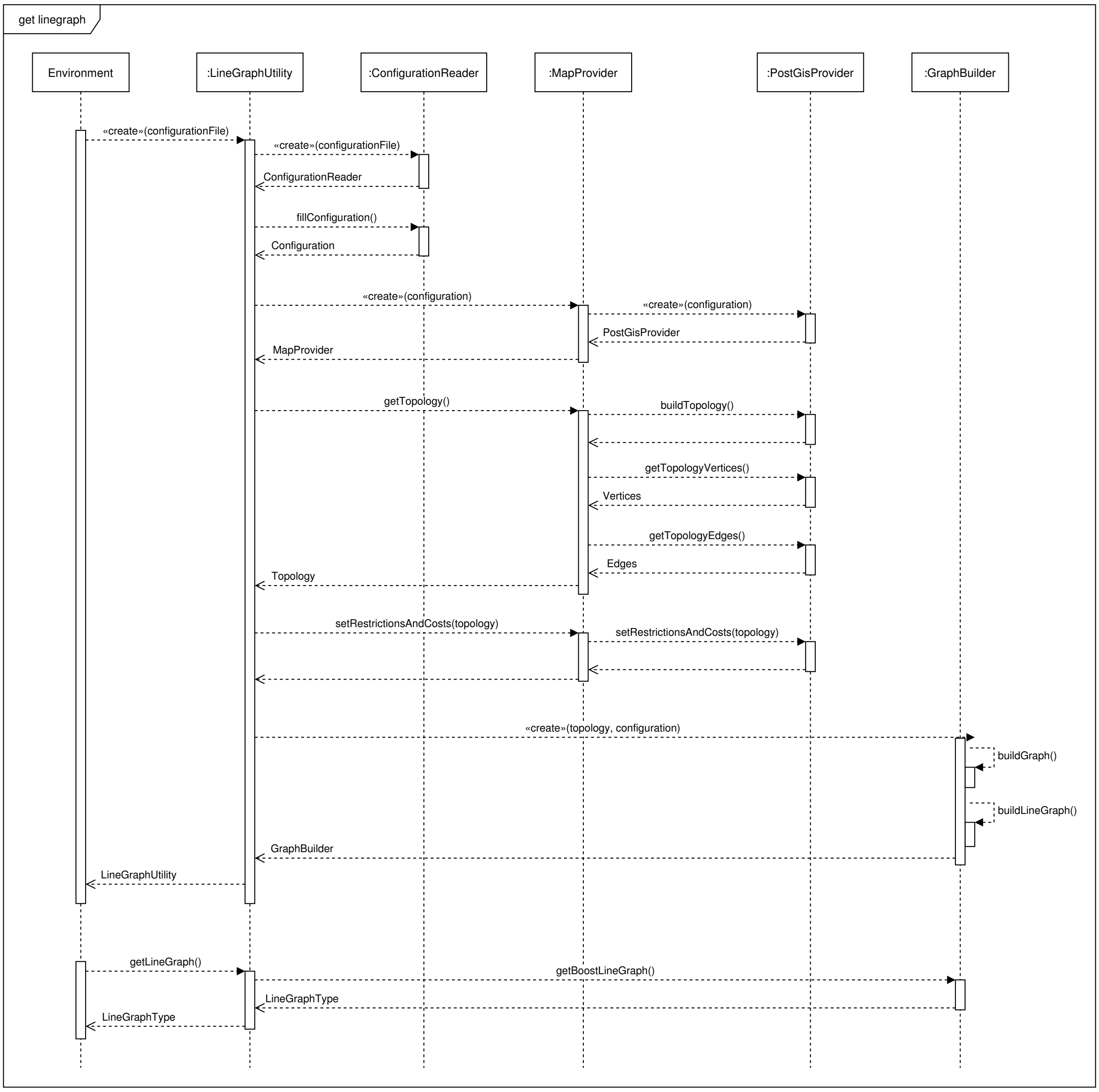


Figure B.1: Sequence diagram of main use case to get a line graph.

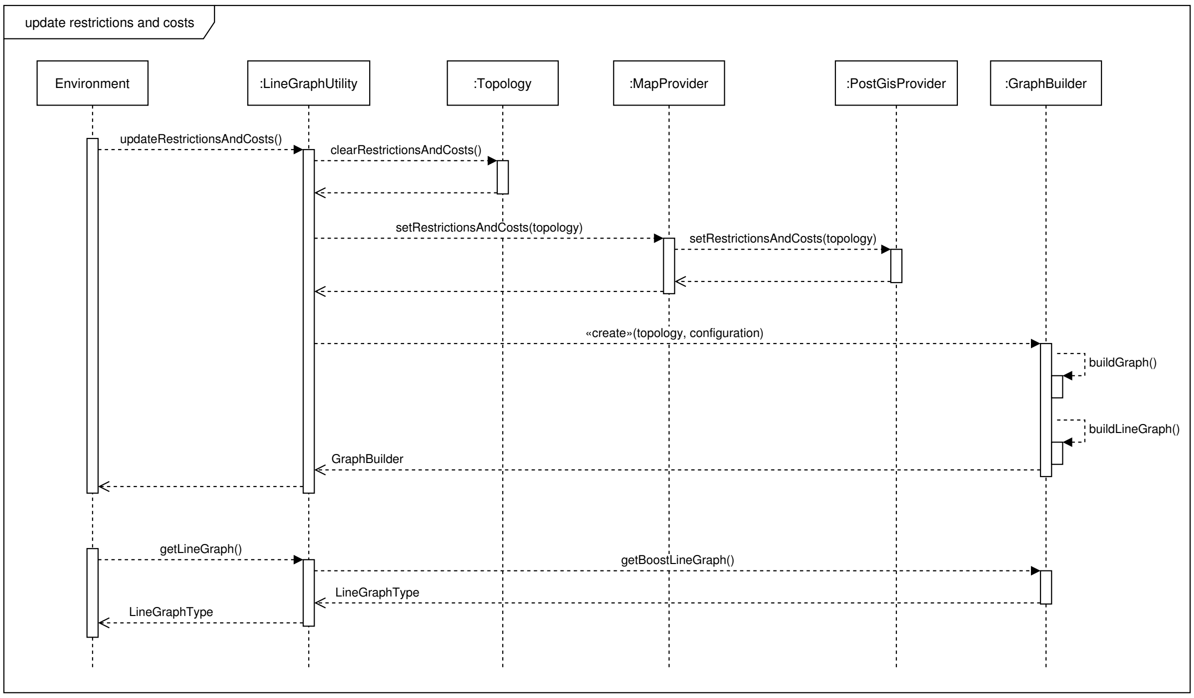


Figure B.2: Sequence diagram of updating costs and restrictions on a topology.

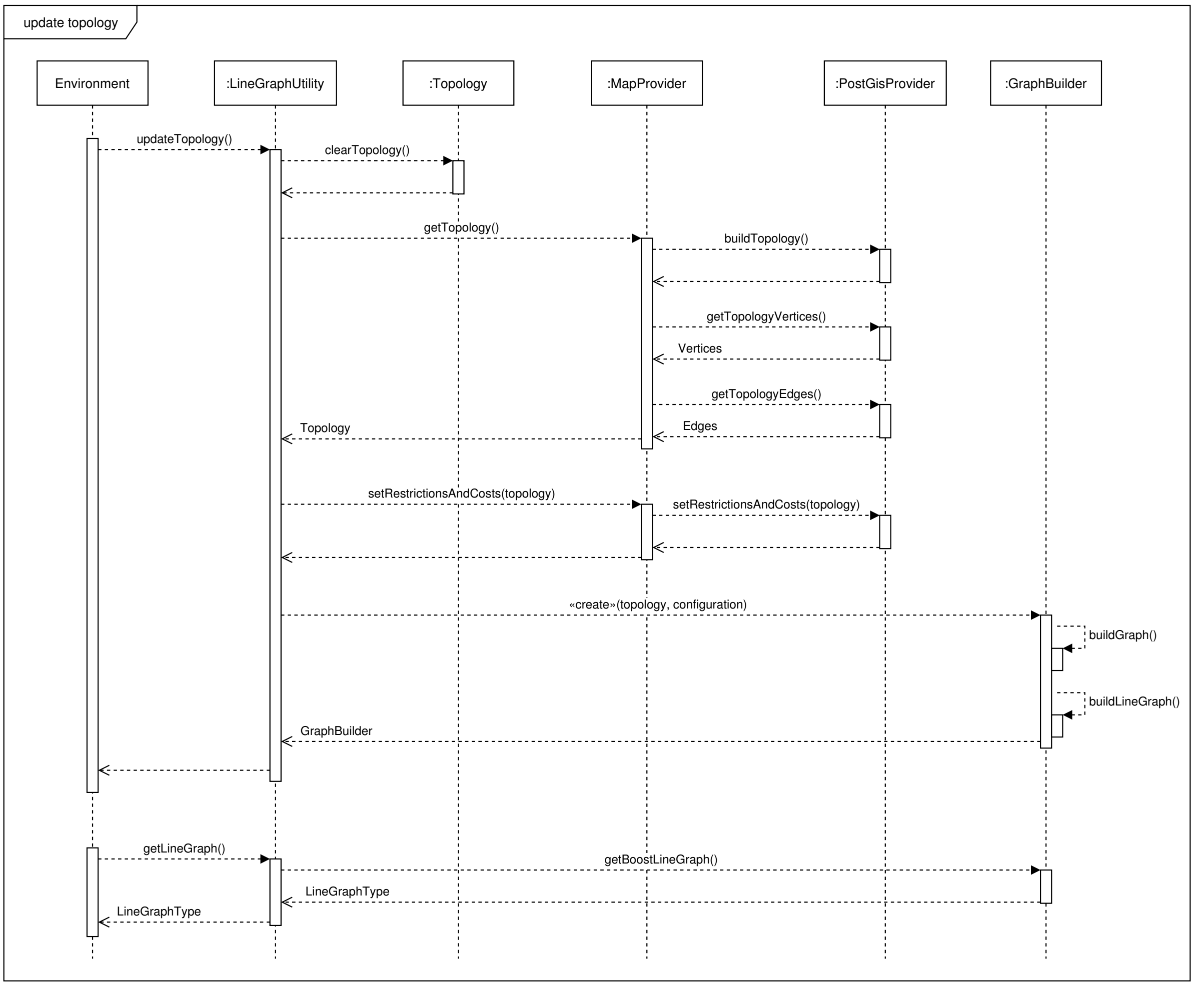


Figure B.3: Sequence diagram of updating the topology.

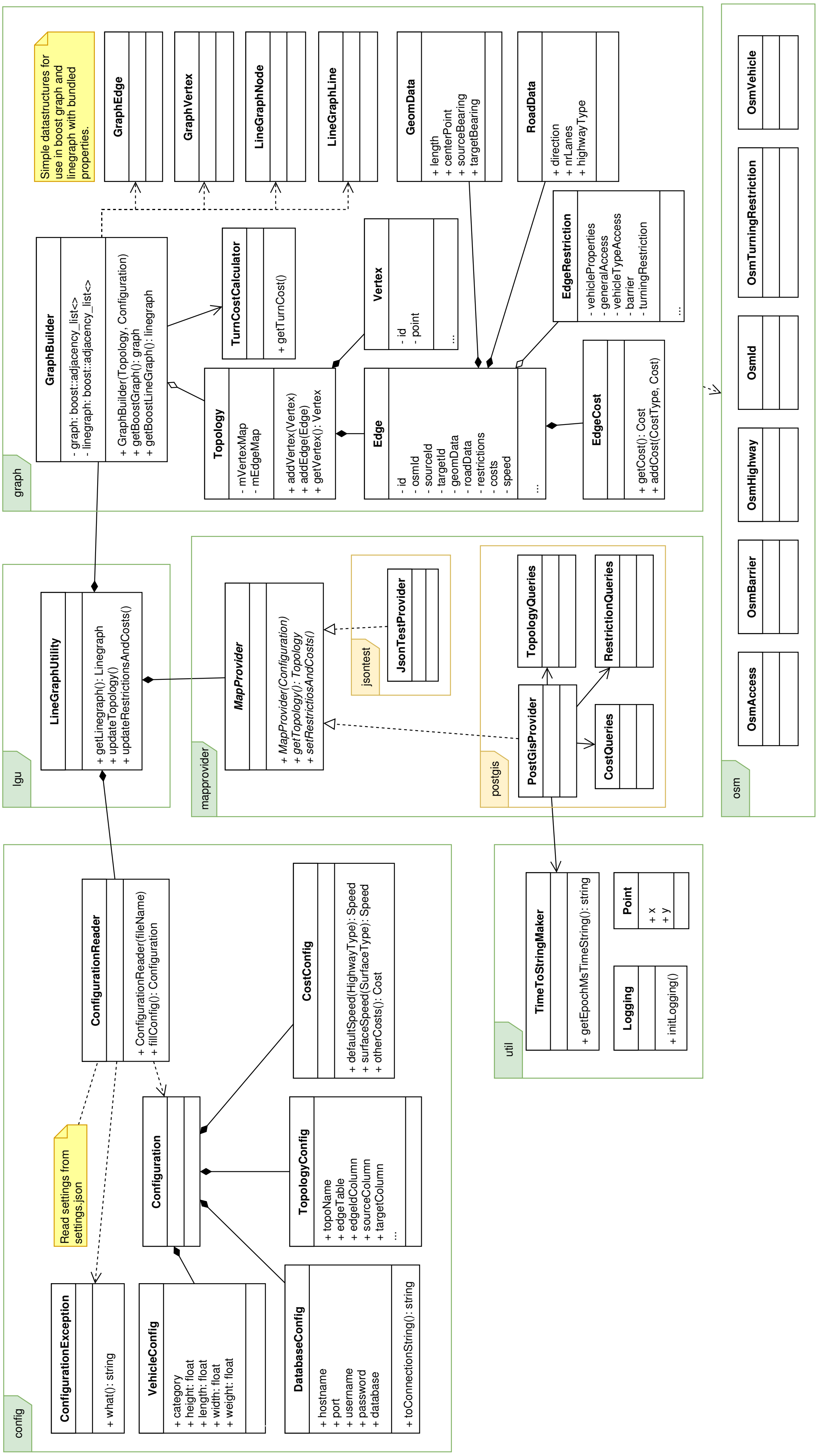


Figure B.4: Class diagram of the *line graph utility*.

C Directory listings

Contents of the directories in this project.

```
/
├── catchtest/
├── config/
├── doc/
├── graph/
├── lgu/
├── mapprovider/
├── osm/
├── preparation/
├── util/
└── README.md
```

Listing C.1: Directory structure in root of the project

```
/
└── catchtest/
    ├── catch.hpp
    ├── catchmain.cc
    ├── README.md
    └── testsettings/
        ├── mikhailovsk-original.json
        ├── mikhailovsk-original-temp.json
        ├── partille-original-temp.json
        ├── partille-original-temp.json
        ├── . . . .................................................. (10 more .json files)
        └── restrictions/
            ├── mikhailovsk-barrier_block.json
            ├── partille-highway_traffic_signals.json
            └── . . . .............................................. (17 more .json files)
```

Listing C.2: Files in /catchtest

```
/
└── config/
    ├── catchtest/
    │   └── ConfigurationReader_test.cc
    ├── Configuration.cc
    ├── Configuration.h
    ├── ConfigurationException.h
    ├── ConfigurationReader.cc
    ├── ConfigurationReader.h
    ├── CostConfig.h
    ├── DatabaseConfig.h
    ├── README.md
    ├── TopologyConfig.h
    └── VehicleConfig.h
```

Listing C.3: Files in `/config`

```
/
└── doc/
    ├── presentation/
    ├── report/
    │   ├── latex/ ........................................ LaTeXoriginal
    │   ├── jobe0900_report.pdf
    │   └── README.md
    ├── uml/
    │   ├── ........................ .xml, .svg, .pdf diagram files
    │   └── README.md
    └── README.md
```

Listing C.4: Files in `/doc`

```
/
└─ graph/
   ├─ catchtest/
   │  ├─ EdgeCost_test.cc
   │  ├─ GraphBuilder_test.cc
   │  ├─ RestrictionsAndCosts_test.cc
   │  ├─ Topology_test.cc
   │  └─ TurnCostCalculator_test.cc
   ├─ Cost.h
   ├─ Edge.cc
   ├─ Edge.h
   ├─ EdgeCost.cc
   ├─ EdgeCost.h
   ├─ EdgeRestriction.cc
   ├─ EdgeRestriction.h
   ├─ GraphBuilder.cc
   ├─ GraphBuilder.h
   ├─ GraphException.h
   ├─ README.md
   ├─ RestrictionsException.h
   ├─ Speed.h
   ├─ Topology.cc
   ├─ Topology.h
   ├─ TopologyException.h
   ├─ TurnCostCalculator.cc
   ├─ TurnCostCalculator.h
   ├─ Vertex.cc
   └─ Vertex.h
```

Listing C.5: Files in /graph

```
/
└─ lgu/
   ├─ catchtest/
   │  └─ LineGraphUtility_test.cc
   ├─ LineGraphUtility.cc
   ├─ LineGraphUtility.h
   ├─ LineGraphUtilityException.h
   └─ README.md
```

Listing C.6: Files in /lgu

```
/
└── mapprovider/
    ├── catchtest/
    │   └── MapProvider_test.cc
    ├── jsontest/
    │   ├── catchtest/
    │   │   ├── JsonTestProvider_test.cc
    │   │   ├── jsontest-settings.json
    │   │   └── test-topology.json
    │   ├── JsonTestProvider.cc
    │   ├── JsonTestProvider.h
    │   └── README.md
    ├── postgis/
    │   ├── catchtest/
    │   │   └── PostGisProvider_test.cc
    │   ├── CostQueries.cc
    │   ├── CostQueries.h
    │   ├── LineGraphSaveQueries.cc
    │   ├── LineGraphSaveQueries.h
    │   ├── PostGisProvider.cc
    │   ├── PostGisProvider.h
    │   ├── RestrictionQueries.cc
    │   ├── RestrictionQueries.h
    │   ├── TopologyQueries.cc
    │   └── README.md
    ├── MapProvider.h
    ├── MapProviderException.h
    └── README.md
```

Listing C.7: Files in /mapprovider

```
/
└─ osm/
   ├─ catchtest/
   │  ├─ OsmAccess_test.cc
   │  ├─ OsmBarrier_test.cc
   │  ├─ OsmHighway_test.cc
   │  ├─ OsmTurningRestriction_test.cc
   │  └─ OsmVehicle_test.cc
   ├─ OsmAccess.cc
   ├─ OsmAccess.h
   ├─ OsmBarrier.cc
   ├─ OsmBarrier.h
   ├─ OsmException.h
   ├─ OsmHighway.cc
   ├─ OsmHighway.h
   ├─ OsmId.cc
   ├─ OsmId.h
   ├─ OsmTurningRestriction.cc
   ├─ OsmTurningRestriction.h
   ├─ OsmVehicle.cc
   ├─ OsmVehicle.h
   └─ README.md
```

Listing C.8: Files in `/osm`

```
/
└─ preparation/
   ├─ restrictions/
   │  └─ . . . ............................. osm-files modified with added restrictions
   ├─ build_postgis_topology.sql
   ├─ init_osm2pgsql_postgis_topology.sql
   ├─ LGU.style
   ├─ mikhailovsk.osm
   ├─ partille.osm
   └─ README.md
```

Listing C.9: Files in `/preparation`

```
/
└─ util/
   ├─ catchtest/ .................................................... empty directory
   ├─ Logging.cc
   ├─ Logging.h
   ├─ Point.h
   ├─ TimeToStringMaker.cc
   └─ TimeToStringMaker.h
```

Listing C.10: Files in `/util`

D Source code

A complete repository can be found at:
https://bitbucket.org/jobe0900/exjobb/src or
https://github.com/jobe0900/exjobb.

D.1 README.md

```
LineGraphUtility (lgu)
======================
This software module uses OpenStreetMap data to fetch topology, restrictions and
 ↪  costs, and uses them to build a Graph, which is converted to a LineGraph.

## State of the software
The software module does not fulfill the specification yet.

#### Working features
- Building graph and linegraph respecting some **edge** restrictions:
    - Turning restrictions.
    - General access restrictions.
    - Vehicle type specific restrictions.
    - Vehicle property restrictions (weight, height...).
- Some restrictions on edges.
- Turning restrictions via a node, not via other roads.
- Costs.

#### NOT implemented features
- Inclination, different speed uphill or down hill.
- Conditional restrictions.
- Turning restrictions that are not one-to-one, but one-to-many.
- Turning restrictions via ways (not via nodes).
- Parsing units, i.e. assuming all dimensions are meters and weight in metric
 ↪  tons and speed in km/h.

### Organization
The code is organized in folders (kind of "packages") to keep it modularized. The
 ↪  packages are:

- **`catchtest`**: The main for the testing framework.
- **`config`**: For configuration related code.
- **`graph`**: For code that is related to Graphs.
- **`lgu`**: The main entry point into this software.
- **`mapprovider`**: The package for code providing access to map data.
- **`osm`**: Classes representing some concepts in OpenStreetMap data.
- **`preparation`**: osm-files and sql-files and instructions on how to set up
 ↪  database.
- **`uml`**: For uml documentation.
- **`util`**: A few utility classes.

Each folder should have its own `README.md` that describes what the contents and
 ↪  the purpose of that package is. Each package should also have their own tests
 ↪  in a `catchtest` folder, and preferably an *exception class*.

### Building
```

```
Right now  all development has been in Eclipse, so it is just a standard Eclipse
 ↪  project with the makefiles that Eclipse has set up in the `Debug` folder. The
 ↪  file `catchtest/catchmain.cc` provides the entry point for the software
 ↪  module during testing.

#### Libraries
There was only need for linking with `-lpqxx` and `-lpq` (for connecting to the
 ↪  database) until *Boost logging* was included, at which point it also became
 ↪  necessary to link with `-lboost_log -lboost_log_setup -lboost_thread
 ↪  -lboost_system -lpthread`.

#### Testing
As mentioned, testing is done with [Catch](https://github.com/philsquared/Catch).
 ↪  Tests can be written BDD-style, and it is header only. A few quirks: some of
 ↪  the macro-keywords, most notably `REQUIRE`, is reported as an error in the
 ↪  Eclipse editor, but one can ignore that.

### Style
I have tried to follow the style given in [C++ Coding
 ↪  Standard](http://www.possibility.com/Cpp/CppCodingStandard.html).

### Design
I have deliberately tried to avoid passing pointers around, and rather pass in
 ↪  references as IN-OUT parameters. The idea is that the central LGU class has
 ↪  stack variables of `Graph`, `Topology` that gets filled in, rather than
 ↪  obtained as pointers to objects on the heap. This is to try to reduce risks
 ↪  of complicated memory handling, while not have too much copying of large
 ↪  objects.

### Logging
Boost logging was the last feature added, and is so far only used in the `Graph`
 ↪  class. It needs to be compiled and linked with a lot of libraries:

    -lboost_log -lboost_log_setup -lboost_thread -lboost_system -lpthread

The log produced is `lgu.log` in the top level of the project.
```

D.2 catchtest

D.2.1 README.md

```
CATCH
=====
This project uses [Catch](https://github.com/philsquared/Catch) for testing. It
 ↪  allows for writing tests BDD-style.

It doesn't play really nicely with Eclipse, as Eclipse's editor marks `REQUIRE`
 ↪  as errors, so the project has a  lot of error markers throughout, without any
 ↪  real errors. But the Catch way of testing is nice, so it is worth is. And
 ↪  Eclipse flags a lot of errors for standard c++11 features as well...

When writing `SCENARIO`s or `TESTCASE`s one can tag those, which makes it easy to
 ↪  test small parts of the code. After building you can modify the Eclipse `Run
 ↪  Configuration` (or write on the command line) to only run those tests.

Example:
```

````
```cpp
SCENARIO ("Testing this module but not other", "[moduletag]")
{
    GIVEN ("a")
    {
        WHEN ("b")
        {
            THEN ("c")
            {
                REQUIRE (c)
            }
        }
    }
}
```

To specify which test to run, go to `Run` > `Run Configurations...`, select the `Arguments` tab and in `Program arguments` write the tag, e.g. `[moduletag]`, click `Apply` and `Run`.

A useful flag to add to the program arguments when running tests is `-s` to have print out of every step, else you only get the final report of how many scenarios have run.
````

D.2.2 catchmain.cc

```cpp
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
```

D.2.3 mikhailovsk-original.json

```json
{
    "database":
    {
        "host":         "127.0.0.1",
        "port":          5432,
        "username":     "tester",
        "password":     "tester",
        "database":     "mikhailovsk-original"
    },

    "topology":
    {
        "provider":       "postgis",

        "postgis":
        {

            "topo_name":      "lgu",
            "roads_prefix":   "highways",
            "schema_prefix":  "topo",
            "build": {
                "temp_topo_name":  "",
                "srid":            900913,
                "tolerance":       1.0
            },
            "edge":
```

```
             {
                 "table":      "edge_data",
                 "id_col":     "edge_id",
                 "source_col": "start_node",
                 "target_col": "end_node",
                 "geom_col":   "geom"
             },
             "vertex":
             {
                 "table":     "node",
                 "id_col":    "node_id",
                 "geom_col":  "geom"
             }
         },

         "pgrouting":
         {
         },

         "jsontest":
         {
             "test_file": ""
         }

     },

     "vehicle":
     {
         "category":   "motorcar",
         "motorcar":
         {
            "height":       1.6,
            "length":       4.5,
            "width":        1.9,
            "weight":       2.0,
            "maxspeed":     200,
            "acceleration": 10,
            "deceleration": 7
         }
     },

         "access":
    {
       "allow":
       [
          "yes",
          "permissive",
          "designated"
       ]
    },

    "restrict":
    {
       "barriers":
       [
          "block",
          "bollard",
```

```
            "bus_trap",
            "chain",
            "cycle_barrier",
            "debris",
            "full-height_turnstile",
            "horse_stile",
            "jersey_barrier",
            "kent_carriage_gap",
            "kissing_gate",
            "log",
            "motorcycle_barrier",
            "rope",
            "sally_port",
            "spikes",
            "stile",
            "sump_buster",
            "swing_gate",
            "turnstile",
            "yes"
        ]

    },

    "cost":
    {
        "default_speed":
        {
            "motorway":       {"high": 110, "low":  90},
            "motorway_link":  {"high":  90, "low":  90},
            "trunk":          {"high":  90, "low":  60},
            "trunk_link":     {"high":  90, "low":  60},
            "primary":        {"high":  90, "low":  60},
            "primary_link":   {"high":  90, "low":  60},
            "secondary":      {"high":  90, "low":  60},
            "secondary_link":{"high":  90, "low":  60},
            "tertiary":       {"high":  90, "low":  60},
            "tertiary_link":  {"high":  90, "low":  60},
            "unclassified":   {"high":  90, "low":  60},
            "residential":    {"high":  90, "low":  60},
            "service":        {"high":  40, "low":  20},
            "living_street":  {"high":  20, "low":  20},
            "bus_guideway":   {"high":  80, "low":  60},
            "road":           {"high":  80, "low":  50}
        },

        "surface":
        {
            "paved":              1000,
            "asphalt":            1000,
            "cobblestone":        20,
            "cobblestone:flattened": 40,
            "sett":               40,
            "concrete":           1000,
            "concrete:lanes":     40,
            "concrete:plates":    100,
            "paving_stones":      40,
            "metal":              60,
```

```
            "wood":             30,
            "unpaved":          60,
            "compacted":        70,
            "dirt":             40,
            "earth":            40,
            "fine_gravel":      50,
            "grass":            10,
            "grass_paver":      20,
            "gravel":           60,
            "ground":           20,
            "ice":              70,
            "mud":              5,
            "pebblestone":      50,
            "salt":             70,
            "sand":             70,
            "snow":             50,
            "woodchips":        5,
            "metal_grid":       40
        },

        "barriers":
        [
            ["border_control",    120],
            ["bump_gate",          30],
            ["bus_trap",           30],
            ["cattle_grid",        20],
            ["entrance",           10],
            ["gate",               30],
            ["hampshire_gate",     60],
            ["height_restrictor",   5],
            ["jersey_barrier",     10],
            ["lift_gate",          60],
            ["sump_buster",        30],
            ["swing_gate",         60],
            ["toll_booth",         40]
        ],

        "highway":
        [
            ["bus_stop",            5],
            ["crossing",            5],
            ["give_way",           20],
            ["mini_roundabout",    20],
            ["stop",               30],
            ["traffic_signals",    30]
        ],

        "railway":
        [
            ["level_crossing",     20]
        ],

        "public_transport":
        [
            ["stop_position",       5]
        ],
```

```
        "traffic_calming":
        [
            ["yes",                 10],
            ["bump",                10],
            ["hump",                10],
            ["table",               10],
            ["cushion",             10],
            ["rumble_strip",        10],
            ["chicane",             10],
            ["choker",              10],
            ["island",               5]
        ]

    }
}
```

D.2.4 mikhailovsk-original-temp.json

```
{
    "database":
    {
        "host":         "127.0.0.1",
        "port":          5432,
        "username":     "tester",
        "password":     "tester",
        "database":     "mikhailovsk-original-temp"
    },

    "topology":
    {
        "provider":       "postgis",

        "postgis":
        {

            "topo_name":      "lgu",
            "roads_prefix":   "highways",
            "schema_prefix":  "topo",
            "build": {
                "temp_topo_name": "epoch_ms",
                "srid":           900913,
                "tolerance":      1.0
            },
            "edge":
            {
                "table":      "edge_data",
                "id_col":     "edge_id",
                "source_col": "start_node",
                "target_col": "end_node",
                "geom_col":   "geom"
            },
            "vertex":
            {
                "table":     "node",
                "id_col":    "node_id",
                "geom_col":  "geom"
            }
```

```
        },

        "pgrouting":
        {
        },

        "jsontest":
        {
            "test_file": ""
        }

    },

    "vehicle":
    {
        "category":   "motorcar",
        "motorcar":
        {
           "height":       1.6,
           "length":       4.5,
           "width":        1.9,
           "weight":       2.0,
           "maxspeed":     200,
           "acceleration": 10,
           "deceleration": 7
        }
    },

   "access":
   {
      "allow":
      [
         "yes",
         "permissive",
         "designated"
      ]
   },

   "restrict":
   {
      "barriers":
      [
         "block",
         "bollard",
         "bus_trap",
         "chain",
         "cycle_barrier",
         "debris",
         "full-height_turnstile",
         "horse_stile",
         "jersey_barrier",
         "kent_carriage_gap",
         "kissing_gate",
         "log",
         "motorcycle_barrier",
         "rope",
         "sally_port",
```

```
            "spikes",
            "stile",
            "sump_buster",
            "swing_gate",
            "turnstile",
            "yes"
        ]

    },

    "cost":
    {
        "default_speed":
        {
            "motorway":      {"high": 110, "low":  90},
            "motorway_link": {"high":  90, "low":  90},
            "trunk":         {"high":  90, "low":  60},
            "trunk_link":    {"high":  90, "low":  60},
            "primary":       {"high":  90, "low":  60},
            "primary_link":  {"high":  90, "low":  60},
            "secondary":     {"high":  90, "low":  60},
            "secondary_link":{"high":  90, "low":  60},
            "tertiary":      {"high":  90, "low":  60},
            "tertiary_link": {"high":  90, "low":  60},
            "unclassified":  {"high":  90, "low":  60},
            "residential":   {"high":  90, "low":  60},
            "service":       {"high":  40, "low":  20},
            "living_street": {"high":  20, "low":  20},
            "bus_guideway":  {"high":  80, "low":  60},
            "road":          {"high":  80, "low":  50}
        },

        "surface":
        {
            "paved":              1000,
            "asphalt":            1000,
            "cobblestone":        20,
            "cobblestone:flattened": 40,
            "sett":               40,
            "concrete":           1000,
            "concrete:lanes":     40,
            "concrete:plates":    100,
            "paving_stones":      40,
            "metal":              60,
            "wood":               30,
            "unpaved":            60,
            "compacted":          70,
            "dirt":               40,
            "earth":              40,
            "fine_gravel":        50,
            "grass":              10,
            "grass_paver":        20,
            "gravel":             60,
            "ground":             20,
            "ice":                70,
            "mud":                5,
            "pebblestone":        50,
```

```
          "salt":            70,
          "sand":            70,
          "snow":            50,
          "woodchips":       5,
          "metal_grid":      40
       },

       "barriers":
       [
          ["border_control",   120],
          ["bump_gate",         30],
          ["bus_trap",          30],
          ["cattle_grid",       20],
          ["entrance",          10],
          ["gate",              30],
          ["hampshire_gate",    60],
          ["height_restrictor",  5],
          ["jersey_barrier",    10],
          ["lift_gate",         60],
          ["sump_buster",       30],
          ["swing_gate",        60],
          ["toll_booth",        40]
       ],

       "highway":
       [
          ["bus_stop",           5],
          ["crossing",           5],
          ["give_way",          20],
          ["mini_roundabout",   20],
          ["stop",              30],
          ["traffic_signals",   30]
       ],

       "railway":
       [
          ["level_crossing",    20]
       ],

       "public_transport":
       [
          ["stop_position",      5]
       ],

       "traffic_calming":
       [
          ["yes",               10],
          ["bump",              10],
          ["hump",              10],
          ["table",             10],
          ["cushion",           10],
          ["rumble_strip",      10],
          ["chicane",           10],
          ["choker",            10],
          ["island",             5]
       ]
    }
```

```
}
```

D.2.5 partille-original.json

```json
{
    "database":
    {
        "host":          "127.0.0.1",
        "port":           5432,
        "username":      "tester",
        "password":      "tester",
        "database":      "partille-original"
    },

    "topology":
    {
        "provider":        "postgis",

        "postgis":
        {

            "topo_name":       "lgu",
            "roads_prefix":    "highways",
            "schema_prefix":   "topo",
            "build": {
                "temp_topo_name": "",
                "srid":             900913,
                "tolerance":        1.0
            },
            "edge":
            {
                "table":       "edge_data",
                "id_col":      "edge_id",
                "source_col": "start_node",
                "target_col": "end_node",
                "geom_col":    "geom"
            },
            "vertex":
            {
                "table":      "node",
                "id_col":     "node_id",
                "geom_col":   "geom"
            }
        },

        "pgrouting":
        {
        },

        "jsontest":
        {
            "test_file": ""
        }

    },

    "vehicle":
```

```
     {
         "category":   "motorcar",
         "motorcar":
         {
            "height":       1.6,
            "length":       4.5,
            "width":        1.9,
            "weight":       2.0,
            "maxspeed":     200,
            "acceleration": 10,
            "deceleration": 7
         }
     },

    "access":
    {
       "allow":
       [
          "yes",
          "permissive",
          "designated"
       ]
    },

    "restrict":
    {
       "barriers":
       [
          "block",
          "bollard",
          "bus_trap",
          "chain",
          "cycle_barrier",
          "debris",
          "full-height_turnstile",
          "horse_stile",
          "jersey_barrier",
          "kent_carriage_gap",
          "kissing_gate",
          "log",
          "motorcycle_barrier",
          "rope",
          "sally_port",
          "spikes",
          "stile",
          "sump_buster",
          "swing_gate",
          "turnstile",
          "yes"
       ]

    },

    "cost":
    {
       "default_speed":
       {
```

```
            "motorway":      {"high": 110, "low":  90},
            "motorway_link": {"high":  90, "low":  90},
            "trunk":         {"high":  90, "low":  60},
            "trunk_link":    {"high":  90, "low":  60},
            "primary":       {"high":  90, "low":  60},
            "primary_link":  {"high":  90, "low":  60},
            "secondary":     {"high":  90, "low":  60},
            "secondary_link":{"high":  90, "low":  60},
            "tertiary":      {"high":  90, "low":  60},
            "tertiary_link": {"high":  90, "low":  60},
            "unclassified":  {"high":  90, "low":  60},
            "residential":   {"high":  90, "low":  60},
            "service":       {"high":  40, "low":  20},
            "living_street": {"high":  20, "low":  20},
            "bus_guideway":  {"high":  80, "low":  60},
            "road":          {"high":  80, "low":  50}
        },

        "surface":
        {
           "paved":             1000,
           "asphalt":           1000,
           "cobblestone":       20,
           "cobblestone:flattened": 40,
           "sett":              40,
           "concrete":          1000,
           "concrete:lanes":    40,
           "concrete:plates":   100,
           "paving_stones":     40,
           "metal":             60,
           "wood":              30,
           "unpaved":           60,
           "compacted":         70,
           "dirt":              40,
           "earth":             40,
           "fine_gravel":       50,
           "grass":             10,
           "grass_paver":       20,
           "gravel":            60,
           "ground":            20,
           "ice":               70,
           "mud":               5,
           "pebblestone":       50,
           "salt":              70,
           "sand":              70,
           "snow":              50,
           "woodchips":         5,
           "metal_grid":        40
        },

        "barriers":
        [
           ["border_control",    120],
           ["bump_gate",          30],
           ["bus_trap",           30],
           ["cattle_grid",        20],
           ["entrance",           10],
```

```
            ["gate",              30],
            ["hampshire_gate",    60],
            ["height_restrictor",  5],
            ["jersey_barrier",    10],
            ["lift_gate",         60],
            ["sump_buster",       30],
            ["swing_gate",        60],
            ["toll_booth",        40]
        ],

        "highway":
        [
            ["bus_stop",           5],
            ["crossing",           5],
            ["give_way",          20],
            ["mini_roundabout",   20],
            ["stop",              30],
            ["traffic_signals",   30]
        ],

        "railway":
        [
            ["level_crossing",    20]
        ],

        "public_transport":
        [
            ["stop_position",      5]
        ],

        "traffic_calming":
        [
            ["yes",               10],
            ["bump",              10],
            ["hump",              10],
            ["table",             10],
            ["cushion",           10],
            ["rumble_strip",      10],
            ["chicane",           10],
            ["choker",            10],
            ["island",             5]
        ]
    }
}
```

D.2.6 partille-original-temp.json

```
{
    "database":
    {
        "host":         "127.0.0.1",
        "port":          5432,
        "username":     "tester",
        "password":     "tester",
        "database":     "partille-original-temp"
    },

```

```
    "topology":
    {
        "provider":      "postgis",

        "postgis":
        {

            "topo_name":      "lgu",
            "roads_prefix":   "highways",
            "schema_prefix":  "topo",
            "build": {
                "temp_topo_name": "epoch_ms",
                "srid":           900913,
                "tolerance":      1.0
            },
            "edge":
            {
                "table":      "edge_data",
                "id_col":     "edge_id",
                "source_col": "start_node",
                "target_col": "end_node",
                "geom_col":   "geom"
            },
            "vertex":
            {
                "table":     "node",
                "id_col":    "node_id",
                "geom_col":  "geom"
            }
        },

        "pgrouting":
        {
        },

        "jsontest":
        {
            "test_file": ""
        }

    },

    "vehicle":
    {
        "category":   "motorcar",
        "motorcar":
        {
          "height":       1.6,
          "length":       4.5,
          "width":        1.9,
          "weight":       2.0,
          "maxspeed":     200,
          "acceleration": 10,
          "deceleration": 7
        }
    },

```

```
    "access":
    {
        "allow":
        [
            "yes",
            "permissive",
            "designated"
        ]
    },

    "restrict":
    {
        "barriers":
        [
            "block",
            "bollard",
            "bus_trap",
            "chain",
            "cycle_barrier",
            "debris",
            "full-height_turnstile",
            "horse_stile",
            "jersey_barrier",
            "kent_carriage_gap",
            "kissing_gate",
            "log",
            "motorcycle_barrier",
            "rope",
            "sally_port",
            "spikes",
            "stile",
            "sump_buster",
            "swing_gate",
            "turnstile",
            "yes"
        ]

    },

    "cost":
    {
        "default_speed":
        {
            "motorway":      {"high": 110, "low":  90},
            "motorway_link": {"high":  90, "low":  90},
            "trunk":         {"high":  90, "low":  60},
            "trunk_link":    {"high":  90, "low":  60},
            "primary":       {"high":  90, "low":  60},
            "primary_link":  {"high":  90, "low":  60},
            "secondary":     {"high":  90, "low":  60},
            "secondary_link":{"high":  90, "low":  60},
            "tertiary":      {"high":  90, "low":  60},
            "tertiary_link": {"high":  90, "low":  60},
            "unclassified":  {"high":  90, "low":  60},
            "residential":   {"high":  90, "low":  60},
            "service":       {"high":  40, "low":  20},
            "living_street": {"high":  20, "low":  20},
```

```
            "bus_guideway":  {"high":  80, "low":  60},
            "road":          {"high":  80, "low":  50}
        },

        "surface":
        {
           "paved":              1000,
           "asphalt":            1000,
           "cobblestone":        20,
           "cobblestone:flattened": 40,
           "sett":               40,
           "concrete":           1000,
           "concrete:lanes":     40,
           "concrete:plates":    100,
           "paving_stones":      40,
           "metal":              60,
           "wood":               30,
           "unpaved":            60,
           "compacted":          70,
           "dirt":               40,
           "earth":              40,
           "fine_gravel":        50,
           "grass":              10,
           "grass_paver":        20,
           "gravel":             60,
           "ground":             20,
           "ice":                70,
           "mud":                5,
           "pebblestone":        50,
           "salt":               70,
           "sand":               70,
           "snow":               50,
           "woodchips":          5,
           "metal_grid":         40
        },

        "barriers":
        [
           ["border_control",     120],
           ["bump_gate",           30],
           ["bus_trap",            30],
           ["cattle_grid",         20],
           ["entrance",            10],
           ["gate",                30],
           ["hampshire_gate",      60],
           ["height_restrictor",    5],
           ["jersey_barrier",      10],
           ["lift_gate",           60],
           ["sump_buster",         30],
           ["swing_gate",          60],
           ["toll_booth",          40]
        ],

        "highway":
        [
           ["bus_stop",             5],
           ["crossing",             5],
```

```
            ["give_way",           20],
            ["mini_roundabout",    20],
            ["stop",               30],
            ["traffic_signals",    30]
        ],

        "railway":
        [
            ["level_crossing",     20]
        ],

        "public_transport":
        [
            ["stop_position",       5]
        ],

        "traffic_calming":
        [
            ["yes",                10],
            ["bump",               10],
            ["hump",               10],
            ["table",              10],
            ["cushion",            10],
            ["rumble_strip",       10],
            ["chicane",            10],
            ["choker",             10],
            ["island",              5]
        ]
    }
}
```

D.2.7 mikhailovsk-barrier_block.json

```
{
    "database":
    {
        "host":         "127.0.0.1",
        "port":          5432,
        "username":     "tester",
        "password":     "tester",
        "database":     "mikhailovsk-barrier_block"
    },

    "topology":
    {
        "provider":       "postgis",

        "postgis":
        {

            "topo_name":      "lgu",
            "roads_prefix":   "highways",
            "schema_prefix":  "topo",
            "build": {
                "temp_topo_name": "",
                "srid":           900913,
                "tolerance":      1.0
```

```
            },
            "edge":
            {
                "table":      "edge_data",
                "id_col":     "edge_id",
                "source_col": "start_node",
                "target_col": "end_node",
                "geom_col":   "geom"
            },
            "vertex":
            {
                "table":     "node",
                "id_col":    "node_id",
                "geom_col":  "geom"
            }
        },

        "pgrouting":
        {
        },

        "jsontest":
        {
            "test_file": ""
        }

    },

    "vehicle":
    {
        "category":   "motorcar",
        "motorcar":
        {
           "height":       1.6,
           "length":       4.5,
           "width":        1.9,
           "weight":       2.0,
           "maxspeed":     200,
           "acceleration": 10,
           "deceleration": 7
        }
    },

    "access":
   {
      "allow":
      [
         "yes",
         "permissive",
         "designated"
      ]
   },

   "restrict":
   {
      "barriers":
      [
```

```
            "block",
            "bollard",
            "bus_trap",
            "chain",
            "cycle_barrier",
            "debris",
            "full-height_turnstile",
            "horse_stile",
            "jersey_barrier",
            "kent_carriage_gap",
            "kissing_gate",
            "log",
            "motorcycle_barrier",
            "rope",
            "sally_port",
            "spikes",
            "stile",
            "sump_buster",
            "swing_gate",
            "turnstile",
            "yes"
         ]

      },

      "cost":
      {
         "default_speed":
         {
             "motorway":       {"high": 110, "low":  90},
             "motorway_link": {"high":  90, "low":  90},
             "trunk":          {"high":  90, "low":  60},
             "trunk_link":     {"high":  90, "low":  60},
             "primary":        {"high":  90, "low":  60},
             "primary_link":  {"high":  90, "low":  60},
             "secondary":      {"high":  90, "low":  60},
             "secondary_link":{"high":  90, "low":  60},
             "tertiary":       {"high":  90, "low":  60},
             "tertiary_link": {"high":  90, "low":  60},
             "unclassified":  {"high":  90, "low":  60},
             "residential":   {"high":  90, "low":  60},
             "service":        {"high":  40, "low":  20},
             "living_street": {"high":  20, "low":  20},
             "bus_guideway":  {"high":  80, "low":  60},
             "road":           {"high":  80, "low":  50}
         },

         "surface":
         {
            "paved":              1000,
            "asphalt":            1000,
            "cobblestone":        20,
            "cobblestone:flattened": 40,
            "sett":               40,
            "concrete":           1000,
            "concrete:lanes":   40,
            "concrete:plates": 100,
```

```
            "paving_stones":   40,
            "metal":           60,
            "wood":            30,
            "unpaved":         60,
            "compacted":       70,
            "dirt":            40,
            "earth":           40,
            "fine_gravel":     50,
            "grass":           10,
            "grass_paver":     20,
            "gravel":          60,
            "ground":          20,
            "ice":             70,
            "mud":             5,
            "pebblestone":     50,
            "salt":            70,
            "sand":            70,
            "snow":            50,
            "woodchips":       5,
            "metal_grid":      40
        },

        "barriers":
        [
            ["border_control",    120],
            ["bump_gate",          30],
            ["bus_trap",           30],
            ["cattle_grid",        20],
            ["entrance",           10],
            ["gate",               30],
            ["hampshire_gate",     60],
            ["height_restrictor",   5],
            ["jersey_barrier",     10],
            ["lift_gate",          60],
            ["sump_buster",        30],
            ["swing_gate",         60],
            ["toll_booth",         40]
        ],

        "highway":
        [
            ["bus_stop",            5],
            ["crossing",            5],
            ["give_way",           20],
            ["mini_roundabout",    20],
            ["stop",               30],
            ["traffic_signals",    30]
        ],

        "railway":
        [
            ["level_crossing",     20]
        ],

        "public_transport":
        [
            ["stop_position",       5]
```

```
        ],

        "traffic_calming":
        [
            ["yes",              10],
            ["bump",             10],
            ["hump",             10],
            ["table",            10],
            ["cushion",          10],
            ["rumble_strip",     10],
            ["chicane",          10],
            ["choker",           10],
            ["island",            5]
        ]

    }
}
```

D.2.8 partille-highway_traffic_signals.json

D.3 config

D.3.1 README.md

```
Configuration
=============

Configurations are set in the file `settings.json`. The different parts of the
↪ configuration is:

- Database
- Topology
- Vehicle
- Access
- Restrictions
- Costs

If one wishes to edit the setting, one can freely add and remove objects in
↪ _braces_ (`[` and `]`), for example if one wishes to edit which values for a
↪ tag inflicts a cost. But the strings in _curly braces_ (`{` and `}`) are keys
↪ that must exist, but the values for the keys can of course be edited.


Database
--------

Configuration for connecting to the database holding map data. The expected keys
↪ and values are:

- **"host"**:
    - *hostname* (e.g. `"localhost"`) or
    - *ip-address* (e.g. `"127.0.0.1"`).
- **"port"**:
    - *portnumber* (e.g. `5432`).
- **"username"**:
    - *username* (e.g. `"tester"`).
- **"password"**:
```

```
    - *password* (e.g. `"tester_pass"`).
- **"database"**:
    - *database name* (e.g. `"db_name"`).


Topology
--------

Configurations for building or reading topology from a database. Might have different meanings depending on which *MapProvider* are used. Topologies can be pre-built, or they can be generated each time, depending on the settings in `"build_topo"`. It is also possible to define a simple json test file, for testing simple topologies.

- **"provider"**:
    - *name* of *MapProvider*
        - `"postgis"` when using `postgis_topology` for building topologies.
        - `"pgrouting"` when using `pgrouting` for building topologies.
        - `"jsontest"` for simple json test topology.

- **"postgis"**:
    - **"topo_name"**:
        - *basename* for pre-built topologies (e.g. `"test"`), combined with `roads_prefix` and `topo_prefix` for actual names such as `"highways_test"` and `"topo_test"`.

    - **"roads_prefix"**:
        - *prefix* to add to `topo_name` (e.g. `"highways"`) for table of roads network, see above.

    - **"schema_prefix"**:
        - *prefix* to add to `topo_name` (e.g. `"topo"`) for schema with topology data when using `postgis_topology`, see above.

    - **"build"**:
        - **"temp_topo_name**:
            - `""` (*empty*) if not building temporary topologies.
            - `"epoch_ms"` for adding a string with the count of milliseconds since "Epoch" as the `topo_name`.

        - **"srid"**:
            - *number* identifying which projection to use.
                - `900913` for geometric metrical projection, unit meters.
                - `4326` for geographic spherical projection, unit degrees.

        - **"tolerance"**:
            - *snapping* of nodes in unit of projection when building topology, e.g. 1.0 for srid 900913, or 0.001 for srid 4326.

    - **"edge"**:

        - **"table"**:
            - *name* of the table with topology edges (e.g. `"edge_data"`), with column for id, source, target and geometry.

        - **"id_col"**:
```

````
            - *name* of the column in edge table with id of edges (e.g. `"edge_id"`).

        - **"source_col"**:
            - *name* of the column in edge table with vertex id of **source** of edge (e.g. `"start_node"`).

        - **"target_col"**:
            - *name* of the column in edge table with vertex id of **target** of edge (e.g. `"end_node"`).

        - **"geom_col"**:
            - *name* of the column in edge table with geometry of edge (e.g. `"geom"`).

    - **"vertex"**:
        - **"table"**:
            - *name* of the table with topology vertices (e.g. `"node"`), with column for id and geometry.

        - **"id_col"**:
            - *name* of the column in vertex table with id of vertices (e.g. `"node_id"`).

        - **"geom_col"**:
            - *name* of the column in vertex table with geometry of vertex (e.g. `"geom"`).

- **"pgrouting"**:
    - TODO.

- **"jsontest"**:
    - **"test_file"**:
        - `""` (*empty*) if not using *json-test provider*.
        - *filename* to a json test-file (e.g. `"test.json"`) looking like:

    ```json
    {
        "vertices": [
            [1,2,0],
            [2,2,1]
        ],
        "edges": [
            [1,1,2,0]
        ]
    }
    ```

        where each row in `vertices` are `[id,x,y]` and each row in `edges` are `[id, source vertex id, target vertex id, direction]`. Values for `direction` is `0 = BOTH`, `1 = FROM_TO`, `2 = TO_FROM`.


Vehicle
-------
````

```
Configuration about the vehicle to route through the topology. Information might be needed to take restrictions in account.

- **"category"**:
    - *name* of OSM category of the vehicle. [OSM Access](http://wiki.openstreetmap.org/wiki/Key:access). (E.g. "motorcar"). Definition of the category must state dimensions as below.
- **_"category_name"_**:
    - *height* of vehicle in meters.
    - *length* of vehicle in meters.
    - *width* of vehicle in meters.
    - *weight* of vehicle in tons.
    - *maxspeed* of vehicle in km/h.
    - *acceleration* is the time it takes from 0 to 100 km/h.
    - *deceleration* is the time it takes from 100 to 0 km/h.


Access
------

- **"allow"**:
    List of which values for tag `access` that permits access. Those values for `access` that are not listed here are considered to restrict access.


Restrictions
------------

- **"barriers"**:
    - List of which values for `barriers` that restricts access. Those values not listed are assumed to allow access.


Cost
----

Configuration relating to costs when routing through the graph.

- **"default_speed"**:
    - each road category has default speeds when none is specified. [OSM default speeds](http://wiki.openstreetmap.org/wiki/OSM_tags_for_routing/Maxspeed). Most roads have two speeds, `high` and `low`, which differentiate the speeds inside and outside of a town. `living_street` is always inside so only the low is important. `motorway` is really the `high` number, and the `low` number is the speed on the links (ramps). It is not trivial to find out if a road is inside or outside of that area, so for this application which is meant to be used for routing in urban areas (?), the `low` number is assumed for all cost calculations.
- **"surface"**:
    - each surface type is associated with a max speed in km/h over which one should not drive.
- **"barriers"**:
    - this is a list of which barriers causes a slow-down, and the number of seconds it is  probable it takes to pass.
- **"highway"**:
    - a list of values for the `highway` tag that can mean a time cost in seconds, such as zebra crossings, bus stops, stop or give way sign, and more.
```

```
- **"railway"**:
    - a list of values for the `railway` tag that can mean a time cost in seconds,
      ↪ such as a crossing between railway and highway (`level_crossing`).
- **"public_transport"**:
    - a list of values for the `public_transport` tag that can mean a time cost in
      ↪ seconds, such as a `stop_position` if one thinks it is suitable to slow
      ↪ down when passing a bus stop.
- **"traffic_calming"**:
    - a list of traffic calming objects and their time costs in seconds.
```

D.3.2 Configuration.h

```cpp
/**  A container for configurations.
 *
 * #include "Configuration.h"
 *
 * @author  Jonas Bergman
 */

#ifndef CONFIG_CONFIGURATION_H_
#define CONFIG_CONFIGURATION_H_

// SYSTEM INCLUDES
//

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//
#include "DatabaseConfig.h"
#include "VehicleConfig.h"
#include "CostConfig.h"
#include "../osm/OsmAccess.h"
#include "../osm/OsmBarrier.h"

// FORWARD REFERENCES
//

/**
 * This class holds configurations for different parts of the utility.
 * The ConfigurationReader is friend so it can populate the different
 * configurations.
 */
class Configuration
{
    friend class ConfigurationReader;
public:
// LIFECYCLE

    /** Default constructor.
     */
    Configuration() = default;


    /** Copy constructor.
```

```
     *
     * @param from The value to copy to this object.
     */
    Configuration(const Configuration& from) = delete;


    /** Destructor.
     */
    ~Configuration(void) = default;


// OPERATORS
// OPERATIONS
// ACCESS
    /** Get the database related parts of the configuration.
     * @return  Reference to a DatabaseConfig.
     */
    const DatabaseConfig& getDatabaseConfig() const;

    /** Get the topology related parts of the configuration.
     * @return  Reference to a TopologyConfig.
     */
    const TopologyConfig& getTopologyConfig() const;

    /** Get the vehicle related parts of the configuration.
     * @return  Reference to a VehicleConfig.
     */
    const VehicleConfig&  getVehicleConfig() const;

    /** Get the rules for which values of the `access`-tag allows access
     * and hence which values restricts access to an Edge.
     * @return  Reference to an AccessRule
     */
    const OsmAccess::AccessRule&
                          getAccessRule() const;

    /** Get the rules for which values of the `barrier`-tag restricts access
     * @return  Reference to an RestrictionsRule
     */
    const OsmBarrier::RestrictionsRule&
                          getBarrierRestrictionsRule() const;

    /** Get the rules for which values of the `barrier`-tag costs to pass
     * @return  Reference to an CostsRule
     */
    const OsmBarrier::CostsRule&
                          getBarrierCostsRule() const;

    /** Get the cost related parts of the configuration.
     * @return  Reference to a CostConfig.
     */
    const CostConfig&     getCostConfig() const;

// INQUIRY

protected:
private:
```

```
// ATTRIBUTES
    DatabaseConfig          mDbConfig;
    TopologyConfig          mTopoConfig;
    VehicleConfig           mVehicleConfig;
    CostConfig              mCostConfig;
    OsmAccess::AccessRule   mAccessRule;
    OsmBarrier::CostsRule   mBarrierCostsRule;
    OsmBarrier::RestrictionsRule mBarrierRestrictionsRule;
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* CONFIG_CONFIGURATION_H_ */
```

D.3.3 Configuration.cc

```
/*
 * Configuration.cc
 * @author  Jonas Bergman
 */

#include "Configuration.h"  // class implemented

/////////////////////////////// PUBLIC ///////////////////////////////////////

//============================= LIFECYCLE ====================================
//============================= OPERATORS ====================================
//============================= OPERATIONS ===================================
//============================= ACESS      ===================================
const DatabaseConfig&
Configuration::getDatabaseConfig() const
{
    return mDbConfig;
}

const TopologyConfig&
Configuration::getTopologyConfig() const
{
    return mTopoConfig;
}

const VehicleConfig&
Configuration::getVehicleConfig() const
{
    return mVehicleConfig;
}

const OsmAccess::AccessRule&
Configuration::getAccessRule() const
{
    return mAccessRule;
}

```

```
const OsmBarrier::RestrictionsRule&
Configuration::getBarrierRestrictionsRule() const
{
    return mBarrierRestrictionsRule;
}

const OsmBarrier::CostsRule&
Configuration::getBarrierCostsRule() const
{
    return mBarrierCostsRule;
}

const CostConfig&
Configuration::getCostConfig() const
{
    return mCostConfig;
}

//============================= INQUIRY    ===================================
/////////////////////////////// PROTECTED  ///////////////////////////////////

/////////////////////////////// PRIVATE    ///////////////////////////////////
```

D.3.4 ConfigurationException.h

```
/**  Exception thrown by the Configuration package.
 *
 * #include "ConfigurationException.h"
 *
 * @author  Jonas Bergman
 */

#ifndef CONFIG_CONFIGURATIONEXCEPTION_H_
#define CONFIG_CONFIGURATIONEXCEPTION_H_

// SYSTEM INCLUDES
//
#include <exception>
#include <string>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//

// FORWARD REFERENCES
//

/**
 * Exception to throw from the 'config' package.
 * More information of the type of exception is given in the 'what()' message.
 */
class ConfigurationException : public std::exception
{
public:
// LIFECYCLE
```

```
    /** Default constructor.
     */
    ConfigurationException() = delete;

    /** Constructor taking a message to display.
     *
     * @param       message                The message to prepend when 'what()' is called.
     */
    ConfigurationException(const std::string& rMessage) noexcept
        : std::exception(), mMessage(rMessage)
    {}

// OPERATORS
// OPERATIONS
// ACCESS
// INQUIRY
    const char* what() const noexcept
    { return (mMessage + " " + std::exception::what()).c_str(); }

protected:
private:
// ATTRIBUTES
    std::string                 mMessage;
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* CONFIG_CONFIGURATIONEXCEPTION_H_ */
```

D.3.5 ConfigurationReader.h

```
/**  Read configurations from a json file.
 *
 * #include "ConfigurationReader.h"
 *
 * @author  Jonas Bergman
 */
#ifndef CONFIG_CONFIGURATIONREADER_H_
#define CONFIG_CONFIGURATIONREADER_H_

// SYSTEM INCLUDES
//
#include <string>

// PROJECT INCLUDES
//
#include <boost/property_tree/ptree.hpp>
#include <boost/property_tree/json_parser.hpp>

// LOCAL INCLUDES
//
#include "Configuration.h"
#include "ConfigurationException.h"
```

```
#include "DatabaseConfig.h"
#include "TopologyConfig.h"
#include "VehicleConfig.h"
#include "../osm/OsmVehicle.h"

// FORWARD REFERENCES
//

/**
 * A class to handle the reading of data from a json configuration file.
 */
class ConfigurationReader
{
public:
// LIFECYCLE
    /** Default constructor.
     */
    ConfigurationReader() = delete;

    /** Constructor.
     * Always initialize a Configuration reader with the configuration file.
     *
     * @param    rFilename    The filename for the configuration json file
     * @throw    ConfigurationException        If invalid file
     */
    ConfigurationReader(const std::string& rFilename);


// OPERATORS
// OPERATIONS

    /** Get the configurations from the file.
     * @param   Reference to a Configuration to populate.
     * @throws  ConfigurationException
     */
    void    fillConfiguration(Configuration& rConfig) const;

// ACCESS
// INQUIRY

protected:

private:
// ATTRIBUTES
    std::string                     mFilename;
    boost::property_tree::ptree     mPropertyTree;

// HELPERS
    /** Read the database part of the configuration and populate config struct.
     * @param    The Database configuration
     * @throw    ConfigurationException    If missing configuration.
     */
    void    fillDatabaseConfiguration(DatabaseConfig& rDatabaseConfig) const;

    /** Read the topology part of the configuration and populate config struct.
     * @param    The Topology configuration
     * @throw    ConfigurationException    If missing configuration.
```

```
     */
    void    fillTopologyConfiguration(TopologyConfig& rTopologyConfig) const;

    /** Read the vehicle part of the configuration and populate config struct.
     * @param   The Vehicle configuration
     * @throw   ConfigurationException  If missing configuration.
     */
    void    fillVehicleConfiguration(VehicleConfig& rVehicleConfig) const;

    /** Read the Access part of the configuration and build the rule for
     * which tags allows access (and hence which tags restricts access).
     * @param   rAccessRule     The rule to fill out.
     * @throw   ConfigurationException  If missing configuration.
     */
    void    fillAccessRule(OsmAccess::AccessRule& rAccessRule) const;

    /** Read the Barrier part of the configuration and build the rule for
     * which barriers restricts access.
     * @param   rRestrictRule     The rule to fill out.
     * @throw   ConfigurationException  If missing configuration.
     */
    void    fillBarrierRestrictRule(OsmBarrier::RestrictionsRule& rRestrictRule) const;

    /** Read the Barrier part of the configuration and build the rule for
     * which barriers imposes a cost.
     * @param   rCostRule     The rule to fill out.
     * @throw   ConfigurationException  If missing configuration.
     */
    void    fillBarrierCostsRule(OsmBarrier::CostsRule& rCostsRule) const;

    /** Read the Cost part of the configuration and populate config struct.
     * @param   The Cost configuration
     * @throw   ConfigurationException  If missing configuration.
     */
    void    fillCostConfiguration(CostConfig& rCostConfig) const;

    /** Helper to `fillCostConfig()`. Fill in the Default Speed part.
     * @param   The Cost configuration.
     */
    void    fillDefaultSpeedCost(CostConfig& rCostConfig) const;

    /** Helper to `fillCostConfig()`. Fill in the Surface Max Speed part.
     * @param   The Cost configuration.
     */
    void    fillSurfaceMaxSpeedCost(CostConfig& rCostConfig) const;

    /** Helper to `fillCostConfig()`. Fill in the cost for other edge costs.
     * @param   The Cost configuration.
     */
    void    fillOtherEdgeCosts(CostConfig& rCostConfig) const;
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//
```

```

#endif /* CONFIG_CONFIGURATIONREADER_H_ */
```

D.3.6 ConfigurationReader.cc

```
/*
 * ConfigurationReader.cc
 *
 * @author  Jonas Bergman
 */

#include "ConfigurationReader.h"  // class implemented

/////////////////////////////// PUBLIC ///////////////////////////////////////

//============================= LIFECYCLE ====================================

ConfigurationReader::ConfigurationReader(const std::string& rFilename)
    : mFilename(rFilename)
{
    try
    {
        boost::property_tree::read_json(mFilename, mPropertyTree);
    }
    catch (boost::property_tree::json_parser_error& e)
    {
        throw ConfigurationException("Could not read file " + mFilename);
    }
}

//============================= OPERATORS ====================================

//============================= OPERATIONS ===================================
void ConfigurationReader::fillConfiguration(Configuration& rConfig) const
{
    fillDatabaseConfiguration(rConfig.mDbConfig);
    fillTopologyConfiguration(rConfig.mTopoConfig);
    fillVehicleConfiguration(rConfig.mVehicleConfig);
    fillAccessRule(rConfig.mAccessRule);
    fillBarrierRestrictRule(rConfig.mBarrierRestrictionsRule);
    fillBarrierCostsRule(rConfig.mBarrierCostsRule);
    fillCostConfiguration(rConfig.mCostConfig);
}

//============================= ACESS      ===================================
//============================= INQUIRY    ===================================
/////////////////////////////// PROTECTED  ///////////////////////////////////

/////////////////////////////// PRIVATE    ///////////////////////////////////
void ConfigurationReader::fillDatabaseConfiguration(
    DatabaseConfig& rDbConfig) const
{
    std::string prefix("database.");

    try
    {
        rDbConfig.hostname = mPropertyTree.get<std::string>(prefix + "host");
```

```
        rDbConfig.port = mPropertyTree.get<int>(prefix + "port");
        rDbConfig.username = mPropertyTree.get<std::string>(
            prefix + "username");
        rDbConfig.password = mPropertyTree.get<std::string>(
            prefix + "password");
        rDbConfig.database = mPropertyTree.get<std::string>(
            prefix + "database");
    }
    catch (boost::property_tree::ptree_error& e)
    {
        throw ConfigurationException(
            std::string("Could not read config ") + e.what());
    }
}

void ConfigurationReader::fillTopologyConfiguration(
    TopologyConfig& rTopoConfig) const
{
    std::string prefix("topology.");

    try
    {
        rTopoConfig.providerName = mPropertyTree.get<std::string>(
            prefix + "provider");

        if(rTopoConfig.providerName == TopologyConfig::PROVIDER_JSONTEST)
        {
            rTopoConfig.testFile = mPropertyTree.get<std::string>(
                prefix + "jsontest.test_file");
        }
        else if(rTopoConfig.providerName == TopologyConfig::PROVIDER_POSTGIS
            || rTopoConfig.providerName == TopologyConfig::PROVIDER_PGROUTING)
        {
            prefix += rTopoConfig.providerName + ".";

            rTopoConfig.topoName = mPropertyTree.get<std::string>(
                prefix + "topo_name");

            rTopoConfig.roadsPrefix = mPropertyTree.get<std::string>(
                prefix + "roads_prefix");
            rTopoConfig.topologySchemaPrefix = mPropertyTree.get<std::string>(
                prefix + "schema_prefix");

            rTopoConfig.tempTopoName = mPropertyTree.get<std::string>(
                prefix + "build.temp_topo_name");
            rTopoConfig.srid = mPropertyTree.get<int>(prefix + "build.srid");
            rTopoConfig.tolerance = mPropertyTree.get<double>(
                prefix + "build.tolerance");

            rTopoConfig.edgeTableName = mPropertyTree.get<std::string>(
                prefix + "edge.table");
            rTopoConfig.edgeIdColumnName = mPropertyTree.get<std::string>(
                prefix + "edge.id_col");
            rTopoConfig.sourceColumnName = mPropertyTree.get<std::string>(
                prefix + "edge.source_col");
            rTopoConfig.targetColumnName = mPropertyTree.get<std::string>(
                prefix + "edge.target_col");
```

```
            rTopoConfig.edgeGeomColumnName = mPropertyTree.get<std::string>(
                prefix + "edge.geom_col");

            rTopoConfig.vertexTableName = mPropertyTree.get<std::string>(
                prefix + "vertex.table");
            rTopoConfig.vertexIdColumnName = mPropertyTree.get<std::string>(
                prefix + "vertex.id_col");
            rTopoConfig.vertexGeomColumnName = mPropertyTree.get<std::string>(
                prefix + "vertex.geom_col");
        }
    }
    catch (boost::property_tree::ptree_error& e)
    {
        throw ConfigurationException(
            std::string("Could not read config ") + e.what());
    }
}

void ConfigurationReader::fillVehicleConfiguration(
    VehicleConfig& rVehicleConfig) const
{
    std::string prefix("vehicle.");

    try
    {
        std::string categoryString = mPropertyTree.get<std::string>(
            prefix + "category");
        rVehicleConfig.category = OsmVehicle::parseString(categoryString);
        prefix += categoryString + ".";
        rVehicleConfig.height = mPropertyTree.get<double>(prefix + "height");
        rVehicleConfig.length = mPropertyTree.get<double>(prefix + "length");
        rVehicleConfig.weight = mPropertyTree.get<double>(prefix + "weight");
        rVehicleConfig.width = mPropertyTree.get<double>(prefix + "width");
        rVehicleConfig.maxspeed = mPropertyTree.get<unsigned>(
            prefix + "maxspeed");
        rVehicleConfig.acceleration = mPropertyTree.get<unsigned>(
            prefix + "acceleration");
        rVehicleConfig.deceleration = mPropertyTree.get<unsigned>(
            prefix + "deceleration");
    }
    catch (ConfigurationException& e)
    {
        throw e;
    }
    catch (boost::property_tree::ptree_error& e)
    {
        throw ConfigurationException(
            std::string("Could not read config ") + e.what());
    }
}

void ConfigurationReader::fillAccessRule(
    OsmAccess::AccessRule& rAccessRule) const
{
    std::string prefix("access.allow");

    try
```

```
    {
        std::vector<OsmAccess::AccessType> allow_tags;
        for (auto& item : mPropertyTree.get_child(prefix))
        {
            std::string tag_string = item.second.get_value<std::string>();
            allow_tags.push_back(OsmAccess::parseString(tag_string));
        }
        rAccessRule.allowAccessToTypes = allow_tags;
    }
    catch (ConfigurationException& e)
    {
        throw e;
    }
    catch (OsmException& ose)
    {
        throw ConfigurationException(
            std::string("Could not read config")
                + ", error parsing access tag: " + ose.what());
    }
    catch (boost::property_tree::ptree_error& e)
    {
        throw ConfigurationException(
            std::string("Could not read config ") + e.what());
    }
}

void ConfigurationReader::fillBarrierRestrictRule(
    OsmBarrier::RestrictionsRule& rRestrictRule) const
{
    std::string prefix("restrict.barriers");

    try
    {
        std::vector<OsmBarrier::BarrierType> restrict_barriers;
        for (auto& item : mPropertyTree.get_child(prefix))
        {
            std::string restrict_string =
                item.second.get_value<std::string>();
            restrict_barriers.push_back(
                OsmBarrier::parseString(restrict_string));
        }
        rRestrictRule.restrictionTypes = restrict_barriers;
    }
    catch (ConfigurationException& e)
    {
        throw e;
    }
    catch (OsmException& ose)
    {
        throw ConfigurationException(
            std::string("Could not read config")
                + ", error parsing barrier restrictions: " + ose.what());
    }
    catch (boost::property_tree::ptree_error& e)
    {
        throw ConfigurationException(
            std::string("Could not read config ") + e.what());
```

```
    }
}

void ConfigurationReader::fillBarrierCostsRule(
    OsmBarrier::CostsRule& rCostsRule) const
{
    std::string prefix("cost.barriers");

    try
    {
        for (auto& row : mPropertyTree.get_child(prefix))
        {
            int i = 0;
            std::string type_string;
            unsigned cost;
            for (auto& item : row.second)
            {
                if(i == 0)
                {
                    type_string = item.second.get_value<std::string>();
                }
                else
                {
                    cost = item.second.get_value<unsigned>();
                }
                ++i;
            }
            OsmBarrier::BarrierType barrier_type = OsmBarrier::parseString(
                type_string);
            rCostsRule.addCost(barrier_type, cost);
        }
    }
    catch (ConfigurationException& e)
    {
        throw e;
    }
    catch (OsmException& ose)
    {
        throw ConfigurationException(
            std::string("Could not read config")
                + ", error parsing barrier costs: " + ose.what());
    }
    catch (boost::property_tree::ptree_error& e)
    {
        throw ConfigurationException(
            std::string("Could not read config ") + e.what());
    }
}

void ConfigurationReader::fillCostConfiguration(CostConfig& rCostConfig) const
{
    try
    {
        fillDefaultSpeedCost(rCostConfig);
        fillSurfaceMaxSpeedCost(rCostConfig);
        fillOtherEdgeCosts(rCostConfig);
    }
```

```
    catch (ConfigurationException& e)
    {
        throw e;
    }
    catch (boost::property_tree::ptree_error& e)
    {
        throw ConfigurationException(
            std::string("Could not read config ") + e.what());
    }
}

void ConfigurationReader::fillDefaultSpeedCost(CostConfig& rCostConfig) const
{
    std::string prefix("cost.default_speed.");

    CostConfig::DefaultSpeed::HighLowSpeed hilo;
    std::string type_string;
    OsmHighway::HighwayType type;

    for (size_t i = 0; i < OsmHighway::NR_HIGHWAY_TYPES; ++i)
    {
        type_string = OsmHighway::typeStrings().at(i);
        hilo.high = mPropertyTree.get<int>(prefix + type_string + ".high");
        hilo.low = mPropertyTree.get<int>(prefix + type_string + ".low");
        type = static_cast<OsmHighway::HighwayType>(i);
        rCostConfig.defaultSpeed.addDefaultSpeed(type, hilo);
    }
}

void ConfigurationReader::fillSurfaceMaxSpeedCost(
    CostConfig& rCostConfig) const
{
    std::string prefix("cost.surface.");

    Speed speed;
    std::string type_string;
    OsmHighway::SurfaceType type;

    for (size_t i = 0; i < OsmHighway::NR_SURFACE_TYPES; ++i)
    {
        type_string = OsmHighway::surfaceTypeStrings().at(i);
        speed = mPropertyTree.get<int>(prefix + type_string);
        type = static_cast<OsmHighway::SurfaceType>(i);
        rCostConfig.surfaceMaxSpeed.addSurfaceMaxSpeed(type, speed);
    }
}

void ConfigurationReader::fillOtherEdgeCosts(CostConfig& rCostConfig) const
{
    std::string section("cost.");
    std::vector<std::string> subsections { "highway", "railway",
        "public_transport", "traffic_calming" };

    try
    {
        for (const auto& sub : subsections)
        {
```

```
            std::string prefix(section + sub + ".");

            for (auto& row : mPropertyTree.get_child(prefix))
            {
                int i = 0;
                std::string key;
                Cost cost;
                for (auto& item : row.second)
                {
                    if(i == 0)
                    {
                        key = item.second.get_value<std::string>();
                    }
                    else
                    {
                        cost = item.second.get_value<Cost>();
                    }
                    ++i;
                }
                rCostConfig.otherEdgeCosts.addOtherCost(sub + "=" + key,
                    cost);
            }
        }
    }
    catch (ConfigurationException& e)
    {
        throw e;
    }
    catch (OsmException& ose)
    {
        throw ConfigurationException(
            std::string("Could not read config")
                + ", error parsing other costs: " + ose.what());
    }
    catch (boost::property_tree::ptree_error& e)
    {
        throw ConfigurationException(
            std::string("Could not read config ") + e.what());
    }
}
```

D.3.7 CostConfig.h

```
/**  Data structure for configuration of costs.
 *
 * #include "CostConfig.h"
 *
 * @author  Jonas Bergman
 */

#ifndef CONFIG_COSTCONFIG_H_
#define CONFIG_COSTCONFIG_H_

// SYSTEM INCLUDES
//
#include <map>
#include <string>
```

```

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//
#include "../osm/OsmHighway.h"
#include "../graph/Cost.h"
#include "../graph/Speed.h"

// FORWARD REFERENCES
//

/**
 * Information about names in the database for cost data.
 */
struct CostConfig
{
// TYPES

    /** Keep track of default speeds for different categories of roads.
     * The values are a high and a low value, depending of if we are inside or
     * outside of an urban area.
     */
    struct DefaultSpeed
    {
        enum HIGH_LOW
        {
            HIGH,
            LOW
        };

        struct HighLowSpeed
        {
            Speed high {0};
            Speed low {0};
        };

        std::map<OsmHighway::HighwayType, HighLowSpeed> defaultSpeed;

        /** Add a speed far a specific road category (highway type).
         * @param   type    The highway type
         * @param   speed   The high and low speed limits.
         */
        void            addDefaultSpeed(
                            OsmHighway::HighwayType type,
                            HighLowSpeed speed)
        {
            defaultSpeed.erase(type);
            defaultSpeed.insert({type, speed});
        }

        /**
         * @param   type    The highway type
         * @return  The high/low speed for this type of highway.
         */
        HighLowSpeed    getDefaultSpeed(OsmHighway::HighwayType type) const
```

```
        {
            const auto& it = defaultSpeed.find(type);
            if(it != defaultSpeed.end())
            {
                return it->second;
            }
            return HighLowSpeed();
        }

        /** Get a high or low speed limit for a highway type.
         * @param   type        The Type of highway.
         * @param   highOrLow   Either HIGH or LOW speed
         * @return  Either the high or low speed for a highway type.
         */
        Speed           getDefaultSpeed(
                            OsmHighway::HighwayType type,
                            HIGH_LOW                highOrLow) const
        {
            HighLowSpeed hl = getDefaultSpeed(type);
            if(highOrLow == HIGH)
            {
                return hl.high;
            }
            else
            {
                return hl.low;
            }
        }
    };

    /** Keep track of max speed that are suitable for different kind of
     * surfaces.
     */
    struct SurfaceMaxSpeed
    {
        std::map<OsmHighway::SurfaceType, Speed> surfaceSpeed;

        /** Add a surface type and the max speed.
         * @param   type    The type of surface.
         * @param   speed   The max suitable speed for the surface type.
         */
        void    addSurfaceMaxSpeed(OsmHighway::SurfaceType type, Speed speed)
        {
            surfaceSpeed.erase(type);
            surfaceSpeed.insert({type, speed});
        }

        /**
         * @return  The suitable max speed for a surface type.
         */
        Speed   getSurfaceMaxSpeed(OsmHighway::SurfaceType type) const
        {
            const auto& it = surfaceSpeed.find(type);
            if(it != surfaceSpeed.end())
            {
                return it->second;
            }
```

```
                return 0;
            }
        };

        /** Other edge costs are kept track of simply by strings as keys and
         * Costs as values. The costs are "penalties" added to the travel time.
         * The string that make up the keys are simply constructed as "tag=value",
         * e.g. "highway=give_way".
         */
        struct OtherEdgeCosts
        {
            std::map<std::string, Cost> otherCostValues;

            /** Add a 'penalty' for another kind of EdgeCost.
             * @param   key     String of "tag=value" that makes up the cost.
             * @param   cost    The cost for this kind of hindrance.
             */
            void    addOtherCost(std::string key, Cost cost)
            {
                otherCostValues.erase(key);
                otherCostValues.insert({key, cost});
            }

            /** Get other costs associated with the key.
             * @param   key
             * @return  The cost for this key.
             */
            Cost    getOtherCost(std::string key) const
            {
                const auto& it = otherCostValues.find(key);
                if(it !=otherCostValues.end())
                {
                    return it->second;
                }
                return 0;
            }
        };

    // ATTRIBUTES
        DefaultSpeed    defaultSpeed;
        SurfaceMaxSpeed surfaceMaxSpeed;
        OtherEdgeCosts  otherEdgeCosts;

    // ACCESS
    // CONSTANTS

    private:
    };

    // INLINE METHODS
    //

    // EXTERNAL REFERENCES
    //

    #endif /* CONFIG_COSTCONFIG_H_ */
```

D.3.8 DatabaseConfig.h

```
/**  Data structure for configuration of database connection.
 *
 * #include "DatabaseConfig.h"
 *
 * @author  Jonas Bergman
 */
#ifndef CONFIG_DATABASECONFIG_H_
#define CONFIG_DATABASECONFIG_H_

// SYSTEM INCLUDES
//
#include <string>
#include <sstream>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//
#include "TopologyConfig.h"

// FORWARD REFERENCES
//

/** A simple data structure for holding the configuration
 * for database connections.
 */
struct DatabaseConfig
{
// ATTRIBUTES
    std::string        hostname;
    int                port;
    std::string        username;
    std::string        password;
    std::string        database;


// OPERATIONS
    /** Construct a connection string from the attributes.
     * @return    A valid connection string for 'pqxx::conn()'
     */
    std::string     getConnectionString() const
    {
        std::ostringstream oss;
        oss << "host=" << hostname
            << " port=" << port
            << " user=" << username
            << " password=" << password
            << " dbname=" << database;
        return  oss.str();
    }

private:
};

// INLINE METHODS
```

```
//

// EXTERNAL REFERENCES
//

#endif /* CONFIG_DATABASECONFIG_H_ */
```

D.3.9 TopologyConfig.h

```
/**  Data structure for configuration of topology data in database.
 *
 * #include "TopologyConfig.h"
 *
 * @author   Jonas Bergman
 */

#ifndef CONFIG_TOPOLOGYCONFIG_H_
#define CONFIG_TOPOLOGYCONFIG_H_

// SYSTEM INCLUDES
//
#include <string>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//

// FORWARD REFERENCES
//

/**
 * Information about names in the database for topology data.
 */
struct TopologyConfig
{
// ATTRIBUTES
    std::string     providerName;

    std::string     tempTopoName;
    std::string     topoName;

    std::string     roadsPrefix;
    std::string     topologySchemaPrefix;

    int             srid;
    double          tolerance;

    std::string     edgeTableName;
    std::string     edgeIdColumnName;
    std::string     sourceColumnName;
    std::string     targetColumnName;
    std::string     edgeGeomColumnName;

    std::string     vertexTableName;
    std::string     vertexIdColumnName;
```

```
    std::string     vertexGeomColumnName;

    std::string     testFile;

// CONSTANTS
    static constexpr const char* PROVIDER_POSTGIS = "postgis";
    static constexpr const char* PROVIDER_PGROUTING = "pgrouting";
    static constexpr const char* PROVIDER_JSONTEST = "jsontest";
    static constexpr const char* TEMP_TOPO_NAMEBASE = "epoch_ms";


};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* CONFIG_TOPOLOGYCONFIG_H_ */
```

D.3.10 VehicleConfig.h

```
/**  Data structure for configuration of vehicle we are routing.
 *
 * #include "VehicleConfig.h"
 *
 * @author  Jonas Bergman
 */
#ifndef CONFIG_VEHICLECONFIG_H_
#define CONFIG_VEHICLECONFIG_H_

// SYSTEM INCLUDES
//
#include <string>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//
#include "../osm/OsmVehicle.h"

// FORWARD REFERENCES
//

/** A simple data structure for holding the configuration
 *  of the vehicle we are routing.
 */
struct VehicleConfig
{
// ATTRIBUTES
    OsmVehicle::VehicleType     category;
    double                      height;
    double                      length;
    double                      weight;
    double                      width;
```

```
    unsigned                    maxspeed;
    unsigned                    acceleration; // seconds 0 - 100 km/h
    unsigned                    deceleration; // seconds 100 - 0 km/h
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* CONFIG_VEHICLECONFIG_H_ */
```

D.3.11 ConfigurationReader_test.cc

```
/*
 * ConfigurationReader_test.cc
 * @author  Jonas Bergman
 */

#include "../../catchtest/catch.hpp"
#include "../Configuration.h"
#include "../ConfigurationReader.h"
#include "../ConfigurationException.h"

SCENARIO ("Use ConfigurationReader to read configuration from json file",
    "[json],[config]")
{
    //--------------------------------------------------------------------------
    GIVEN ("a filename to a valid configuration file")
    {
        std::string filename("catchtest/testsettings/testsettings.json");

        WHEN ("asking for database configuration")
        {
            ConfigurationReader config_reader(filename);
            Configuration config;
            config_reader.fillConfiguration(config);
            const DatabaseConfig& r_db_config = config.getDatabaseConfig();

            THEN ("we get a  database configuration filled out")
            {
                REQUIRE (r_db_config.hostname == "127.0.0.1");
                REQUIRE (r_db_config.port == 5432);
                REQUIRE (r_db_config.username == "tester");
                REQUIRE (r_db_config.password == "tester");
                REQUIRE (r_db_config.database == "mikh_0530");
            }
        }
    }

    //--------------------------------------------------------------------------
    GIVEN ("a filename to a configuration file with missing information")
    {
        std::string filename("catchtest/testsettings/testsettings-missing-name.json");

        WHEN ("asking for database configuration")
```

```
        {
            ConfigurationReader config_reader(filename);

            THEN ("we get an exception")
            {
                Configuration config;
                REQUIRE_THROWS_AS (config_reader.fillConfiguration(config),
                    ConfigurationException&);
            }
        }
    }

    //--------------------------------------------------------------------------
    GIVEN ("a filename to a non-existing file")
    {
        std::string filename("config/catchtest/foo.json");

        WHEN ("asking for database configuration")
        {
            THEN ("we get an exception")
            {
                REQUIRE_THROWS_AS (ConfigurationReader config_reader(filename),
                    ConfigurationException&);
            }
        }
    }

    //--------------------------------------------------------------------------
    GIVEN ("a filename to a valid configuration file")
    {
        std::string filename("catchtest/testsettings/testsettings.json");
        ConfigurationReader config_reader(filename);
        Configuration config;
        config_reader.fillConfiguration(config);

        //..................................................................
        WHEN ("asking for topology configuration")
        {
            const TopologyConfig& r_topo_config = config.getTopologyConfig();

            THEN ("we get a topology configuration filled out")
            {
                REQUIRE (r_topo_config.providerName == "postgis");
                REQUIRE (r_topo_config.topoName == "lgu");
                REQUIRE (r_topo_config.roadsPrefix == "highways");
                REQUIRE (r_topo_config.topologySchemaPrefix == "topo");
                REQUIRE (r_topo_config.tempTopoName == "");
                REQUIRE (r_topo_config.srid == 900913);
                REQUIRE (r_topo_config.tolerance == Approx(1.0));

                REQUIRE (r_topo_config.edgeTableName == "edge_data");
                REQUIRE (r_topo_config.edgeIdColumnName == "edge_id");
                REQUIRE (r_topo_config.sourceColumnName == "start_node");
                REQUIRE (r_topo_config.targetColumnName == "end_node");
                REQUIRE (r_topo_config.edgeGeomColumnName == "geom");
                REQUIRE (r_topo_config.vertexTableName == "node");
                REQUIRE (r_topo_config.vertexIdColumnName == "node_id");
```

```
                REQUIRE (r_topo_config.vertexGeomColumnName == "geom");
            }
        }

        //..............................................................
        WHEN ("asking for vehicle configuration")
        {
            const VehicleConfig& r_vehicle_config = config.getVehicleConfig();

            THEN ("we get a vehicle configuration filled out")
            {
                REQUIRE (r_vehicle_config.category == OsmVehicle::MOTORCAR);
                REQUIRE (r_vehicle_config.height == Approx(1.6));
                REQUIRE (r_vehicle_config.length == Approx(4.5));
                REQUIRE (r_vehicle_config.weight == Approx(2.0));
                REQUIRE (r_vehicle_config.width == Approx(1.9));
                REQUIRE (r_vehicle_config.maxspeed == 200);
                REQUIRE (r_vehicle_config.acceleration == 10);
                REQUIRE (r_vehicle_config.deceleration == 7);
            }
        }

        //..............................................................
        WHEN ("asking for cost configuration")
        {
            const CostConfig& r_cost_config = config.getCostConfig();

            THEN ("we get a cost configuration filled out")
            {
                REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
                    OsmHighway::MOTORWAY, CostConfig::DefaultSpeed::HIGH) == 110);
                REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
                    OsmHighway::MOTORWAY, CostConfig::DefaultSpeed::LOW) == 90);
                REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
                    OsmHighway::MOTORWAY_LINK, CostConfig::DefaultSpeed::HIGH) == 90);
                REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
                    OsmHighway::MOTORWAY_LINK, CostConfig::DefaultSpeed::LOW) == 90);

                REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
                    OsmHighway::TRUNK, CostConfig::DefaultSpeed::HIGH) == 90);
                REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
                    OsmHighway::TRUNK, CostConfig::DefaultSpeed::LOW) == 60);
                REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
                    OsmHighway::TRUNK_LINK, CostConfig::DefaultSpeed::HIGH) == 90);
                REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
                    OsmHighway::TRUNK_LINK, CostConfig::DefaultSpeed::LOW) == 60);

                REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
                    OsmHighway::PRIMARY, CostConfig::DefaultSpeed::HIGH) == 90);
                REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
                    OsmHighway::PRIMARY, CostConfig::DefaultSpeed::LOW) == 60);
                REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
                    OsmHighway::PRIMARY_LINK, CostConfig::DefaultSpeed::HIGH) == 90);
                REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
                    OsmHighway::PRIMARY_LINK, CostConfig::DefaultSpeed::LOW) == 60);

                REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
```

```
            OsmHighway::SECONDARY, CostConfig::DefaultSpeed::HIGH) == 90);
        REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
            OsmHighway::SECONDARY, CostConfig::DefaultSpeed::LOW) == 60);
        REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
            OsmHighway::SECONDARY_LINK, CostConfig::DefaultSpeed::HIGH) == 90);
        REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
            OsmHighway::SECONDARY_LINK, CostConfig::DefaultSpeed::LOW) == 60);

        REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
            OsmHighway::TERTIARY, CostConfig::DefaultSpeed::HIGH) == 90);
        REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
            OsmHighway::TERTIARY, CostConfig::DefaultSpeed::LOW) == 60);
        REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
            OsmHighway::TERTIARY_LINK, CostConfig::DefaultSpeed::HIGH) == 90);
        REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
            OsmHighway::TERTIARY_LINK, CostConfig::DefaultSpeed::LOW) == 60);

        REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
            OsmHighway::UNCLASSIFIED, CostConfig::DefaultSpeed::HIGH) == 90);
        REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
            OsmHighway::UNCLASSIFIED, CostConfig::DefaultSpeed::LOW) == 60);

        REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
            OsmHighway::RESIDENTIAL, CostConfig::DefaultSpeed::HIGH) == 90);
        REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
            OsmHighway::RESIDENTIAL, CostConfig::DefaultSpeed::LOW) == 60);

        REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
            OsmHighway::SERVICE, CostConfig::DefaultSpeed::HIGH) == 40);
        REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
            OsmHighway::SERVICE, CostConfig::DefaultSpeed::LOW) == 20);

        REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
            OsmHighway::LIVING_STREET, CostConfig::DefaultSpeed::HIGH) == 20);
        REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
            OsmHighway::LIVING_STREET, CostConfig::DefaultSpeed::LOW) == 20);

        REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
            OsmHighway::BUS_GUIDEWAY, CostConfig::DefaultSpeed::HIGH) == 80);
        REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
            OsmHighway::BUS_GUIDEWAY, CostConfig::DefaultSpeed::LOW) == 60);

        REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
            OsmHighway::ROAD, CostConfig::DefaultSpeed::HIGH) == 80);
        REQUIRE (r_cost_config.defaultSpeed.getDefaultSpeed(
            OsmHighway::ROAD, CostConfig::DefaultSpeed::LOW) == 50);

        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::PAVED) == 1000);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::ASPHALT) == 1000);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::COBBLESTONE) == 20);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::COBBLESTONE_FLATTENED) == 40);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::SETT) == 40);
```

```
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::CONCRETE) == 1000);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::CONCRETE_LANES) == 40);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::CONCRETE_PLATES) == 100);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::PAVING_STONES) == 40);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::METAL) == 60);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::WOOD) == 30);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::UNPAVED) == 60);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::COMPACTED) == 70);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::DIRT) == 40);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::EARTH) == 40);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::FINE_GRAVEL) == 50);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::GRASS) == 10);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::GRASS_PAVER) == 20);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::GRAVEL) == 60);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::GROUND) == 20);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::ICE) == 70);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::MUD) == 5);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::PEBBLESTONE) == 50);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::SALT) == 70);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::SAND) == 70);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::SNOW) == 50);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::WOODCHIPS) == 5);
        REQUIRE (r_cost_config.surfaceMaxSpeed.getSurfaceMaxSpeed(
            OsmHighway::METAL_GRID) == 40);

        REQUIRE(r_cost_config.otherEdgeCosts.getOtherCost(
            "highway=bus_stop") == 5);
        REQUIRE(r_cost_config.otherEdgeCosts.getOtherCost(
            "highway=crossing") == 5);
        REQUIRE(r_cost_config.otherEdgeCosts.getOtherCost(
            "highway=give_way") == 20);
        REQUIRE(r_cost_config.otherEdgeCosts.getOtherCost(
            "highway=mini_roundabout") == 20);
        REQUIRE(r_cost_config.otherEdgeCosts.getOtherCost(
            "highway=stop") == 30);
```

```
                REQUIRE(r_cost_config.otherEdgeCosts.getOtherCost(
                    "highway=traffic_signals") == 30);

                REQUIRE(r_cost_config.otherEdgeCosts.getOtherCost(
                    "railway=level_crossing") == 20);

                REQUIRE(r_cost_config.otherEdgeCosts.getOtherCost(
                    "public_transport=stop_position") == 5);

                REQUIRE(r_cost_config.otherEdgeCosts.getOtherCost(
                    "traffic_calming=yes") == 10);
                REQUIRE(r_cost_config.otherEdgeCosts.getOtherCost(
                    "traffic_calming=bump") == 10);
                REQUIRE(r_cost_config.otherEdgeCosts.getOtherCost(
                    "traffic_calming=table") == 10);
                REQUIRE(r_cost_config.otherEdgeCosts.getOtherCost(
                    "traffic_calming=cushion") == 10);
                REQUIRE(r_cost_config.otherEdgeCosts.getOtherCost(
                    "traffic_calming=rumble_strip") == 10);
                REQUIRE(r_cost_config.otherEdgeCosts.getOtherCost(
                    "traffic_calming=chicane") == 10);
                REQUIRE(r_cost_config.otherEdgeCosts.getOtherCost(
                    "traffic_calming=choker") == 10);
                REQUIRE(r_cost_config.otherEdgeCosts.getOtherCost(
                    "traffic_calming=island") == 5);
            }
        }

        WHEN ("asking for access rules")
        {
            const OsmAccess::AccessRule r_access_rule = config.getAccessRule();

            THEN ("we get an AccessRule filled out")
            {
                std::vector<OsmAccess::AccessType> types =
                    r_access_rule.allowAccessToTypes;
                REQUIRE (types.size() == 3);

                auto it = std::find(types.begin(), types.end(),
                    OsmAccess::AccessType::YES);
                INFO ("Allows access:" + OsmAccess::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmAccess::AccessType::PERMISSIVE);
                INFO ("Allows access:" + OsmAccess::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmAccess::AccessType::DESIGNATED);
                INFO ("Allows access:" + OsmAccess::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmAccess::AccessType::NO);
                INFO ("Denies access: no");
                REQUIRE (it == types.end());
```

```
            }
        }

        WHEN ("asking for restrictions rules")
        {
            const OsmBarrier::RestrictionsRule
            r_restrict_rule = config.getBarrierRestrictionsRule();

            THEN ("we get RestrictionsRule filled out")
            {
                std::vector<OsmBarrier::BarrierType> types =
                    r_restrict_rule.restrictionTypes;
                REQUIRE (types.size() == 21);

                auto it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::BLOCK);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::BOLLARD);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::BUS_TRAP);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::CHAIN);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::CYCLE_BARRIER);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::DEBRIS);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::FULLHEIGHT_TURNSTILE);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::HORSE_STILE);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::JERSEY_BARRIER);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
```

```
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::KENT_CARRIAGE_GAP);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::KISSING_GATE);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::LOG);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::MOTORCYCLE_BARRIER);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::ROPE);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::SALLY_PORT);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::SPIKES);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::STILE);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::SUMP_BUSTER);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::SWING_GATE);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::TURNSTILE);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
                REQUIRE (it != types.end());

```

```
                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::YES);
                INFO ("Restrict: " + OsmBarrier::toString(*it));
                REQUIRE (it != types.end());

                it = std::find(types.begin(), types.end(),
                    OsmBarrier::BarrierType::GATE);
                INFO ("Allow: gate");
                REQUIRE (it == types.end());
            }
        }

        WHEN ("asking for costs rules")
        {
            const OsmBarrier::CostsRule r_costs_rule = config.getBarrierCostsRule();

            THEN ("we get CostssRule filled out")
            {
                REQUIRE (r_costs_rule.costs.size() == 13);

                INFO("Costs: border control");
                REQUIRE (r_costs_rule.getCost(
                    OsmBarrier::BarrierType::BORDER_CONTROL) == 120);

                INFO("Costs: bump gate");
                REQUIRE (r_costs_rule.getCost(
                    OsmBarrier::BarrierType::BUMP_GATE) == 30);

                INFO("Costs: bus trap");
                REQUIRE (r_costs_rule.getCost(
                    OsmBarrier::BarrierType::BUS_TRAP) == 30);

                INFO("Costs: cattle grid");
                REQUIRE (r_costs_rule.getCost(
                    OsmBarrier::BarrierType::CATTLE_GRID) == 20);

                INFO("Costs: entrance");
                REQUIRE (r_costs_rule.getCost(
                    OsmBarrier::BarrierType::ENTRANCE) == 10);

                INFO("Costs: gate");
                REQUIRE (r_costs_rule.getCost(
                    OsmBarrier::BarrierType::GATE) == 30);

                INFO("Costs: hampshire gate");
                REQUIRE (r_costs_rule.getCost(
                    OsmBarrier::BarrierType::HAMPSHIRE_GATE) == 60);

                INFO("Costs: height restrictor");
                REQUIRE (r_costs_rule.getCost(
                    OsmBarrier::BarrierType::HEIGHT_RESTRICTOR) == 5);

                INFO("Costs: jersey barrier");
                REQUIRE (r_costs_rule.getCost(
                    OsmBarrier::BarrierType::JERSEY_BARRIER) == 10);

                INFO("Costs: lift gate");
```

```
                    REQUIRE (r_costs_rule.getCost(
                        OsmBarrier::BarrierType::LIFT_GATE) == 60);

                    INFO("Costs: sump buster");
                    REQUIRE (r_costs_rule.getCost(
                        OsmBarrier::BarrierType::SUMP_BUSTER) == 30);

                    INFO("Costs: swing gate");
                    REQUIRE (r_costs_rule.getCost(
                        OsmBarrier::BarrierType::SWING_GATE) == 60);

                    INFO("Costs: toll both");
                    REQUIRE (r_costs_rule.getCost(
                        OsmBarrier::BarrierType::TOLL_BOOTH) == 40);

                    INFO("No cost: yes");
                    REQUIRE (r_costs_rule.costsToPass(
                        OsmBarrier::BarrierType::YES) == false);
                }
            }
        }
}
```

D.4 doc

D.4.1 README.md

```
Documentation
=============

This directory contains a directory for the project's report, and a directory for
 ↪  the UML diagrams.
```

D.5 report

D.5.1 README.md

```
Report
======

The originals for the project's report. Written in
 ↪  [ShareLaTeX](https://www.sharelatex.com).
```

D.6 uml

D.6.1 README.md

```
UML
===

Class and sequence diagrams to get an idea of the concepts, although not 100%
 ↪  accurate.

Diagrams are created in `draw.io`, exported as  `.svg` files, opened in
 ↪  `Inkscape` and saved as `.pdf` files (for usage in report).
```

D.7 graph

D.7.1 README.md

```
graph
=====

The `graph` package consists of classes for representing graphs.

## GraphBuilder
The GraphBuilder is responsible for buildign graphs and linegraphs. It takes a
↪  `Topology` and a `Configuration`and uses them for building a `graph` and
↪  `linegraph` based on [Boost adjacency
↪  lists](http://www.boost.org/doc/libs/1_54_0/libs/graph/doc/adjacency_list.html).
↪  The `GraphBuilder` class holds several `maps` that connects the original
↪  Edges and Vertices to those used internally in the Boost graphs, so that it
↪  is possible to backtrack information about those elements. The internal Boost
↪  types keeps some properties
↪  ["bundled"](http://www.boost.org/doc/libs/1_54_0/libs/graph/doc/bundles.html),
↪  instead of as "interior" properties.

The ordinary `graph` is a directed graph that connects the _**vertices**_ and
↪  _**edges**_ from the topology. The `linegraph` transforms that graph to an
↪  edge-based graph that turns the graph's edges into _**nodes**_ in the
↪  linegraph, and those edges are connected with _**lines**_.

### Topology
`Topology` is a class holding `Edges` and `Vertices` for the topology fetched
↪  from the `MapProvider`. It simply states which `Vertices` are connected by
↪  which `Edges`, without any costs or restrictions or directions. When created
↪  it validates that the `source` and `target` Vertices of the Edges actually
↪  exists in the topology.

### Edge
The Edge holds some relevant data from the topology. It has an `id`, and a field
↪  for which the original `osm_id` was before building the database topology. It
↪  also holds id to `source` Vertex and id to `target` Vertex, some data about
↪  the geometry and the "road" such as number of lanes, a structure for costs
↪  and optionally for restrictions.

### EdgeCost
The cost for travel among an edge is the number of seconds it takes. The base for
↪  this calculation is of course dependent on the length of the edge, and the
↪  speed. The speed an be set as an explicit `maxspeed` restriction, or by
↪  looking up the configuration for a `surface` if such is stated, else the
↪  speed is found by a look up for the default speed for the "highway type"
↪  (road category).

The travel time cost can than be modified by barriers, speed bumps, traffic
↪  lights ... on the edge (or points that can be applied on the edge).

### EdgeRestriction
The `EdgeRestriction` keeps track of restrictions that can be imposed on
an edge. Those restrictions are:

- **Vehicle properties**: weight, height, length, width, maxspeed.
- **General access**: [OSM wiki for
↪  access](http://wiki.openstreetmap.org/wiki/Key:access).
```

```
- **Vehicle type access**: as for General access, but specified for a category of vehicles, such as `motorcar` or `goods`.
- **Barrier**: if the edge is blocked with some kind of barrier.
- **Turning restrictions**: [OSM wiki for turn restrictions](http://wiki.openstreetmap.org/wiki/Relation:restriction).
- **Disused**: if the edge (road) is marked as no longer in use.
- **NoExit**: if the edge has no exit, it should not be used for building a linegraph.

(Turn restriction via other edges and not just via a vertex are difficult. At the time when converting the topology to a line graph it is impossible to have the relevant information. The solution is to set a flag on the Edge that there exist a VIA_WAY restriction that must be taken into account when routing, and the routing module must look up and make its own decisions somehow.)

### Vertex
The Vertex class is simple with just an `id` and a `point` location.

### TurnCostCalculator
This is a static helper class that assist in calculating the cost of making turns, when transforming the _graph_ into a _linegraph_. The cost for making turns are dependent on the angle of the turn, the category of the roads and the size of the vehicle. Other factors can be added.

### Exceptions
`GraphException` is the main public exception to be thrown from this package. `RestrictionsException` and `TopologyException` are thrown when building those classes, but not as exposed externally.
```

D.7.2 Edge.h

```cpp
/**  Data structure for edges in Topology.
 *
 * #include "Edge.h"
 *
 * @author  Jonas Bergman
 */

#ifndef GRAPH_EDGE_H_
#define GRAPH_EDGE_H_

// SYSTEM INCLUDES
//
#include <limits>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//
#include "Vertex.h"
#include "../config/Configuration.h"
#include "../osm/OsmHighway.h"
#include "../osm/OsmId.h"
#include "EdgeCost.h"
#include "Speed.h"

```

```
// FORWARD REFERENCES
//
typedef long    EdgeIdType;
class           EdgeRestriction;

/**
 * Data structure for edges in the topology.
 */
class Edge
{
public:
// TYPES
    //------------------------------------------------------------------------
    enum DirectionType
    {
        BOTH,       // bidirectional
        TO_FROM,    // one-way: from Source to Target
        FROM_TO     // one-way: from Target to Source
    };

    //------------------------------------------------------------------------
    /** A data structure for geometric information for the Edge.
     * Bearing is the compass direction in degrees at source and at target.
     */
    struct GeomData
    {
        double      length {1.0};
        Point       centerPoint;
        int         sourceBearing {0};
        int         targetBearing {0};

        /** Constructor. */
        GeomData() = default;

        /** Constructor. */
        GeomData(double length,
                 Point  centerPoint,
                 int    sourceBearing,
                 int    targetBearing);
    };

    //------------------------------------------------------------------------
    /** A data structure for road related information for the Edge.
     */
    struct RoadData
    {
        DirectionType               direction {BOTH};
        size_t                      nrLanes   {1};
        OsmHighway::HighwayType     roadType  {OsmHighway::HighwayType::ROAD};

        /** Constructor. */
        RoadData() = default;

        /** Constructor. */
        RoadData(DirectionType direction, size_t nrLanes);

        /** Print this information. */
```

```
        void print(std::ostream& os) const;
    };

    static const EdgeIdType MAX_ID;

// LIFECYCLE
    /** Constructor.
     */
    Edge() = delete;

    /** Constructor.
     * @param       id                    Id for this Edge
     * @param   osmId       The original OsmId this edge belongs to.
     * @param       source              Source vertex
     * @param       target              Target vertex
     * @param   geomData    Geometric data for the edge.
     * @param   roadData    Road data for the edge.
     */
    Edge(EdgeIdType      id,
         OsmIdType       osmId,
         VertexIdType    source,
         VertexIdType    target,
         GeomData        geomData,
         RoadData        roadData);

    /** Constructor.
     * Using default values for geometry and road.
     * @param       id                Id for this Edge
     * @param   osmId   The original OsmId this edge belongs to.
     * @param       source        Source vertex
     * @param       target        Target vertex
     */
    Edge(EdgeIdType      id,
         OsmIdType       osmId,
         VertexIdType    source,
         VertexIdType    target);

    /** Move constructor.
     * @param   from    The Edge to make a move of.
     */
    Edge(Edge&& from);

    /** Destructor.
     */
    ~Edge();

// OPERATORS
    /** Textual output of Edge.
     */
    friend
    std::ostream&     operator<<(std::ostream& os, const Edge& rEdge);

// OPERATIONS
    /** Set the Geometric data for this edge.
     * @param   geomData    The GeomData to use.
     */
    void                 setGeomData(GeomData geomData);
```

```

    /** Set the Road data for this edge.
     * @param   roadData    The RoadData to use.
     */
    void                setRoadData(RoadData roadData);

    /** Set the OsmId corresponding to this edge.
     * @param   osmId   The OsmId to set.
     */
    void                setOsmId(OsmIdType osmId);

    /** Set the restrictions for this edge.
     * @param   pRestrictions   The restrictions for this edge.
     */
    void                setRestrictions(EdgeRestriction* pRestrictions);

    /** Set the speed for the edge in this actual configuration.
     * @param   speed   The speed to set in km/h.
     */
    void                setSpeed(Speed speed);

    /** Remove the restrictions for this edge.
     */
    void                clearCostsAndRestrictions();

    /** Parse a string into an EdgeIdType.
     *  @param  idString    The string representing the id.
     *  @return The corresponding edge id.
     *  @throw  std::invalid_argument
     *  @throw  std::out_of_range
     */
    static EdgeIdType parse(const std::string& idStr);

// ACCESSORS
    /**
     * @return  The id of this edge.
     */
    EdgeIdType          id()        const;

    /**
     * @return  The source vertex for this edge.
     */
    VertexIdType        sourceId()  const;

    /**
     * @return  The target vertex of this edge.
     */
    VertexIdType        targetId()  const;

    /**
     * @return  The original OSM id for this edge.
     */
    OsmIdType           osmId()     const;

    /**
     * @return The geometric data for this edge.
     */
```

```
    const GeomData&   geomData()  const;

    /**
     * @return The road data for this edge.
     */
    const RoadData&   roadData()  const;

    /** Get hold of the restrictions associated with the edge.
     * @return  Reference to EdgeRestriction
     * @throw   RestrictionException if no restriction is applied on Edge.
     */
    EdgeRestriction&  restrictions();

    /** Get hold of the restrictions associated with the edge.
     * @return  Reference to EdgeRestriction
     * @throw   RestrictionException if no restriction is applied on Edge.
     */
    const EdgeRestriction&
                      restrictions() const;

    /** Get the structure of different costs for traveling the edge.
     * @return  Reference to EdgeCost
     */
    EdgeCost&         edgeCost();

    /** Get the structure of different costs for traveling the edge.
     * @return  Reference to EdgeCost
     */
    const EdgeCost&   edgeCost() const;

    /**
     * @return  The cost or weight for this edge.
     */
    Cost              cost() const;

    /** The speed must be kept track of because of turn cost calculations,
     * but they are not  part of `RoadData` which are meant to be constant,
     * while the speed varies with configuration.
     * @return  The speed for this edge in km/h
     */
    Speed             speed() const;

// INQUIRY
    /**
     * @return  true if there exists restrictions for this edge.
     */
    bool              hasRestrictions() const;

    /** An edge needs special attention during routing if there exists
     * a turning restriction via other ways (edges).
     * @return  true if there exists a turn restriction via ways.
     */
    bool              hasViaWayRestriction() const;

    /** Check if travel on the Edge is restricted given the configuration.
     * @param   rConfig     Configuration with restriction rules.
     * @return  true        If travel is restricted.
```

```
     * @throws  RestrictionsException
     */
    bool              isRestricted(const Configuration& rConfig) const;

private:
// ATTRIBUTES
    EdgeIdType        mId;      // id in topology
    OsmIdType         mOsmId;
    VertexIdType      mSourceId;
    VertexIdType      mTargetId;
    GeomData          mGeomData;
    RoadData          mRoadData;
    EdgeRestriction*  mpRestrictions;
    EdgeCost          mCost;
    Speed             mSpeed;
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* GRAPH_EDGE_H_ */
```

D.7.3 Edge.cc

```
/*
 * Edge.cc
 *
 *@author   Jonas Bergman
 */

#include "Edge.h"  // class implemented

#include "EdgeRestriction.h"

//============================ TYPES     ====================================
const EdgeIdType Edge::MAX_ID = std::numeric_limits<EdgeIdType>::max();

// Edge::GeomData ------------------------------------------------------------
Edge::GeomData::GeomData(double length,
                         Point  centerPoint,
                         int    sourceBearing,
                         int    targetBearing)
    : length(length),
      centerPoint(centerPoint),
      sourceBearing(sourceBearing),
      targetBearing(targetBearing)
{}

// Edge::RoadData ------------------------------------------------------------
Edge::RoadData::RoadData(DirectionType direction, size_t nrLanes)
    : direction(direction), nrLanes(nrLanes)
{}

void
```

```
Edge::RoadData::print(std::ostream& os) const
{
    os  << "direction: ";

    switch(direction)
    {
        case Edge::DirectionType::BOTH:
            os << "BOTH"; break;
        case Edge::DirectionType::FROM_TO:
            os << "FROM_TO"; break;
        case Edge::DirectionType::TO_FROM:
            os << "TO_FROM"; break;
    }

    os << ", #lanes: " << nrLanes;
    os << ", type: " << OsmHighway::toString(roadType);
}


// Edge ---------------------------------------------------------------------
/////////////////////////////// PUBLIC ///////////////////////////////////////

//============================= LIFECYCLE ====================================
Edge::Edge(EdgeIdType       id,
           OsmIdType        osmId,
           VertexIdType     source,
           VertexIdType     target,
           Edge::GeomData   geomData,
           Edge::RoadData   roadData)
    : mId(id),
      mOsmId(osmId),
      mSourceId(source),
      mTargetId(target),
      mGeomData(geomData),
      mRoadData(roadData),
      mpRestrictions(nullptr),
      mCost(),
      mSpeed()
{ }

Edge::Edge(EdgeIdType       id,
           OsmIdType        osmId,
           VertexIdType     source,
           VertexIdType     target)
    : mId(id),
      mOsmId(osmId),
      mSourceId(source),
      mTargetId(target),
      mGeomData(),
      mRoadData(),
      mpRestrictions(nullptr),
      mCost(),
      mSpeed()
{ }

Edge::Edge(Edge&& from)
    : mId(from.mId),
```

```
      mOsmId(from.mOsmId),
      mSourceId(from.mSourceId),
      mTargetId(from.mTargetId),
      mGeomData(from.mGeomData),
      mRoadData(from.mRoadData),
      mpRestrictions(from.mpRestrictions),
      mCost(),
      mSpeed()
{
    from.mpRestrictions = nullptr;
}

Edge::~Edge()
{
    delete mpRestrictions;
}

//============================ OPERATORS ===================================
std::ostream&
operator<<(std::ostream& os, const Edge& rEdge)
{
    os  << "Edge [id: " << rEdge.id()
        << ", osmId: " << rEdge.osmId()
        << ", source: " << rEdge.sourceId()
        << ", target: " << rEdge.targetId()
        << ", cost: " << rEdge.cost()
        << ", length: " << rEdge.geomData().length
        << ", speed: " << rEdge.speed()
        << "\n   road data: ";
    rEdge.roadData().print(os);

    os  << "]";

    return os;
}


//============================ OPERATIONS ==================================

void
Edge::setGeomData(Edge::GeomData geomData)
{ mGeomData = geomData; }

void
Edge::setRoadData(Edge::RoadData roadData)
{ mRoadData = roadData; }

void
Edge::setOsmId(OsmIdType osmId)
{ mOsmId = osmId; }

void
Edge::setRestrictions(EdgeRestriction* pRestrictions)
{
    delete mpRestrictions;
    mpRestrictions = pRestrictions;
}
```

```
void
Edge::setSpeed(Speed speed)
{
    mSpeed = speed;
}

void
Edge::clearCostsAndRestrictions()
{
    mCost.clearCosts();

    delete mpRestrictions;
    mpRestrictions = nullptr;

    mSpeed = 0;
}

//static
EdgeIdType
Edge::parse(const std::string& idStr)
{
    return static_cast<EdgeIdType>(std::stoul(idStr));
}

//============================= ACESS      ===================================
EdgeIdType
Edge::id() const
{ return mId; }

VertexIdType
Edge::sourceId() const
{ return mSourceId; }

VertexIdType
Edge::targetId() const
{ return mTargetId; }

OsmIdType
Edge::osmId() const
{ return mOsmId; }

const Edge::GeomData&
Edge::geomData() const
{ return mGeomData; }

const Edge::RoadData&
Edge::roadData() const
{ return mRoadData; }


EdgeRestriction&
Edge::restrictions()
{
    if(mpRestrictions == nullptr) {
        mpRestrictions = new EdgeRestriction();
    }
```

```
    return *mpRestrictions;
}

const EdgeRestriction&
Edge::restrictions() const
{
    if(mpRestrictions == nullptr) {
        throw RestrictionsException(std::string("No restriction on edge ")
            + std::to_string(mId));
    }
    return *mpRestrictions;
}

EdgeCost&
Edge::edgeCost()
{
    return mCost;
}

const EdgeCost&
Edge::edgeCost() const
{
    return mCost;
}

Cost
Edge::cost() const
{
    return mCost.getCost();
}

Speed
Edge::speed() const
{
    return mSpeed;
}

//============================ INQUIRY    ===================================
bool
Edge::hasRestrictions() const
{ return mpRestrictions != nullptr; }

bool
Edge::hasViaWayRestriction() const
{
    if(hasRestrictions())
    {
        return mpRestrictions->hasViaWayRestriction();
    }
    return false;
}

bool
Edge::isRestricted(const Configuration& rConfig) const
{
    if(mpRestrictions == nullptr)
    {
```

```
        return false;
    }

    try
    {
        return mpRestrictions->restricts(rConfig);
    }
    catch (RestrictionsException& re)
    {
        re.addEdgeId(std::to_string(mId));
        throw re;
    }
}

/////////////////////////////// PROTECTED  ///////////////////////////////////

/////////////////////////////// PRIVATE    ///////////////////////////////////
```

D.7.4 Vertex.h

```
/**  Data structure for vertices in Topology.
 *
 * #include "Vertex.h"
 *
 * @author  Jonas Bergman
 */

#ifndef GRAPH_VERTEX_H_
#define GRAPH_VERTEX_H_

// SYSTEM INCLUDES
//
#include <limits>
#include <ostream>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//
#include "../util/Point.h"

// FORWARD REFERENCES
//
typedef long        VertexIdType;

/**
 * Data structure for vertices in the topology.
 */
class Vertex
{
public:
// TYPES and CONSTANTS
    static const VertexIdType MAX_ID;
// LIFECYCLE
    /** Constructor.
     * @param   id                 Id for this vertex.
```

```
     * @param    point             The Point (geometry).
     */
    Vertex(VertexIdType id, Point point);

    /** Default constructor. Deleted */
    Vertex() = delete;

    /** Copy constructor. Default. */
    Vertex(const Vertex&) = default;

// OPERATORS
    friend
    std::ostream&   operator<<(std::ostream& os, const Vertex& rVertex);

    bool            operator==(const Vertex& rhs) const;

// OPERATIONS
// ACCESS
    /**
     * @return  The id of this Vertex.
     */
    VertexIdType    id() const;

    /**
     * @return  The coordinates for this Vertex.
     */
    Point           point() const;

//INQUIRY
    /**
     * @return  True if the Vertex has restrictions.
     */
    bool            hasRestrictions() const;

private:
// ATTRIBUTES
    VertexIdType    mId;
    Point           mPoint;
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* GRAPH_VERTEX_H_ */
```

D.7.5 Vertex.cc

```
/*
 * Vertex.cc
 *
 * @author  Jonas Bergman
 */

#include "Vertex.h"  // class implemented
```

```


/////////////////////////////// PUBLIC ////////////////////////////////////////
const VertexIdType Vertex::MAX_ID = std::numeric_limits<VertexIdType>::max();

//============================= LIFECYCLE ====================================
Vertex::Vertex(VertexIdType id, Point point)
    : mId(id), mPoint(point)
{}

//============================= OPERATORS ====================================
std::ostream&
operator<<(std::ostream& os, const Vertex& rVertex)
{
    os  << "Vertex [id: " << rVertex.mId
        << ", point: " << rVertex.mPoint << "]";
    return os;
}

bool
Vertex::operator==(const Vertex& rhs) const
{
    return (rhs.mId == mId) && (rhs.mPoint == mPoint);
}

//============================= OPERATIONS ===================================
//============================= ACESS      ===================================
VertexIdType
Vertex::id() const
{ return mId; }

Point
Vertex::point() const
{ return mPoint; }

//============================= INQUIRY    ===================================
bool
Vertex::hasRestrictions() const
{ return false; }

/////////////////////////////// PROTECTED  ///////////////////////////////////

/////////////////////////////// PRIVATE    ///////////////////////////////////
```

D.7.6 Topology.h

```
/**  A class holding the elements of the topology.
 *
 * #include "Topology.h"
 *
 * @author  Jonas Bergman
 */

#ifndef GRAPH_TOPOLOGY_H_
#define GRAPH_TOPOLOGY_H_

// SYSTEM INCLUDES
```

```cpp
//
#include <map>

// PROJECT INCLUDES
//
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/graph_traits.hpp>

// LOCAL INCLUDES
//
#include "Edge.h"
#include "TopologyException.h"
#include "Vertex.h"
#include "../util/Point.h"


// FORWARD REFERENCES
//

// TYPES
//
/** Mapping of topology vertex id and topology Vertex object. */
typedef std::map<VertexIdType, Vertex>          TopoVertexMapType;

/** Mapping of topology edge id and topology Edge object. */
typedef std::map<EdgeIdType, Edge>              TopoEdgeMapType;

/** Keep track of which topology edges that make up an edge in the original
 * OSM map data, as the OSM edge might have been split into several edges when
 * building the topology.
 */
typedef std::multimap<OsmIdType, EdgeIdType>    OsmIdToTopoIdEdgeMap;

/** This class holds Edges and Vertices such as they are in the database.
 */
class Topology
{
    friend class GraphBuilder;

public:
// LIFECYCLE

    /** Default constructor.
     */
    Topology();

    /** Copy constructor.
     *
     * @param from The value to copy to this object.
     */
    Topology(const Topology& from) = delete;


// OPERATORS
// OPERATIONS

    /** Try to add a vertex to the topology.
```

```
     * If a vertex with the id already exists: return old value.
     * @param   id       Id for the vertex
     * @param   point    The position of the vertex
     * @return  A reference to a vertex with given id
     */
    Vertex&         addVertex(VertexIdType id, Point point);


    /** Try to add an edge to the topology.
     * If an edge with the id already exists: return old value.
     * @param   id          Id for the edge
     * @param   osmId       The original OsmId this edge belongs to.
     * @param   source      Id for source vertex
     * @param   target      Id for target vertex
     * @param   geomData    Geometric data for the edge
     * @param   roadData    Road data for the edge
     * @return  A reference to an edge with given id
     * @throw   Topology Exception if vertices are not in topology.
     */
    Edge&           addEdge(EdgeIdType      id,
                            OsmIdType       osmId,
                            VertexIdType    source,
                            VertexIdType    target,
                            Edge::GeomData  geomData,
                            Edge::RoadData  roadData);

    /** Try to add an edge to the topology.
     * Using default values for geometric and road data.
     * If an edge with the id already exists: return old value.
     * @param   id      Id for the edge
     * @param   osmId   The original OsmId this edge belongs to.
     * @param   source  Id for source vertex
     * @param   target  Id for target vertex
     * @return  A reference to an edge with given id
     * @throw   Topology Exception if vertices are not in topology.
     */
    Edge&           addEdge(EdgeIdType      id,
                            OsmIdType       osmId,
                            VertexIdType    source,
                            VertexIdType    target);

    /** Fetch the vertex with given id.
     * @param   id        Id of the vertex to get
     * @return  Reference to the found vertex
     * @throws  TopologyException if vertex does not exist.
     */
    Vertex&         getVertex(VertexIdType id);
    const Vertex&   getVertex(VertexIdType id) const;

    /** Fetch the edge with given id.
     * @param   id        Id of the edge to get
     * @return  Reference to the found vertex
     * @throws  TopologyException if vertex does not exist.
     */
    Edge&           getEdge(EdgeIdType id);
    const Edge&     getEdge(EdgeIdType id) const;

```

```
    /** Clear everything in the topology: edges and vertices.
     */
    void            clearTopology();

    /** Remove restrictions and costs on all edges in the topology.
     */
    void            clearEdgeCostAndRestrictions();

// ACCESS
    /**
     * @return    the Number of vertices in topology.
     */
    size_t          nrVertices() const;

    /**
     * @return    the Number of vertices in topology.
     */
    size_t          nrEdges() const;

// INQUIRY

protected:
private:
// ATTRIBUTES
    TopoVertexMapType     mVertexMap;
    TopoEdgeMapType       mEdgeMap;
    OsmIdToTopoIdEdgeMap  mOsmEdgeMap;
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* GRAPH_TOPOLOGY_H_ */
```

D.7.7 Topology.cc

```
/*
 * Topology.cc
 *
 * @author  Jonas Bergman
 */

#include "Topology.h"  // class implemented

#include <utility>

/////////////////////////////// PUBLIC ///////////////////////////////////////

//============================= LIFECYCLE ====================================
Topology::Topology()
    : mVertexMap(), mEdgeMap(), mOsmEdgeMap()
{
}
//============================= OPERATORS ====================================
```

```
//============================ OPERATIONS ===================================

Vertex&
Topology::addVertex(VertexIdType id, Point point)
{
    auto res = mVertexMap.emplace(id, Vertex(id, point));
    return res.first->second;
}


Edge&
Topology::addEdge(EdgeIdType        id,
                  OsmIdType         osmId,
                  VertexIdType      source,
                  VertexIdType      target,
                  Edge::GeomData    geomData,
                  Edge::RoadData    roadData)
{
     try
    {
        getVertex(source);
        getVertex(target);
        Edge edge(id, osmId, source, target, geomData, roadData);
        auto res = mEdgeMap.emplace(id, std::move(edge));
        mOsmEdgeMap.insert({osmId, id});
        return res.first->second;
    }
    catch (TopologyException& e)
    {
        throw TopologyException("Cannot add edge: " + std::to_string(id) +
                "." " + e.what());
    }
}

Edge&
Topology::addEdge(EdgeIdType        id,
                  OsmIdType         osmId,
                  VertexIdType      source,
                  VertexIdType      target)
{
    Edge::GeomData gd;
    Edge::RoadData rd;
    return addEdge(id, osmId, source, target, gd, rd);
}

Vertex&
Topology::getVertex(VertexIdType id)
{
    auto it = mVertexMap.find(id);
    if(it == mVertexMap.end()) {
        throw TopologyException("Vertex not found: " + std::to_string(id));
    }
    return it->second;
}

const Vertex&
Topology::getVertex(VertexIdType id) const
```

```
{
    return const_cast<Topology&>(*this).getVertex(id);
}

Edge&
Topology::getEdge(EdgeIdType id)
{
    auto it = mEdgeMap.find(id);
    if(it == mEdgeMap.end()) {
        throw TopologyException("Edge not found: " + std::to_string(id));
    }
    return it->second;
}

const Edge&
Topology::getEdge(EdgeIdType id) const
{
    auto it = mEdgeMap.find(id);
    if(it == mEdgeMap.end()) {
        throw TopologyException("Edge not found: " + std::to_string(id));
    }
    return it->second;
}

void
Topology::clearTopology()
{
    mVertexMap.clear();
    mEdgeMap.clear();
    mOsmEdgeMap.clear();
}

void
Topology::clearEdgeCostAndRestrictions()
{
    for(auto& it : mEdgeMap)
    {
        it.second.clearCostsAndRestrictions();
    }
}


//============================= ACESS      ===================================
size_t
Topology::nrVertices() const
{
    return mVertexMap.size();
}

size_t
Topology::nrEdges() const
{
    return mEdgeMap.size();
}
//============================= INQUIRY    ===================================
/////////////////////////////// PROTECTED  ///////////////////////////////////

```

```
////////////////////////////// PRIVATE    ///////////////////////////////////
```

D.7.8 GraphBuilder.h

```
/**  GraphBuilder.
 *
 * #include "GraphBuilder.h"
 *
 * @author  Jonas Bergman
 */

#ifndef GRAPH_GRAPHBUILDER_H_
#define GRAPH_GRAPHBUILDER_H_

// SYSTEM INCLUDES
//
#include <algorithm>
#include <map>
#include <ostream>

// PROJECT INCLUDES
//
#include <boost/graph/adjacency_list.hpp>

// LOCAL INCLUDES
//
#include "Edge.h"
#include "EdgeRestriction.h"
#include "GraphException.h"
#include "Topology.h"
#include "TurnCostCalculator.h"
#include "Vertex.h"
#include "../config/Configuration.h"
#include "../util/Logging.h"

// FORWARD REFERENCES
//

// TYPES
typedef EdgeIdType  NodeIdType;
typedef EdgeIdType  LineIdType;

/**
 * Map the GraphEdges to the original Edge id in the Topology.
 * Also indicate if the GraphEdge is the same or opposite direction to
 * the graph in the topology.
 */
struct GraphEdge
{
    EdgeIdType      graphEdgeId;
    EdgeIdType      topoEdgeId;
    bool            oppositeDirection {false};
};

/**
 * A Node in the LineGraph corresponds directly to an Edge in the original
 * GraphBuilder and topology. It is connected to another Node (Edge) if both the
```

```
 * edges are adjacent and there is no restriction in the Vertex for travel
 * along them.
 * lgNodeId === graphEdgeId
 */
typedef GraphEdge   LineGraphNode;

/**
 * Map the GraphVertices to the original Vertex id in the Topology.
 */
struct GraphVertex
{
    VertexIdType    graphVertexId;
    VertexIdType    topoVertexId;
};

/**
 * A LineGraphLine corresponds to a travel along an incoming edge,
 * via a vertex and out an outgoing edge.
 * The cost is the cost of travel on the incoming edge and the turn cost
 * at the vertex.
 * The Line connects two edges in the graph with an allowed turn in between.
 */
struct LineGraphLine
{
    NodeIdType      lgSourceNodeId;
    NodeIdType      lgTargetNodeId;
    VertexIdType    topoViaVertexId;
    double          cost;
};

/** The 'normal' vertex based graph type. */
typedef boost::adjacency_list
    < boost::listS, boost::vecS, boost::directedS,
    GraphVertex, GraphEdge >                                    GraphType;
/** The edge based graph type. */
typedef boost::adjacency_list
    < boost::listS, boost::vecS, boost::directedS,
    LineGraphNode, LineGraphLine >                              LineGraphType;

/** A vertex in the normal graph. */
typedef boost::graph_traits<GraphType>::vertex_descriptor      VertexType;
/** An edge in the normal graph. */
typedef boost::graph_traits<GraphType>::edge_descriptor        EdgeType;

/** A node in the line graph. */
typedef boost::graph_traits<LineGraphType>::vertex_descriptor  NodeType;
/** An edge in the line graph. */
typedef boost::graph_traits<LineGraphType>::edge_descriptor    LineType;

/** Mapping of a topology vertex id and graph vertex object. */
typedef std::map<VertexIdType, VertexType>   TopoVertexIdToGraphVertexMapType;
/** Mapping of a topology edge id and graph edge object. */
typedef std::multimap<EdgeIdType, EdgeType>  TopoEdgeIdToGraphEdgeMapType;
/** Mapping of a graph edge id and linegraph node object. */
typedef std::map<EdgeIdType, NodeType>       GraphEdgeIdToNodeMapType;
```

```
/**
 * A class for building (Boost) Graph and LineGraph from a Topology and
 * a Configuration with optional Restrictions and Costs applied.
 */
class GraphBuilder
{
public:
// LIFECYCLE
    /** Constructor.
     * Disabled.
     */
    GraphBuilder() = delete;

    /** Constructor.
     * GraphBuilder should be based on the supplied topology.
     * @param   rTopology       The topology to use as basis for the graph.
     * @param   rConfig         The configuration used for topology and all.
     * @param   useRestrictions If the graph should be built with restrictions or not.
     */
    GraphBuilder(
        Topology& rTopology,
        const Configuration& rConfig,
        bool useRestrictions = true);

    /** Copy constructor.
     * Disabled.
     */
    GraphBuilder(const GraphBuilder& from) = delete;

    /** Destructor.
     */
    ~GraphBuilder();

// OPERATORS
    /** Output operator to print to a stream.
     */
    friend
    std::ostream&       operator<<(std::ostream& os, const GraphBuilder& rGraph);

// OPERATIONS
// ACCESS
    /**
     * @return  The number of Vertices in the Graph.
     */
    size_t              nrVertices() const;

    /**
     * @return  The number of Edges in the Graph.
     */
    size_t              nrEdges() const;

    /**
     * @return  The number of Nodes in the LineGraph.
     */
    size_t              nrNodes() const;

    /**
```

```
     * @return  The number of Nodes in the LineGraph.
     */
    size_t              nrLines() const;

    /** Builds graph if necessary before returning.
     * @return  The Boost Graph representation of the Graph.
     * @throws  GraphException if something goes wrong building the graph.
     */
    const GraphType&    getBoostGraph();

    /** Get a reference to the line graph.
     * @return  The Boost Graph representation of the LineGraph.
     * @throws  GraphException if something goes wrong building the graph.
     */
    LineGraphType&       getBoostLineGraph();
    const LineGraphType& getBoostLineGraph() const;

    /** Get access to the topology that is the base for the graph.
     * @return  The Topology
     */
    const Topology&     getTopology() const;

// INQUIRY
    /**
     * @return  true    If graph has a vertex with given id.
     */
    bool                hasVertex(VertexIdType vertexId) const;

    /**
     * @return  true    If LineGraph has a node with given id.
     */
    bool                hasNode(EdgeIdType nodeId) const;

    /** Get an already existing Node from the LineGraph.
     * @param   id  The Edge id (== the Node id).
     * @param   The LineGraph Node.
     * @throw   GraphException if there is no Node with that id.
     */
    const NodeType&     getLineGraphNode(NodeIdType id) const;

    /**
     * @return  true    If graph was built with restrictions.
     */
    bool                isRestricted() const;

    /** Output information about # vertices, edges, nodes, lines.
     */
    void                printGraphInformation(std::ostream& os) const;

protected:

private:
// HELPERS

    // buildGraph() -----------------------------------------------------------
    // Used when constructing the internal Boost graph representation
    // from the Topology.
```

```
    /** Build the graph by adding vertices and edges from the topology. */
    void                buildGraph();

    /** Add the topology vertices to the graph, respecting restrictions.
     * Helper for 'buildGraph()'.
     */
    void                addTopoVerticesToGraph();

    /** Add the topology edges to the graph, respecting restrictions.
     * Helper for 'buildGraph()'.
     */
    void                addTopoEdgesToGraph();

    /** Check if an edge is restricted
     * @param   rEdge   Reference to edge
     * @return  bool
     */
    bool                isEdgeRestricted(const Edge& rEdge) const;

    /** Add the correct number of directed edges from the topo Edge.
     * @param   rEdge        The topological graph data
     * @param   rNewEdgeId   The running id for the graph's directed edges.
     */
    void                addDirectedGraphEdges(
                            const Edge& rEdge,
                            EdgeIdType& rNewEdgeId);

    /** Add a directed edge from source to target.
     * Helper for 'addTopoEdgesToGraph()'.
     * @param   id      The edge's topology id.
     * @param   source  The source vertex.
     * @param   target  The target vertex.
     * @param   e_ix    The running index amongst edges added to graph.
     * @param   oppositeDirection
     *                  If the directed edge runs opposite of the original
     *                  edge direction as specified in the topology.
     */
    void                addDirectedEdge(
                            EdgeIdType id,
                            const VertexType& source,
                            const VertexType& target,
                            EdgeIdType ix,
                            bool  oppositeDirection);

    /** Get the graph vertex corresponding to a given id.
     * @param   id      The vertex' topology id.
     * @return  Reference to the Graph vertex corresponding to id.
     * @throw   GraphException if there is no corresponding vertex to id.
     */
    const VertexType&   getGraphVertex(VertexIdType id) const;

    // buidlLineGraph() ------------------------------------------------------
    // Used when transforming the Graph to a LineGraph

    /** Start converting the GraphBuilder to a LineGraph.
     */
```

```
    void                buildLineGraph();

    /** Add Edges from the graph as Nodes in the Linegraph.
     * Helper for 'buildLineGraph()'
     */
    void                addGraphEdgesToLineGraph();

    /** Actually add a graph edge as a linegraph node, checking if it already
     * exists or not.
     * @param   rGraphEdge  The Edge to add to the LineGraph as Node.
     * @param   rNode       The Node corresponding to the edge returned here.
     */
    void                addGraphEdgeAsLineGraphNode(
                            const EdgeType& rGraphEdge,
                            NodeType&       rNode);


    /** Connect the newly added Node to all Nodes it should be connected to,
     * that is look up which outgoing edges there are from the Edge's (node's)
     * target vertex, and if there are no restrictions: add the Edge as a Node
     * to the LineGraph and add a Line between the Nodes.
     * @param   rSourceNode     The Node to add Lines from.
     * @parma   rViaVertex      Are there any restrictions in the vertex?
     * @throw   GraphException
     */
    void                connectSourceNodeToTargetNodesViaVertex(
                            const NodeType& rSourceNode,
                            const VertexType& rViaVertex);

    /** Extract LineGraphNode data from the LineGraph.
     * @param   rNode     The descriptor in the LineGraph
     * @return  a LineGraphNode
     */
    LineGraphNode       getLineGraphNodeData(const NodeType& rNode) const;

    /** Add a line in the LineGraph, connecting the source and target nodes.
     * @param   rSourceNode
     * @param   rTargetNode
     * @return  the added line
     * @throw   GraphException
     */
    LineType            addLineGraphLine(
                            const NodeType& rSourceNode,
                            const NodeType& rTargetNode);

    /** Add meta data ids for source, target and vertex to the newly added Line.
     * @param   rLine
     * @param   sourceId
     * @param   targetId
     * @param   viaVertexId
     */
    void                addLineMetaIds(
                            const LineType& rLine,
                            EdgeIdType sourceId,
                            EdgeIdType targetId,
                            VertexIdType viaVertexId);
```

```
    /** Add the meta information about the cost to the new line.
     * @param   rLine
     * @param   rSourceEdge     The Source Edge
     * @param   targetId        The id of the target edge in topology
     */
    void                addLineMetaCost(
                            const LineType& rLine,
                            const Edge&     rSourceEdge,
                            EdgeIdType      targetId);

    /** Calculate the cost for making a turn from source edge to target.
     * Helper to `connectSourceNodeToTargetNodesViaVertex()`.
     * @param   sourceEdgeId    The edge (and node) id of the source.
     * @param   targetEdgeId    The edge (and node) id of the target.
     */
    double              calculateTurnCost(
                            EdgeIdType sourceEdgeId,
                            EdgeIdType targetEdgeId) const;

    /**
     * @param   edgeId  Id to edge to look up.
     * @return  true if this edge has no exits, meaning it is no use adding it.
     */
    bool                edgeHasNoExit(EdgeIdType edgeId);

    /**
     * @return A vector of all Edges going out from a vertex.
     */
    std::vector<EdgeIdType>
                        getOutEdges(VertexIdType vertexId) const;

    /**
     * @param   rSourceNode     The LineGraph Node
     * @return  A vector of all restricted edges from this Edge.
     */
    std::vector<EdgeIdType>
                        getRestrictedTargets(
                            const LineGraphNode& rSourceNode) const;

    /** Look through the targets from a source to find which are restricted
     * and add them to a collection of restricted.
     * @param   rSsourceEdge        The source edge.
     * @param   rTargets            Targets from that source.
     * @param   rRestrictedTargets  A collection to build up.
     */
    void                findRestrictedTargets(
                            const Edge&                     rSourceEdge,
                            const std::vector<EdgeIdType>&  rTargets,
                            std::vector<EdgeIdType>&        rRestrictedTargets) const;

    /** Add the turning restricted targets to the other restricted targets.
     * @param   rSourceEdge         The source edge.
     * @param   rRestrictedTargets  The collection of restricted targets.
     */
    void                addTurningRestrictedTargets(
                            const Edge&               rSource,
                            std::vector<EdgeIdType>&  rRestrictedTargets) const;
```

```

    /**
     * @return  true if this target edge has restricted access from the source.
     */
    bool                isTargetRestricted(
                            const std::vector<EdgeIdType>&  rRestrictedTargets,
                            EdgeIdType                      targetId) const;

    void                printVertices(std::ostream& os) const;
    void                printEdges(std::ostream& os)    const;
    void                printNodes(std::ostream& os)    const;
    void                printLines(std::ostream& os)    const;

// ATTRIBUTES
    GraphType                           mGraph;
    LineGraphType                       mLineGraph;
    TopoVertexIdToGraphVertexMapType    mIdToVertexMap;     // map original id to GraphVertex
    TopoEdgeIdToGraphEdgeMapType        mIdToEdgeMap;       // map original id to GraphEdge
    GraphEdgeIdToNodeMapType            mEdgeIdToNodeMap;   // map GraphEdge.id to LineGraphNode
    Topology&                           mrTopology;
    const Configuration&                mrConfiguration;
    mutable boost::log::sources::severity_logger
        <boost::log::trivial::severity_level>
                                        mLog;
    bool                                mUseRestrictions;

// CONSTANTS
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* GRAPH_GRAPHBUILDER_H_ */
```

D.7.9 GraphBuilder.cc

```
/*
 * GraphBuilder.cc
 *
 * @author  Jonas Bergman
 */

#include "GraphBuilder.h"  // class implemented

#include <typeinfo>

/////////////////////////////// PUBLIC ///////////////////////////////////////

//============================= LIFECYCLE ====================================
GraphBuilder::GraphBuilder(
    Topology& rTopology,
    const Configuration& rConfig,
    bool useRestrictions)
    : mGraph(),
```

```
      mLineGraph(),
      mIdToVertexMap(),
      mIdToEdgeMap(),
      mrTopology(rTopology),
      mrConfiguration(rConfig),
      mLog(),
      mUseRestrictions(useRestrictions)
{
    Logging::initLogging();
    boost::log::add_common_attributes();

    buildGraph();
    buildLineGraph();
}

GraphBuilder::~GraphBuilder()
{
}

//============================== OPERATORS ===================================
std::ostream&
operator<<(std::ostream& os, const GraphBuilder& rGraph)
{
    rGraph.printGraphInformation(os);

    os << std::endl << "Vertices: " << std::endl;
    rGraph.printVertices(os);

    os << std::endl << "Edges: " << std::endl;
    rGraph.printEdges(os);

    os << std::endl << "Nodes: " << std::endl;
    rGraph.printNodes(os);

    os << std::endl << "Lines: " << std::endl;
    rGraph.printLines(os);

    return os;
}
//============================== OPERATIONS ==================================
//============================== ACESS      ==================================
size_t
GraphBuilder::nrVertices() const
{
    return mIdToVertexMap.size();
}

size_t
GraphBuilder::nrEdges() const
{
    return mIdToEdgeMap.size();
}

size_t
GraphBuilder::nrNodes() const
{
    return boost::num_vertices(mLineGraph);
```

```
}

size_t
GraphBuilder::nrLines() const
{
    return boost::num_edges(mLineGraph);
}

const GraphType&
GraphBuilder::getBoostGraph()
{
    return mGraph;
}

LineGraphType&
GraphBuilder::getBoostLineGraph()
{
    return mLineGraph;
}

const LineGraphType&
GraphBuilder::getBoostLineGraph() const
{
    return mLineGraph;
}

const Topology&
GraphBuilder::getTopology() const
{
    return mrTopology;
}

//============================ INQUIRY    ===================================
bool
GraphBuilder::hasVertex(VertexIdType vertexId) const
{
    const auto& it = mIdToVertexMap.find(vertexId);
    return (it != mIdToVertexMap.end());
}

bool
GraphBuilder::hasNode(EdgeIdType nodeId) const
{
    const auto& it = mEdgeIdToNodeMap.find(nodeId);
    return (it != mEdgeIdToNodeMap.end());
}

const NodeType&
GraphBuilder::getLineGraphNode(NodeIdType id) const
{
    const auto& res = mEdgeIdToNodeMap.find(id);
    if(res == mEdgeIdToNodeMap.end())
    {
        throw GraphException("Graph:getLineGraphNode: Missing node: "
            + std::to_string(id));
    }
    return res->second;
```

```
}

bool
GraphBuilder::isRestricted() const
{
    return mUseRestrictions;
}

/////////////////////////////// PROTECTED  ///////////////////////////////////

/////////////////////////////// PRIVATE    ///////////////////////////////////
void
GraphBuilder::buildGraph()
{
    addTopoVerticesToGraph();
    addTopoEdgesToGraph();
}

void
GraphBuilder::addTopoVerticesToGraph()
{
    VertexIdType v_ix = 0;
    for(const auto& vertexpair : mrTopology.mVertexMap)
    {
        VertexType v = boost::add_vertex(mGraph);
        mIdToVertexMap.insert({vertexpair.second.id(), v});
        mGraph[v].graphVertexId = v_ix;
        mGraph[v].topoVertexId = vertexpair.second.id();
        ++v_ix;
    }
}

void
GraphBuilder::addTopoEdgesToGraph()
{
    EdgeIdType e_ix = 0;

    for(const auto& edgepair : mrTopology.mEdgeMap)
    {
        const Edge& e = edgepair.second;

        if(isEdgeRestricted(e))
        {
            continue;
        }

        addDirectedGraphEdges(e, e_ix);
    }
}


bool
GraphBuilder::isEdgeRestricted(const Edge& rEdge) const
{
    if(mUseRestrictions && rEdge.isRestricted(mrConfiguration))
    {
        BOOST_LOG_SEV(mLog, boost::log::trivial::info)
```

```
                                << "Graph:addTopoEdgeToGraph(): "
                                << "Restricted Edge id " << rEdge.id();
        return true;
    }
    return false;
}

void
GraphBuilder::addDirectedGraphEdges(const Edge& rEdge, EdgeIdType& rNewEdgeId)
{
    const VertexType& s = getGraphVertex(rEdge.sourceId());
    const VertexType& t = getGraphVertex(rEdge.targetId());

    // add all lanes in forward direction
    if(rEdge.roadData().direction == Edge::DirectionType::FROM_TO
        || rEdge.roadData().direction == Edge::DirectionType::BOTH)
    {
        for(size_t lane = 1; lane <= rEdge.roadData().nrLanes; ++lane) {
            addDirectedEdge(rEdge.id(), s, t, rNewEdgeId, false);
            ++rNewEdgeId;
        }
    }

    // add all lanes in backward direction
    if(rEdge.roadData().direction == Edge::DirectionType::TO_FROM
        || rEdge.roadData().direction == Edge::DirectionType::BOTH)
    {
        for(size_t lane = 1; lane <= rEdge.roadData().nrLanes; ++lane) {
            addDirectedEdge(rEdge.id(), t, s, rNewEdgeId, true);
            ++rNewEdgeId;
        }
    }
}

void
GraphBuilder::addDirectedEdge(
    EdgeIdType          id,
    const VertexType&   source,
    const VertexType&   target,
    EdgeIdType          e_ix,
    bool                oppositeDirection)
{
    const auto& res = boost::add_edge(source, target, mGraph);
    if(res.second == true)
    {
        mIdToEdgeMap.insert({id, res.first});
        mGraph[res.first].graphEdgeId = e_ix;
        mGraph[res.first].topoEdgeId = id;
        mGraph[res.first].oppositeDirection = oppositeDirection;
    }
    else
    {
        throw GraphException("Graph:addDirectedEdge: cannot add edge: "
            + std::to_string(id));
    }
}
```

```
const VertexType&
GraphBuilder::getGraphVertex(VertexIdType id) const
{
    const auto& res = mIdToVertexMap.find(id);
    if(res == mIdToVertexMap.end())
    {
        throw GraphException("Graph:getGraphVertex: Missing vertex: "
            + std::to_string(id));
    }
    return res->second;
}

void
GraphBuilder::buildLineGraph()
{
    mLineGraph.clear();
    addGraphEdgesToLineGraph();
}

void
GraphBuilder::addGraphEdgesToLineGraph()
{
    // iterate through edges: add as Node.
    for(auto e_it = boost::edges(mGraph);
        e_it.first != e_it.second;
        ++e_it.first)
    {
        const EdgeType& edge = *(e_it.first);

        NodeType node;
        addGraphEdgeAsLineGraphNode(edge, node);

        // look up targetId vertex.
        VertexType via_vertex = boost::target(edge, mGraph);

        // connect all possible travels from 'edge' via the vertex
        connectSourceNodeToTargetNodesViaVertex(node, via_vertex);
    }
}

void
GraphBuilder::addGraphEdgeAsLineGraphNode(const EdgeType& rGraphEdge,
                                          NodeType& rNode)
{
    EdgeIdType e_graph_id =
        boost::get(&GraphEdge::graphEdgeId, mGraph, rGraphEdge);
    EdgeIdType e_topo_id  =
        boost::get(&GraphEdge::topoEdgeId, mGraph, rGraphEdge);

    if(!hasNode(e_graph_id))
    {
        rNode = boost::add_vertex(mLineGraph);
        mLineGraph[rNode].graphEdgeId = e_graph_id;
        mLineGraph[rNode].topoEdgeId = e_topo_id;
        mEdgeIdToNodeMap.insert({e_graph_id, rNode});
    }
    else
```

```
    {
        rNode = getLineGraphNode(e_graph_id);
    }
}


void
GraphBuilder::connectSourceNodeToTargetNodesViaVertex(
    const NodeType& rSourceNode,
    const VertexType& rViaVertex)
{
    // SOURCE
    LineGraphNode source_node = getLineGraphNodeData(rSourceNode);

    if(edgeHasNoExit(source_node.topoEdgeId))
    {
        return;
    }

    // get the edge corresponding to the node
    Edge& source_edge = mrTopology.getEdge(source_node.topoEdgeId);

    // VIA
    VertexIdType via_topo_vertex_id =
        boost::get(&GraphVertex::topoVertexId, mGraph, rViaVertex);

    // TARGET
    // get targets that are restricted
    std::vector<EdgeIdType> restricted_targets =
        getRestrictedTargets(source_node);

    // look at all out edges from the via-vertex
    for(auto target_it = boost::out_edges(rViaVertex, mGraph);
        target_it.first != target_it.second;
        ++target_it.first)
    {
        const EdgeType& target = *(target_it.first);
        EdgeIdType target_topo_id =
            boost::get(&GraphEdge::topoEdgeId, mGraph, target);

        if(!isTargetRestricted(restricted_targets, target_topo_id))
        {
            // add nodes to LineGraph
            NodeType target_node;
            addGraphEdgeAsLineGraphNode(target, target_node);

            NodeIdType target_edge_id =
                boost::get(&LineGraphNode::graphEdgeId, mLineGraph, target_node);

            // add Line between Nodes to the LineGraph
            try
            {
                LineType line = addLineGraphLine(rSourceNode, target_node);
                addLineMetaIds(
                    line,
                    source_node.graphEdgeId,
                    target_edge_id,
```

```
                    via_topo_vertex_id);
                addLineMetaCost(line, source_edge, target_topo_id);
            }
            catch (GraphException& ge)
            {
                throw GraphException(
                    "Graph:connectSourceNodeToTargetNodesViaVertex: source: "
                    + std::to_string(source_node.graphEdgeId)
                    + ", target: " + std::to_string(target_edge_id));
            }
        }
        else // log restricted targetId
        {
            Edge& s = mrTopology.getEdge(source_node.topoEdgeId);
            Edge& t = mrTopology.getEdge(target_topo_id);
            BOOST_LOG_SEV(mLog, boost::log::trivial::info)
            << "Graph:connectSourceNodeToTargetNodesViaVertex(): Restricted: "
            << "Source: " << source_node.topoEdgeId << " (osm: " << s.osmId()
            << ") , Target: " << target_topo_id << " (osm: " << t.osmId() << ")";
        }
    }
}

LineGraphNode
GraphBuilder::getLineGraphNodeData(const NodeType& rNode) const
{
    LineGraphNode node;
    node.topoEdgeId = boost::get(&LineGraphNode::topoEdgeId, mLineGraph, rNode);
    node.graphEdgeId = boost::get(&LineGraphNode::graphEdgeId, mLineGraph, rNode);
    node.oppositeDirection =
        boost::get(&LineGraphNode::oppositeDirection, mLineGraph, rNode);

    return node;
}

LineType
GraphBuilder::addLineGraphLine(const NodeType& rSourceNode,
                               const NodeType& rTargetNode)
{
    const auto& line_add =
        boost::add_edge(rSourceNode, rTargetNode, mLineGraph);
    if(line_add.second == true)
    {
        return line_add.first;
    }
    else // could not add the line to the linegraph
    {
        throw GraphException("GraphBuilder:addLineGraphLine");
    }
}

void
GraphBuilder::addLineMetaIds(
    const LineType& rLine,
    EdgeIdType      sourceId,
    EdgeIdType      targetId,
    VertexIdType    viaVertexId)
```

```
{
    mLineGraph[rLine].lgSourceNodeId = sourceId;
    mLineGraph[rLine].lgTargetNodeId = targetId;
    mLineGraph[rLine].topoViaVertexId = viaVertexId;
}

void
GraphBuilder::addLineMetaCost(
    const LineType& rLine,
    const Edge&     rSourceEdge,
    EdgeIdType      targetId)
{
    mLineGraph[rLine].cost =
        rSourceEdge.cost() +
        calculateTurnCost(rSourceEdge.id(), targetId);
}

double
GraphBuilder::calculateTurnCost(EdgeIdType sourceEdgeId,
                                EdgeIdType targetEdgeId) const
{
    const Edge& source = mrTopology.getEdge(sourceEdgeId);
    const Edge& target = mrTopology.getEdge(targetEdgeId);
    return TurnCostCalculator::getTurnCost(source, target, mrConfiguration);
}

bool
GraphBuilder::edgeHasNoExit(EdgeIdType edgeId)
{
    Edge& e = mrTopology.getEdge(edgeId);
    if(e.hasRestrictions() && e.restrictions().hasNoExitRestriction())
    {
        return true;
    }
    return false;
}

std::vector<EdgeIdType>
GraphBuilder::getOutEdges(VertexIdType vertexId) const
{
    std::vector<EdgeIdType> out_edges;
    VertexType graphVertex = getGraphVertex(vertexId);
    auto edge_iterators = boost::out_edges(graphVertex, mGraph);
    while(edge_iterators.first != edge_iterators.second) {
        const EdgeType& e = *(edge_iterators.first);
        EdgeIdType edgeId = boost::get(&GraphEdge::topoEdgeId, mGraph, e);
        out_edges.push_back(edgeId);
        ++edge_iterators.first;
    }
    return out_edges;
}

std::vector<EdgeIdType>
GraphBuilder::getRestrictedTargets(const LineGraphNode& rSourceNode) const
{
    std::vector<EdgeIdType> restricted_targets;

```

```
    // Find all out edges from the targetId vertex of the edge,
    // which depends on if the edge is the opposite direction of the topo edge.
    Edge& sourceEdge = mrTopology.getEdge(rSourceNode.topoEdgeId);

    VertexIdType target_vertex =
        rSourceNode.oppositeDirection ?
            sourceEdge.sourceId() : sourceEdge.targetId();

    std::vector<EdgeIdType> out_edges = getOutEdges(target_vertex);
    std::vector<EdgeIdType> targets;
    targets.insert(targets.end(), out_edges.begin(), out_edges.end());

    // build map of restricted targets
    findRestrictedTargets(sourceEdge, targets, restricted_targets);

    return restricted_targets;
}

void
GraphBuilder::findRestrictedTargets(
    const Edge&                   rSourceEdge,
    const std::vector<EdgeIdType>&  rTargets,
    std::vector<EdgeIdType>&        rRestrictedTargets) const
{
    for(EdgeIdType e_id : rTargets)
    {
        // don't add self to targetId
        if(e_id == rSourceEdge.id())
        {
            continue;
        }

        Edge& e = mrTopology.getEdge(e_id);

        if(e.isRestricted(mrConfiguration))
        {
            BOOST_LOG_SEV(mLog, boost::log::trivial::info)
                        << "Graph:getRestrictedTargets(): "
                        << "Source id " << rSourceEdge.id()
                        << " has restricted target: " << e_id;
            rRestrictedTargets.push_back(e_id);
        }
    }

    addTurningRestrictedTargets(rSourceEdge, rRestrictedTargets);
}

void
GraphBuilder::addTurningRestrictedTargets(
    const Edge& rSourceEdge,
    std::vector<EdgeIdType>& rRestrictedTargets) const
{
    if(rSourceEdge.hasRestrictions() &&
        rSourceEdge.restrictions().hasTurningRestriction())
    {
        BOOST_LOG_SEV(mLog, boost::log::trivial::info)
                    << "Graph:getRestrictedTargets(): "
```

```
                    << "Source id " << rSourceEdge.id()
                    << " has TURN restricted targets. ";
        std::vector<EdgeIdType> turn_restricted_targets =
            rSourceEdge.restrictions().restrictedTargetEdges();
        rRestrictedTargets.insert(rRestrictedTargets.end(),
            turn_restricted_targets.begin(), turn_restricted_targets.end());
    }
}

bool
GraphBuilder::isTargetRestricted(
    const std::vector<EdgeIdType>& rRestrictedTargets,
    EdgeIdType targetId) const
{
    if(mUseRestrictions && rRestrictedTargets.size() > 0)
    {
        const auto& restr_it = std::find(
            rRestrictedTargets.begin(),
            rRestrictedTargets.end(),
            targetId);
        if(restr_it != rRestrictedTargets.end())
        {
            BOOST_LOG_SEV(mLog, boost::log::trivial::info)
                << "Graph:isTargetRestricted(): "
                << "Restricted target id " << targetId;
            return true;
        }
    }
    return false;
}

void
GraphBuilder::printGraphInformation(std::ostream& os) const
{
    os << "Graph: #vertices: " << nrVertices()
       << ", #edges: " << nrEdges()
       << ". LineGraph: #nodes: " << nrNodes()
       << ", #lines: " << nrLines()
       << std::endl;
}

void
GraphBuilder::printVertices(std::ostream& os) const
{
    for(auto v_it = boost::vertices(mGraph);
        v_it.first != v_it.second;
        ++v_it.first)
    {
        const VertexType& v = *v_it.first;
        VertexIdType graph_vertex_id =
            boost::get(&GraphVertex::graphVertexId, mGraph, v);
        VertexIdType topo_vertex_id  =
            boost::get(&GraphVertex::topoVertexId, mGraph, v);
        const Vertex& vertex = mrTopology.getVertex(topo_vertex_id);

        os << "   graph_vertex_id: " << graph_vertex_id
           << ", topo_vertex_id: " << topo_vertex_id
```

```
            << "\n      v: " << v
            << "  " << vertex << std::endl;
    }
}

void
GraphBuilder::printEdges(std::ostream& os) const
{
    for(auto e_it = boost::edges(mGraph);
        e_it.first != e_it.second;
        ++e_it.first)
    {
        const EdgeType& e = *(e_it.first);
        EdgeIdType graph_edge_id =
            boost::get(&GraphEdge::graphEdgeId, mGraph, e);
        EdgeIdType topo_edge_id  =
            boost::get(&GraphEdge::topoEdgeId, mGraph, e);
        const Edge& edge = mrTopology.getEdge(topo_edge_id);

        os << "   graph_edge_id: " << graph_edge_id
           << ", e_topo_id: " << topo_edge_id
           << "\n      e: " << e
           << "  " << edge << std::endl;
    }
}

void
GraphBuilder::printNodes(std::ostream& os) const
{
    for(auto n_it = boost::vertices(mLineGraph);
        n_it.first != n_it.second;
        ++n_it.first)
    {
        const NodeType& node = *(n_it.first);
        NodeIdType lg_node_id =
            boost::get(&LineGraphNode::graphEdgeId, mLineGraph, node);
        EdgeIdType topo_edge_id  =
            boost::get(&LineGraphNode::graphEdgeId, mLineGraph, node);

        os << "   lg_node_id (graph_edge_id): " << lg_node_id
           << ", topo_edge_id: " << topo_edge_id << std::endl;
    }
}

void
GraphBuilder::printLines(std::ostream& os) const
{
    for(auto line_it = boost::edges(mLineGraph);
        line_it.first != line_it.second;
        ++line_it.first)
    {
        const LineType& line = *(line_it.first);
        NodeIdType lg_source_id =
            boost::get(&LineGraphLine::lgSourceNodeId, mLineGraph, line);
        NodeIdType lg_target_id =
            boost::get(&LineGraphLine::lgTargetNodeId, mLineGraph, line);
        VertexIdType topo_via_vertex_id =
```

```
            boost::get(&LineGraphLine::topoViaVertexId, mLineGraph, line);

        os << "   lg_source_id: " << lg_source_id
           << ", lg_target_id: " << lg_target_id
           << ", topo_via_vertex_id: " << topo_via_vertex_id << std::endl;
    }
}
```

D.7.10 Cost.h

```
/* The Cost type.
 *
 * Cost.h
 *
 * @author  Jonas Bergman
 */

#ifndef GRAPH_COST_H_
#define GRAPH_COST_H_

typedef double Cost;

#endif /* GRAPH_COST_H_ */
```

D.7.11 Speed.h

```
/** The Speed type.
 * Speed.h
 *
 * @author  Jonas Bergman
 */

#ifndef GRAPH_SPEED_H_
#define GRAPH_SPEED_H_


typedef unsigned Speed;


#endif /* GRAPH_SPEED_H_ */
```

D.7.12 EdgeCost.h

```
/**  The Costs for an Edge.
 * The cost or weight can be thought of as seconds, with the time to travel
 * the edge as a base, and different obstacles as additional costs.
 *
 * #include "EdgeCost.h"
 *
 * @author  Jonas Bergman
 */
#ifndef GRAPH_EDGECOST_H_
#define GRAPH_EDGECOST_H_

// SYSTEM INCLUDES
//
#include <map>
```

```
// PROJECT INCLUDES
//

// LOCAL INCLUDES
//
#include "Cost.h"

// FORWARD REFERENCES
//
//typedef double Cost;

/** Costs for edges:
 * - speed (either set explicitly or default  from road category)
 * - barriers (should be imported from the EdgeRestriction)
 * - incline (not implemented yet)
 * - surface
 * - traffic_calming
 * - highway => bus_stop
 * - highway => crossing
 * - highway => give_way
 * - highway => mini_roundabout
 * - highway => stop
 * - highway => traffic_signals
 * - public_transport => stop_position
 * - railway => level_crossing
 */
class EdgeCost
{
public:
// TYPES
    enum CostType
    {
        TRAVEL_TIME,
        BARRIER,
        OTHER
    };

// LIFECYCLE

    /** Default constructor.
     */
    EdgeCost() = default;


    /** Copy constructor.
     *
     * @param from The value to copy to this object.
     */
    EdgeCost(const EdgeCost& from) = delete;


    /** Destructor.
     */
    ~EdgeCost() = default;


```

```
// OPERATORS

    /** Accumulate a cost of a certain type, except `travel` which can not
     * be accumulated.
     * @param   type    The type of cost
     * @param   cost    The value of the cost in seconds.
     */
    void    addCost(CostType type, Cost cost);

    /** Clear out all costs.
     */
    void    clearCosts();

// OPERATIONS
// ACCESS

    /**
     * @return  The sum of all costs
     */
    Cost    getCost() const;

    /**
     * @return  The accumulated costs of a CostType
     */
    Cost    getCost(CostType type) const;

// INQUIRY
    /** Find out if there are costs of a certain type
     * @return  True if there is such a cost
     */
    bool    hasCost(CostType type) const;

protected:
private:
    std::map<CostType, Cost> costs;
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* GRAPH_EDGECOST_H_ */
```

D.7.13 EdgeCost.cc

```
/*
 * EdgeCost.cc
 *
 * @author  Jonas Bergman
 */

#include "EdgeCost.h"  // class implemented

/////////////////////////////// PUBLIC ///////////////////////////////////////
//============================= LIFECYCLE ====================================
```

```
//============================ OPERATORS ===================================
void
EdgeCost::addCost(CostType type, Cost cost)
{
    if(hasCost(type))
    {
        if(type != EdgeCost::TRAVEL_TIME)
        {
            cost += getCost(type);
        }
        costs.erase(type);
    }
    costs.insert({type, cost});
}

void
EdgeCost::clearCosts()
{
    costs.clear();
}
//============================ OPERATIONS ==================================
//============================ ACESS      ==================================
Cost
EdgeCost::getCost() const
{
    Cost sum {0};
    for(const auto& pair : costs)
    {
        sum += pair.second;
    }
    return sum;
}

Cost
EdgeCost::getCost(EdgeCost::CostType type) const
{
    const auto& it = costs.find(type);
    if(it != costs.end())
    {
        return it->second;
    }
    return 0;
}

//============================ INQUIRY    ==================================
bool
EdgeCost::hasCost(EdgeCost::CostType type) const
{
    return costs.find(type) != costs.end();
}
/////////////////////////////// PROTECTED  ///////////////////////////////////
/////////////////////////////// PRIVATE    ///////////////////////////////////
```

D.7.14 EdgeRestriction.h

```
/**  The EdgeRestriction class contains different restrictions for edges
 * in the graph such as dimensions, access, turn restrictions.
```

```
 *
 * #include "EdgeRestriction.h"
 *
 * @author  Jonas Bergman
 */

#ifndef GRAPH_EDGERESTRICTION_H_
#define GRAPH_EDGERESTRICTION_H_

// SYSTEM INCLUDES
//
#include <limits>
#include <map>
#include <set>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//
#include "Edge.h"
#include "RestrictionsException.h"
#include "Speed.h"
#include "Vertex.h"
#include "../config/Configuration.h"
#include "../osm/OsmAccess.h"
#include "../osm/OsmBarrier.h"
#include "../osm/OsmTurningRestriction.h"
#include "../osm/OsmVehicle.h"

// FORWARD REFERENCES
//
//class OsmTurningRestriction;

/**
 * EdgeRestriction are:
 * - vehicle properties
 * - General access to an edge
 * - vehicle type specific access
 * - barriers
 * - turn restrictions
 * - disused roads
 * - no-exit roads
 */
class EdgeRestriction
{
public:
// TYPES
    /** EdgeRestrictions on Vehicles to travel an Edge.
     *  Dimensions in meters.
     *  Speed in km/h
     */
    struct VehicleProperties
    {
        static double DEFAULT_DIMENSION_MAX;
        static Speed  DEFAULT_SPEED_MAX;
        static Speed  DEFAULT_SPEED_MIN;
```

```

        double      maxHeight   {DEFAULT_DIMENSION_MAX};
        double      maxLength   {DEFAULT_DIMENSION_MAX};
        double      maxWeight   {DEFAULT_DIMENSION_MAX};
        double      maxWidth    {DEFAULT_DIMENSION_MAX};

        Speed       maxSpeed    {DEFAULT_SPEED_MAX};
        Speed       minSpeed    {DEFAULT_SPEED_MIN};

        /** Look if the vehicle properties restricts
         * vehicle with given configuration.
         * @return  True if these vehicle properties restricts access.
         */
        bool    restrictsAccess(const VehicleConfig& rVehicleConfig) const
        {
            return (maxHeight <= rVehicleConfig.height)
                || (maxLength <= rVehicleConfig.length)
                || (maxWeight <= rVehicleConfig.weight)
                || (maxWidth  <= rVehicleConfig.width)
                || (minSpeed  >= rVehicleConfig.maxspeed);
        }
    };

    /** Types of restrictions.
     */
    enum RestrictionType
    {
        VEHICLE_PROPERTIES,
        GENERAL_ACCESS,
        VEHICLE_TYPE_ACCESS,
        BARRIER,
        TURNING,
        DISUSED,
        NO_EXIT,

        NR_RESTRICTION_TYPES
    };

// LIFECYCLE

    /** Default constructor.
     */
    EdgeRestriction() = default;


    /** Copy constructor.
     *
     * @param from The value to copy to this object.
     */
    EdgeRestriction(const EdgeRestriction& from) = delete;


    /** Destructor.
     */
    ~EdgeRestriction();


```

```
// OPERATORS
// OPERATIONS

    /** Check if this Restriction restricts when the Configuration is applied.
     * @param   rConfig     Configuration
     * @throw   RestrictionsException
     */
    bool                restricts(const Configuration& rConfig) const;

    /** Set vehicle properties for the specified edge.
     * Replacing any existing properties with the new ones.
     * @param   pVehicleProperties  The properties to install for the edge.
     */
    void                setVehiclePropertyRestriction(
                            VehicleProperties* pVehicleProperties);

    /** Set access restrictions for this edge, that is restrictions for all.
     * @param   pGeneralAccess  The access to set.
     */
    void                setGeneralAccessRestriction(
                            OsmAccess*  pGeneralAccess);

    /** Set access restrictions for this edge, that is restrictions for all.
     * @param   generalAccessType  The access type to set.
     */
    void                setGeneralAccessRestriction(
                            OsmAccess::AccessType  generalAccessType);

    /** Set access restrictions for edge based on vehicle type.
     * There can be several vehicle restrictions for each edge.
     * @param   vehicleType The type of vehicle to restrict on the edge.
     * @param   pAccess     The access restriction for that vehicle type
     *                      on this edge.
     */
    void                addVehicleTypeAccessRestriction(
                            OsmVehicle::VehicleType  vehicleType,
                            OsmAccess*               pAccess);

    /** Set access restrictions for edge based on vehicle type.
     * There can be several vehicle restrictions for each edge.
     * @param   vehicleType The type of vehicle to restrict on the edge.
     * @param   accessType  The access restriction for that vehicle type
     *                      on this edge.
     */
    void                addVehicleTypeAccessRestriction(
                            OsmVehicle::VehicleType  vehicleType,
                            OsmAccess::AccessType    accessType);

    /** Set barrier restricting this edge.
     * @param   pBarrier        The barrier to set.
     */
    void                setBarrierRestriction(
                            OsmBarrier* pBarrier);

    /** Set barrier restricting this edge.
     * @param   barrierType    The barrier type to set.
     */
```

```
    void                setBarrierRestriction(
                            OsmBarrier::BarrierType barrierType);

    /** Add turning restrictions from this edge.
     * Actually just adds the restriction without checking if there already is
     * a restriction between those two edges.
     * @param   pTurningRestriction  The turning restriction to set.
     */
    void                addTurningRestriction(
                            OsmTurningRestriction* pTurningRestriction);

    /** Set disused flag on this edge.
     */
    void                setDisusedRestriction();

    /** Set no exit flag on this edge.
     */
    void                setNoExitRestriction();

    /** Flag this edge as part of a via way restriction that needs attention
     * when routing.
     */
    void                setViaWayRestriction();

// ACCESS
    /** Get which kinds of restrictions this edge has.
     * @return  A vector with all types of restrictions.
     */
    std::vector<RestrictionType>
                        restrictionTypes() const;

    /** Try to fetch the vehicle property restrictions for an Edge.
     * @return  The Vehicle properties
     * @throw   RestrictionsException if no entry exists for Edge.
     */
    const VehicleProperties&
                        vehicleProperties() const;

    /** Try to fetch the vehicle property restrictions for an Edge.
     * @return  The Vehicle properties
     * @throw   RestrictionsException if no entry exists for Edge.
     */
    VehicleProperties&  vehicleProperties();

    /** Fetch the max speed for this edge. If no explicit speed is set it
     * returns `VehicleProperties::DEFAULT_SPEED_MAX`. One can query to see if
     * if there exists an explicit limit with  `hasMaxSpeedRestriction()`
     * @return  Either the explicit speed limit or a default if not set.
     */
    Speed               maxSpeed() const;

    /** Try to fetch the general access restrictions for this edge.
     * @param   edgeId  The id of the edge.
     * @return  reference to the OsmAccess object.
     * @throw   RestrictionsException if no entry exists for Edge.
     */
    const OsmAccess&    generalAccess() const;
```

```
    /** Try to fetch the general access restrictions for this edge.
     * @param   edgeId  The id of the edge.
     * @return  reference to the OsmAccess object.
     * @throw   RestrictionsException if no entry exists for Edge.
     */
    OsmAccess&          generalAccess();

    /** Try to fetch the vehicle type specific access restrictions for this edge.
     * @param   vehicelType    The type of Vehicle to get access restriction
     * @return  reference to the OsmAccess object.
     * @throw   RestrictionsException if no entry exists for Edge.
     */
    const OsmAccess&    vehicleTypeAccess(
                            OsmVehicle::VehicleType vehicleType) const;

    /** Try to fetch the vehicle type specific access restrictions for this edge.
     * @param   vehicelType    The type of Vehicle to get access restriction
     * @return  reference to the OsmAccess object.
     * @throw   RestrictionsException if no entry exists for Edge.
     */
    OsmAccess&          vehicleTypeAccess(
                            OsmVehicle::VehicleType vehicleType);

    /** Get a list of the types of vehicles with restrictions on this edge.
     * @return  a Vector with restriction types.
     */
    std::vector<OsmVehicle::VehicleType>
                        vehicleTypesWithRestrictions() const;

    /** Fetch the barrier restricting this edge.
     * @return  reference to a OsmBarrier object.
     * @throw   RestrictionsException if no entry exists for this Edge.
     */
    const OsmBarrier&   barrier() const;

    /** Get a list of the turning restrictions from this edge.
     * @return  a Vector with turning restrictions.
     * @throw   RestrictionsException if edge has no turning restrictions.
     */
    const std::vector<OsmTurningRestriction*>&
                        turningRestrictions() const;

    /** Get a list of all edge id's to which travel from edge is not allowed.
     * @return  A vector of edgeIds to which travel is not allowed.
     */
    std::vector<EdgeIdType>
                        restrictedTargetEdges() const;

// INQUIRY

    /** Ask if an Edge has restriction of a certain type.
     * @param   restrictionType     The type of restriction
     * @return  true if there is a restriction of that type, false if not.
     */
    bool                hasRestriction(
                            RestrictionType restrictionType) const;
```

```
    /**
     * @return true if there is a VehicleProperty restriction for edge.
     */
    bool                    hasVehiclePropertyRestriction() const;

    /** Convenience method to query for max speed.
     * @return true if there is a max speed restriction for the edge.
     */
    bool                    hasMaxSpeedRestriction() const;

    /**
     * @return true if there is a General Access restriction for the edge.
     */
    bool                    hasGeneralAccessRestriction() const;

    /**
     * @return true if there are any Vehicle Type Access restrictions for the edge.
     */
    bool                    hasVehicleTypeAccessRestriction() const;

    /**
     * @return true if there are Vehicle Type Access restrictions for the edge
     *         for that specific type of vehicle.
     */
    bool                    hasVehicleTypeAccessRestriction(
                                OsmVehicle::VehicleType vehicleType) const;

    /**
     * @return true if there are any barriers restricting access to the edge.
     */
    bool                    hasBarrierRestriction() const;

    /**
     * @return true if there are any turning restrictions traveling from edge.
     */
    bool                    hasTurningRestriction() const;

    /**
     * @return true if the edge is 'disused'.
     */
    bool                    hasDisusedRestriction() const;

    /**
     * @return true if the edge has no exit.
     */
    bool                    hasNoExitRestriction() const;

    /**
     * @return true if the edge is part of a turning restriction via another way.
     */
    bool                    hasViaWayRestriction() const;


protected:
private:
    VehicleProperties*      mpVehicleProperties {nullptr};
    OsmAccess*              mpGeneralAccess {nullptr};
```

```
    std::map<OsmVehicle::VehicleType,  OsmAccess*>
                                mVehicleTypeAccessMap;
    OsmBarrier*                 mpBarrier {nullptr};
    std::vector<OsmTurningRestriction*>
                                mTurningRestrictions;
    bool                        mIsDisusedEdge {false};
    bool                        mIsNoExitEdge {false};
    bool                        mHasViaWayRestriction {false};
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* GRAPH_EDGERESTRICTION_H_ */
```

D.7.15 EdgeRestriction.cc

```
/*
 * EdgeRestriction.cc
 *
 * @author  Jonas Bergman
 */

#include "EdgeRestriction.h"  // class implemented

// STATIC INITIALIZATION
/*static*/ double EdgeRestriction::VehicleProperties::DEFAULT_DIMENSION_MAX
    = std::numeric_limits<double>::max();

/*static*/ Speed  EdgeRestriction::VehicleProperties::DEFAULT_SPEED_MAX
    = std::numeric_limits<unsigned>::max();

/*static*/ Speed  EdgeRestriction::VehicleProperties::DEFAULT_SPEED_MIN
    = 0;

/////////////////////////////// PUBLIC ///////////////////////////////////////
//============================= LIFECYCLE ====================================
EdgeRestriction::~EdgeRestriction()
{
    delete mpVehicleProperties;
    delete mpGeneralAccess;
    delete mpBarrier;
    for(auto it : mVehicleTypeAccessMap)
    {
        delete it.second;
    }
    for(auto it : mTurningRestrictions)
    {
        delete it;
    }
}
//============================= OPERATORS ====================================
//============================= OPERATIONS ===================================
bool
```

```
EdgeRestriction::restricts(const Configuration& rConfig) const
{
    bool is_restricted = false;
    bool is_generally_restricted = false;
    bool is_vehicle_banned = false;

    for(const auto& r : restrictionTypes())
    {
        switch (r)
        {
            case EdgeRestriction::DISUSED:
                is_restricted = true; break;
            case EdgeRestriction::VEHICLE_PROPERTIES:
                if(vehicleProperties()
                    .restrictsAccess(rConfig.getVehicleConfig()))
                {
                    is_restricted = true;
                }
                break;
            case EdgeRestriction::BARRIER:
                if(barrier()
                    .restrictsAccess(rConfig.getBarrierRestrictionsRule()))
                {
                    is_restricted = true;
                }
                break;
            case EdgeRestriction::GENERAL_ACCESS:
                if(!generalAccess()
                    .allowsAccess(rConfig.getAccessRule()))
                {
                    is_generally_restricted = true;
                }
                continue;
            case EdgeRestriction::VEHICLE_TYPE_ACCESS:
            {
                OsmVehicle::VehicleType type =
                    rConfig.getVehicleConfig().category;
                if(hasVehicleTypeAccessRestriction(type))
                {
                    if(!vehicleTypeAccess(type)
                        .allowsAccess(rConfig.getAccessRule()))
                    {
                        is_vehicle_banned = true;
                    }
                }
            }
            continue;
            default:
                continue;
        }
    }

    if(is_restricted
        || (is_generally_restricted && is_vehicle_banned)
        || is_vehicle_banned)
    {
        return true;
```

```
    }
    return false;
}

void
EdgeRestriction::setVehiclePropertyRestriction(
    EdgeRestriction::VehicleProperties* pVehicleProperties)
{
    delete mpVehicleProperties;
    mpVehicleProperties = pVehicleProperties;
}

void
EdgeRestriction::setGeneralAccessRestriction(
    OsmAccess*  pGeneralAccess)
{
    delete mpGeneralAccess;
    mpGeneralAccess = pGeneralAccess;
}

void
EdgeRestriction::setGeneralAccessRestriction(
    OsmAccess::AccessType  generalAccessType)
{
    delete mpGeneralAccess;
    mpGeneralAccess = new OsmAccess(generalAccessType);
}

void
EdgeRestriction::addVehicleTypeAccessRestriction(
    OsmVehicle::VehicleType  vehicleType,
    OsmAccess*               pAccess)
{
    if(hasVehicleTypeAccessRestriction(vehicleType))
    {
        auto old_access = mVehicleTypeAccessMap.find(vehicleType);
        delete old_access->second;
        mVehicleTypeAccessMap.erase(vehicleType);
    }
    mVehicleTypeAccessMap.insert({vehicleType, pAccess});
}

void
EdgeRestriction::addVehicleTypeAccessRestriction(
    OsmVehicle::VehicleType  vehicleType,
    OsmAccess::AccessType    accessType)
{
    addVehicleTypeAccessRestriction(vehicleType, new OsmAccess(accessType));
}

void
EdgeRestriction::setBarrierRestriction(
    OsmBarrier* pBarrier)
{
    delete mpBarrier;
    mpBarrier = pBarrier;
}
```

```

void
EdgeRestriction::setBarrierRestriction(
    OsmBarrier::BarrierType barrierType)
{
    delete mpBarrier;
    mpBarrier = new OsmBarrier(barrierType);
}

void
EdgeRestriction::addTurningRestriction(
    OsmTurningRestriction*   pTurningRestriction)
{
    mTurningRestrictions.push_back(pTurningRestriction);
}

void
EdgeRestriction::setDisusedRestriction()
{
    mIsDisusedEdge = true;
}

void
EdgeRestriction::setNoExitRestriction()
{
    mIsNoExitEdge = true;
}

void
EdgeRestriction::setViaWayRestriction()
{
    mHasViaWayRestriction = true;
}
//============================= ACESS      ===================================
std::vector<EdgeRestriction::RestrictionType>
EdgeRestriction::restrictionTypes() const
{
    std::vector<EdgeRestriction::RestrictionType> rest_types;

    for(int i = EdgeRestriction::VEHICLE_PROPERTIES;
        i < EdgeRestriction::NR_RESTRICTION_TYPES;
        ++i)
    {
        RestrictionType type = static_cast<RestrictionType>(i);
        if(hasRestriction(type))
        {
            rest_types.push_back(type);
        }
    }

    return rest_types;
}

const EdgeRestriction::VehicleProperties&
EdgeRestriction::vehicleProperties() const
{
    if(!hasVehiclePropertyRestriction())
```

```
    {
        throw RestrictionsException(
            "Restrictions:vehicleProperties: no restriction for edge");
    }
    return *mpVehicleProperties;
}

EdgeRestriction::VehicleProperties&
EdgeRestriction::vehicleProperties()
{
    if(!hasVehiclePropertyRestriction())
    {
        throw RestrictionsException(
            "Restrictions:vehicleProperties: no restriction for edge");
    }
    return *mpVehicleProperties;
}

Speed
EdgeRestriction::maxSpeed() const
{
    if(hasVehiclePropertyRestriction())
    {
        return mpVehicleProperties->maxSpeed;
    }
    return VehicleProperties::DEFAULT_SPEED_MAX;
}

const OsmAccess&
EdgeRestriction::generalAccess() const
{
    if(!hasGeneralAccessRestriction())
    {
        throw RestrictionsException(
            "Restrictions:generalAccess: no restriction for edge");
    }
    return *mpGeneralAccess;
}

OsmAccess&
EdgeRestriction::generalAccess()
{
    if(!hasGeneralAccessRestriction())
    {
        throw RestrictionsException(
            "Restrictions:generalAccess: no restriction for edge");
    }
    return *mpGeneralAccess;
}

const OsmAccess&
EdgeRestriction::vehicleTypeAccess(
    OsmVehicle::VehicleType vehicleType) const
{
    if(!hasVehicleTypeAccessRestriction(vehicleType))
    {
        throw RestrictionsException(
```

```
            std::string("Restrictions:vehicleTypeAccess: no restriction for")
            + " vehicle type " + OsmVehicle::toString(vehicleType)
            + " for edge ");
    }
    return *(mVehicleTypeAccessMap.find(vehicleType)->second);
}

OsmAccess&
EdgeRestriction::vehicleTypeAccess(
    OsmVehicle::VehicleType vehicleType)
{
    return const_cast<OsmAccess&>
        (static_cast<const EdgeRestriction&>
            (*this).vehicleTypeAccess(vehicleType)
        );
}

std::vector<OsmVehicle::VehicleType>
EdgeRestriction::vehicleTypesWithRestrictions() const
{
    std::vector<OsmVehicle::VehicleType> types;

    for( int i = 0; i < OsmVehicle::NR_VEHICLE_TYPES; ++i)
    {
        OsmVehicle::VehicleType type = static_cast<OsmVehicle::VehicleType>(i);

        if(hasVehicleTypeAccessRestriction(type))
        {
            types.push_back(type);
        }
    }

    return types;
}

const OsmBarrier&
EdgeRestriction::barrier() const
{
    if(!hasBarrierRestriction())
    {
        throw RestrictionsException(
            "Restrictions:barrier: no restriction for edge");
    }
    return *mpBarrier;
}

const std::vector<OsmTurningRestriction*>&
EdgeRestriction::turningRestrictions() const
{
    if(!hasTurningRestriction())
    {
        throw RestrictionsException(
            "Restriction:turningRestriction: no turning restriction for edge");
    }
    return mTurningRestrictions;
}

```

```
std::vector<EdgeIdType>
EdgeRestriction::restrictedTargetEdges() const
{
    std::vector<EdgeIdType> restricted_targets;

    try
    {
        const auto& r_vec = this->turningRestrictions();

        for(const auto& restr : r_vec)
        {
            restricted_targets.push_back(restr->toEdgeId());
        }
    }
    catch (RestrictionsException& re)
    {
        // never mind
    }
    return restricted_targets;
}

//============================ INQUIRY    ===================================
bool
EdgeRestriction::hasRestriction(
    EdgeRestriction::RestrictionType type) const
{
    switch (type)
    {
        case VEHICLE_PROPERTIES:
            return hasVehiclePropertyRestriction(); break;
        case GENERAL_ACCESS:
            return hasGeneralAccessRestriction(); break;
        case VEHICLE_TYPE_ACCESS:
            return hasVehicleTypeAccessRestriction(); break;
        case BARRIER:
            return hasBarrierRestriction(); break;
        case TURNING:
            return hasTurningRestriction(); break;
        case DISUSED:
            return hasDisusedRestriction(); break;
        case NO_EXIT:
            return hasNoExitRestriction(); break;
        default:
            return false;
    }
}

bool
EdgeRestriction::hasVehiclePropertyRestriction() const
{
    return mpVehicleProperties != nullptr;
}

bool
EdgeRestriction::hasMaxSpeedRestriction() const
{
    if(hasVehiclePropertyRestriction())
```

```
    {
        return mpVehicleProperties->maxSpeed != VehicleProperties::DEFAULT_SPEED_MAX;
    }
    return false;
}

bool
EdgeRestriction::hasGeneralAccessRestriction() const
{
    return mpGeneralAccess != nullptr;
}

bool
EdgeRestriction::hasVehicleTypeAccessRestriction() const
{
    return mVehicleTypeAccessMap.size() > 0;
}

bool
EdgeRestriction::hasVehicleTypeAccessRestriction(
    OsmVehicle::VehicleType vehicleType) const
{
    auto it = mVehicleTypeAccessMap.find(vehicleType);
    if (it != mVehicleTypeAccessMap.end())
    {
        return true;
    }
    return false;
}

bool
EdgeRestriction::hasBarrierRestriction() const
{
    return  mpBarrier != nullptr;
}

bool
EdgeRestriction::hasTurningRestriction() const
{
    return mTurningRestrictions.size() > 0;
}

bool
EdgeRestriction::hasDisusedRestriction() const
{
    return mIsDisusedEdge;
}

bool
EdgeRestriction::hasNoExitRestriction() const
{
    return mIsNoExitEdge;
}

bool
EdgeRestriction::hasViaWayRestriction() const
{
```

```
    return mHasViaWayRestriction;
}
/////////////////////////////// PROTECTED  ///////////////////////////////////

/////////////////////////////// PRIVATE    ///////////////////////////////////
```

D.7.16 TurnCostCalculator.h

```
/** Calculate the turn cost for making a turn between to edges (roads).
 *
 * #include "TurnCostCalculator.h"
 *
 * @author  Jonas Bergman
 */

#ifndef GRAPH_TURNCOSTCALCULATOR_H_
#define GRAPH_TURNCOSTCALCULATOR_H_

// SYSTEM INCLUDES
//
#include <initializer_list>
#include <cstdlib> // abs()

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//
#include "Cost.h"
#include "Edge.h"
#include "Speed.h"
#include "../config/Configuration.h"

// FORWARD REFERENCES
//

// TYPES
//

/**
 * Calculate the cost for making turns.
 * Based on "Route Planning in Road Networks with Turn Costs"
 * by Lars Volker. Universitat Karslruhe 2008.
 * http://algo2.iti.kit.edu/documents/routeplanning/volker_sa.pdf
 *
 * The cost is dependent on
 * - angle between roads
 * - size of vehicle
 * - road category precedence
 */
class TurnCostCalculator
{
public:
// LIFECYCLE
    TurnCostCalculator() = delete;
    virtual ~TurnCostCalculator() = delete;

```

```
// OPERATORS
// OPERATIONS
// ACCESS
    /**
     * @return  The cost of the turn.
     */
    static double   getTurnCost(
                        const Edge& rSource,
                        const Edge& rTarget,
                        const Configuration& rConfig);
// INQUIRY
private:
// HELPERS
    /**
     * @param   rSource     Source edge.
     * @param   rTarget     Target edge.
     * @return  The speed dependent on the angle between edges.
     */
    static Speed    getAngleSpeed(
                        const Edge& rSource,
                        const Edge& rTarget);

    /**
     * @param   rSource         Source edge.
     * @param   rTarget         Target edge.
     * @param   vehicle_length  Length of the vehicle.
     * @param   angleSpeed      The angle dependent speed
     * @return  The speed dependent on the size of the routed vehicle.
     */
    static Speed    getVehicleSizeSpeed(
                        const Edge& rSource,
                        const Edge& rTarget,
                        double      vehicle_length,
                        Speed       angleSpeed);

    /**
     * @param   speeds      A set of speeds.
     * @return  The smallest speed
     */
    static Speed    getSmallestSpeed(std::initializer_list<Speed> speeds);

    /** Get the angle between source and target as
     * -180 < angle < 180
     * That means that 0 is straight ahead, > 0 to the right
     * and < 0 to the left.
     */
    static int      getTurnAngle(
                        const Edge& rSource,
                        const Edge& rTarget);

    /** Calculate a penalty for making sharp right turns with long vehicles.
     * @param   turnAngle       The turning angle in degrees (-180 < a < 180).
     * @param   vehicleLength   The length of the vehicle.
     * @return  A factor 0.33 - 1.0
     */
    static double   calculateLengthPenaltyFactor(
                        int     turnAngle,
```

```
                         double  vehicleLength);

    /** Look if target is of a more important highway type than the source,
     * in that case we must add a penalty for giving way when entering
     * the target road.
     * @param   rSource     The source edge.
     * @param   rTarget     The target edge.
     * @return  A cost for giving way.
     */
    static Cost     giveWayToHigherRoadCategoryCost(
                        const Edge& rSource,
                        const Edge& rTarget);

// ATTRIBUTES
// CONSTANTS
    static constexpr double VEHICLE_PENALTY_LENGTH = 4.5;

};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* GRAPH_TURNCOSTCALCULATOR_H_ */
```

D.7.17 TurnCostCalculator.cc

```
/*
 * TurnCostCalculator.cc
 */

#include "TurnCostCalculator.h" // class implemented

////////////////////////////// PUBLIC ////////////////////////////////////
//============================= LIFECYCLE ====================================
//============================= OPERATORS ====================================
//============================= OPERATIONS ===================================
//============================= ACESS      ===================================
/* static */
double
TurnCostCalculator::getTurnCost(
    const Edge& rSource,
    const Edge& rTarget,
    const Configuration& rConfig)
{
    double vehicle_length = rConfig.getVehicleConfig().length;

    Speed angle_speed = getAngleSpeed(rSource, rTarget);
    Speed size_speed =
        getVehicleSizeSpeed(rSource, rTarget, vehicle_length, angle_speed);
    Speed turn_speed = getSmallestSpeed({angle_speed, size_speed});

    double decel_factor = rConfig.getVehicleConfig().acceleration / 100.0;
    double accel_factor = rConfig.getVehicleConfig().deceleration / 100.0;

```

```
    Cost deceleration_cost = decel_factor * (rSource.speed() - turn_speed);
    Cost acceleration_cost = accel_factor * (rTarget.speed() - turn_speed);
    Cost additional_cost = giveWayToHigherRoadCategoryCost(rSource, rTarget);

    Cost turn_cost = deceleration_cost + acceleration_cost + additional_cost;

    return static_cast<double>(turn_cost);
}
//============================= INQUIRY    ===================================
///////////////////////////////// PROTECTED  //////////////////////////////////

///////////////////////////////// PRIVATE    //////////////////////////////////
/* static */
Speed
TurnCostCalculator::getAngleSpeed(const Edge& rSource, const Edge& rTarget)
{
    int turn_angle = getTurnAngle(rSource, rTarget);
    // make sure there is some speed and not 0
    if(abs(turn_angle) > 175)
    {
        turn_angle = 175;
    }
    double reduction_factor(1 - (abs(turn_angle)/180.0));
    double speed =
        reduction_factor * getSmallestSpeed({rSource.speed(), rTarget.speed()});
    return static_cast<Speed>(speed);
}

/* static */
Speed
TurnCostCalculator::getVehicleSizeSpeed(
    const Edge& rSource,
    const Edge& rTarget,
    double      vehicle_length,
    Speed       angleSpeed)
{
    int turn_angle (getTurnAngle(rSource, rTarget));
    double length_penalty_factor =
        calculateLengthPenaltyFactor(turn_angle, vehicle_length);

    Speed speed = angleSpeed
                    * (VEHICLE_PENALTY_LENGTH / vehicle_length)
                    * length_penalty_factor;
    return speed;
}

/* static */
Speed
TurnCostCalculator::getSmallestSpeed(std::initializer_list<Speed> speeds)
{
    Speed min {1000};

    if(speeds.size() > 0)
    {
        for(const auto& s : speeds)
        {
            if(s < min)
```

```
                {
                    min = s;
                }
            }
        }
        return min;
}

/* static */
int
TurnCostCalculator::getTurnAngle(const Edge& rSource, const Edge& rTarget)
{
    int angle =
        rSource.geomData().targetBearing - rTarget.geomData().sourceBearing;
    if(angle < -180)
    {
        angle += 360;
    }
    if(angle > 180)
    {
        angle -= 360;
    }
    return angle;
}

/* static */
double
TurnCostCalculator::calculateLengthPenaltyFactor(
    int turnAngle,
    double vehicleLength)
{
    double factor(1.0);

    if(vehicleLength > VEHICLE_PENALTY_LENGTH)
    {
        if(turnAngle > 0)
        {
            factor = 1.0 - ((2.0/3.0) * (turnAngle/180.0));
        }
    }
    return factor;
}

/* static */
Cost
TurnCostCalculator::giveWayToHigherRoadCategoryCost(
    const Edge& rSource,
    const Edge& rTarget)
{
    if(rSource.roadData().roadType > rTarget.roadData().roadType)
    {
        return 5;
    }
    return 0;
}
```

D.7.18 GraphException.h

```
/**  Exception thrown by the Graph package.
 *
 * #include "GraphException.h"
 *
 * @author  Jonas Bergman
 */

#ifndef GRAPH_GRAPHEXCEPTION_H_
#define GRAPH_GRAPHEXCEPTION_H_

// SYSTEM INCLUDES
//
#include <exception>
#include <string>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//

// FORWARD REFERENCES
//

/**
 * Exception to throw from the 'graph' package.
 * More information of the type of exception is given in the 'what()' message.
 */
class GraphException : public std::exception
{
public:
// LIFECYCLE
    /** Default constructor.
     */
    GraphException() = delete;

    /** Constructor taking a message to display.
     *
     * @param    message    The message to prepend when 'what()' is called.
     */
    GraphException(const std::string& rMessage) noexcept
        : std::exception(), mMessage(rMessage)
    {}

// OPERATORS
// OPERATIONS
// ACCESS
// INQUIRY
    const char* what() const noexcept
    { return mMessage.c_str(); }

protected:
private:
// ATTRIBUTES
    std::string        mMessage;
};
```

```

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* GRAPH_GRAPHEXCEPTION_H_ */
```

D.7.19 RestrictionsException.h

```
/**  Exception thrown by the Restrictions.
 *
 * #include "RestrictionsException.h"
 *
 * @author  Jonas Bergman
 */

#ifndef GRAPH_RESTRICTIONSEXCEPTION_H_
#define GRAPH_RESTRICTIONSEXCEPTION_H_

// SYSTEM INCLUDES
//
#include <exception>
#include <string>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//

// FORWARD REFERENCES
//

/**
 * Exception to throw from Restrictions class.
 * More information of the type of exception is given in the 'what()' message.
 */
class RestrictionsException : public std::exception
{
public:
// LIFECYCLE
    /** Default constructor.
     */
    RestrictionsException() = delete;

    /** Constructor taking a message to display.
     *
     * @param   message     The message to prepend when 'what()' is called.
     */
    RestrictionsException(const std::string& rMessage) noexcept
        : std::exception(), mMessage(rMessage)
    {}

// OPERATORS
    void    addEdgeId(std::string edgeIdString) { mMessage += edgeIdString; }
```

```
// OPERATIONS
// ACCESS
// INQUIRY
    const char* what() const noexcept
    { return mMessage.c_str(); }

protected:
private:
// ATTRIBUTES
    std::string     mMessage;
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* GRAPH_RESTRICTIONSEXCEPTION_H_ */
```

D.7.20 TopologyException.h

```
/**  Exception thrown by the Topology package.
 *
 * #include "TopologyException.h"
 *
 * @author  Jonas Bergman
 */

#ifndef GRAPH_TOPOLOGYEXCEPTION_H_
#define GRAPH_TOPOLOGYEXCEPTION_H_

// SYSTEM INCLUDES
//
#include <exception>
#include <string>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//

// FORWARD REFERENCES
//

/**
 * Exception to throw from the 'topology' package.
 * More information of the type of exception is given in the 'what()' message.
 */
class TopologyException : public std::exception
{
public:
// LIFECYCLE
    /** Default constructor.
     */
    TopologyException() = delete;
```

```

    /** Constructor taking a message to display.
     *
     * @param    message        The message to prepend when 'what()' is called.
     */
    TopologyException(const std::string& rMessage) noexcept
        : std::exception(), mMessage(rMessage)
    {}

// OPERATORS
// OPERATIONS
// ACCESS
// INQUIRY
    const char* what() const noexcept
    { return mMessage.c_str(); }

protected:
private:
// ATTRIBUTES
    std::string        mMessage;
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* GRAPH_TOPOLOGYEXCEPTION_H_ */
```

D.7.21 EdgeCost_test.cc

```
/* Tests for EdgeCost class
 * EdgeCost_test.cc
 *
 * @author  Jonas Bergman
 */

#include "../../catchtest/catch.hpp"

#include "../EdgeCost.h"

SCENARIO ("Keeping track of costs for an Edge", "[graph][edgecost]")
{
    EdgeCost costs;

    GIVEN ("an EdgeCost object")
    {
        WHEN ("no costs are added")
        {
            THEN ("there should be no costs")
            {
                REQUIRE (costs.getCost() == 0);
                REQUIRE (costs.hasCost(EdgeCost::TRAVEL_TIME) == false);
                REQUIRE (costs.getCost(EdgeCost::BARRIER) == 0);
            }
        }
```

```

        WHEN ("travel cost is added")
        {
            costs.addCost(EdgeCost::TRAVEL_TIME, 10);

            THEN ("there should be costs")
            {
                REQUIRE (costs.getCost() > 0);
                REQUIRE (costs.hasCost(EdgeCost::TRAVEL_TIME) == true);
                REQUIRE (costs.getCost(EdgeCost::TRAVEL_TIME) == Approx(10.0));
                REQUIRE (costs.getCost(EdgeCost::BARRIER) == 0);
            }
        }

        WHEN ("two travel costs are added")
        {
            costs.addCost(EdgeCost::TRAVEL_TIME, 10);
            costs.addCost(EdgeCost::TRAVEL_TIME, 20);

            THEN ("only the last should be reported")
            {
                REQUIRE (costs.hasCost(EdgeCost::TRAVEL_TIME) == true);
                REQUIRE (costs.getCost(EdgeCost::TRAVEL_TIME) == Approx(20.0));
            }
        }

        WHEN ("travel a travel and a barrier cost are added")
        {
            costs.addCost(EdgeCost::TRAVEL_TIME, 10);
            costs.addCost(EdgeCost::BARRIER, 20);

            THEN ("the costs should be added")
            {
                REQUIRE (costs.getCost() == Approx(30.0));
                REQUIRE (costs.hasCost(EdgeCost::TRAVEL_TIME) == true);
                REQUIRE (costs.hasCost(EdgeCost::BARRIER) == true);
                REQUIRE (costs.hasCost(EdgeCost::OTHER) == false);
            }
        }
    }
}
```

D.7.22 GraphBuilder_test.cc

```
/*
 * Graph_test.cc
 *
 * @author  Jonas Bergman
 */

#include <iostream>

#include "../GraphBuilder.h"
#include "../Topology.h"
#include "../../catchtest/catch.hpp"
#include "../../config/ConfigurationReader.h"
#include "../../mapprovider/postgis/PostGisProvider.h"
```

```

SCENARIO ("Building a small graph", "[graph][basic]")
{
    // ----------------------------------------------------------------------
    GIVEN ("Three points and two edges for a topology")
    {
        size_t nr_vertices = 3;
        size_t nr_edges = 2;
        OsmIdType osm_id(1);

        Topology topology;
        const Vertex& v1 = topology.addVertex(1, Point(0,0));
        const Vertex& v2 = topology.addVertex(2, Point(1,2));
        const Vertex& v3 = topology.addVertex(3, Point(3,1));
        Edge& e1 = topology.addEdge(1,osm_id,1,2);
        Edge& e2 = topology.addEdge(2,osm_id,2,3);

        Configuration config;

        // ......................................................................
        WHEN ("we try create a Graph from the Topology")
        {
            THEN ("we should not get an Exception")
            {
                INFO ("calling Graph constructor");
                REQUIRE_NOTHROW (GraphBuilder g(topology, config));
            }
        }

        // ......................................................................
        WHEN ("building a graph from the topology")
        {
            GraphBuilder g(topology, config);
            const auto& boost_graph = g.getBoostGraph();
            LineGraphType& r_boost_line_graph = g.getBoostLineGraph();

            // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
            THEN ("the # of vertices in the graph representation"
                " should be as in the topology"
                " and the # edges the double") // default is bidirectional
            {
                REQUIRE (boost::num_vertices(boost_graph) == nr_vertices);
                REQUIRE (boost::num_edges(boost_graph) == nr_edges * 2);
            }

            // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
            THEN ("the number of nodes in the LineGraph"
                  " should be as many as edges in the graph")
            {
                REQUIRE (boost::num_vertices(r_boost_line_graph) ==
                         boost::num_edges(boost_graph));
            }
        }

        // ......................................................................
        WHEN ("we try print out a Graph from the Topology")
        {
```

```
            GraphBuilder g(topology, config);

            THEN ("we should get a print out")
            {
                INFO (g);
                REQUIRE (true);
            }
        }

        // ..................................................................
        WHEN ("adding unidirectional information to edges before"
              " building graph")
        {

            Edge::RoadData rd1;
            rd1.direction = Edge::DirectionType::FROM_TO;
            e1.setRoadData(rd1);

            Edge::RoadData rd2;
            rd2.direction = Edge::DirectionType::FROM_TO;
            e2.setRoadData(rd2);

            GraphBuilder g2(topology, config);

            THEN ("the # of edges in the graph representation"
                  " should as many as in the topology")
            {
                INFO (g2);
                const auto& boost_graph = g2.getBoostGraph();
                REQUIRE (boost::num_edges(boost_graph) == topology.nrEdges());
            }
        }

        // ..................................................................
        WHEN ("adding an extra lane to an edge before"
              " building graph")
        {

            Edge::RoadData rd1;
            rd1.direction = Edge::DirectionType::FROM_TO;
            rd1.nrLanes = 2;
            e1.setRoadData(rd1);

            Edge::RoadData rd2;
            rd2.direction = Edge::DirectionType::FROM_TO;
            e2.setRoadData(rd2);

            GraphBuilder g2(topology, config);

            THEN ("the # of edges in the graph representation"
                " should be one more than in the topology")
            {
                INFO (g2);
                const auto& boost_graph = g2.getBoostGraph();
                REQUIRE (boost::num_edges(boost_graph) == topology.nrEdges() + 1);
            }
        }
```

```
    }
}

SCENARIO ("Building graph with restrictions", "[graph][restrictions]")
{
    try
    {

        // ==================================================================
        GIVEN ("Configuration to build a Graph with restrictions ")
        {
            std::string config_file(
                "catchtest/testsettings/mikh_restr_0617-testsettings.json");
            ConfigurationReader config_reader(config_file);
            Configuration config;
            config_reader.fillConfiguration(config);

            PostGisProvider pgp(config);

            Topology topology;
            pgp.getTopology(topology);
            pgp.setRestrictionsAndCosts(topology);

            GraphBuilder graph_restr(topology, config);

            GraphBuilder graph_unrestr(topology, config, false);

            // ...............................................................
            WHEN ("Adding a turning restriction and a point restriction (barrier)")
            {

                THEN ("there should be equally many vertices "
                      "in restricted and unrestricted")
                {
                    INFO ("  Restricted # Vertices: " << graph_restr.nrVertices());
                    INFO ("UNRestricted # Vertices: " << graph_unrestr.nrVertices());
                    REQUIRE (graph_restr.nrVertices() == graph_unrestr.nrVertices());
                }

                THEN ("there should be 2 less edges "
                      "in restricted and unrestricted")
                {
                    INFO ("  Restricted # Edges:    " << graph_restr.nrEdges());
                    INFO ("UNRestricted # Edges:    " << graph_unrestr.nrEdges());
                    REQUIRE (graph_restr.nrEdges() == graph_unrestr.nrEdges() - 2);
                }

                THEN ("there should be 2 less nodes "
                      "in restricted and unrestricted")
                {
                    INFO ("  Restricted # Nodes:    " << graph_restr.nrNodes());
                    INFO ("UNRestricted # Nodes:    " << graph_unrestr.nrNodes());
                    REQUIRE (graph_restr.nrNodes() == graph_unrestr.nrNodes() - 2);
                }
                THEN ("there should be 9 lines less "
                      "in restricted than unrestricted")
                {
```

```
                    // 1 turn restriction
                    // 3*2 where target is restricted by barrier (lift gate)
                    // 2 u-turns on restricted edge
                    INFO ("  Restricted # Lines:    " << graph_restr.nrLines());
                    INFO ("UNRestricted # Lines:    " << graph_unrestr.nrLines());
                    REQUIRE (graph_restr.nrLines() == graph_unrestr.nrLines() - 9);
                }
                THEN ("we can print the info for an edge and it should have a cost")
                {
                    EdgeIdType id = 270;
                    const Edge& edge = topology.getEdge(id);
                    INFO ("Edge " << id << ": " << edge);
                    REQUIRE (true);
                }
            }
        }
    }
    catch (ConfigurationException& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (MapProviderException& dbe)
    {
        INFO(dbe.what());
        REQUIRE (false);    // force output of error and failure

    }
}
```

D.7.23 RestrictionsAndCosts_test.cc

```
/* Tests for the different kind of restrictions
 *
 * Graph_test.cc
 * @author  Jonas Bergman
 */

#include <iostream>

#include "../../catchtest/catch.hpp"

#include "../Topology.h"
#include "../../config/ConfigurationReader.h"
#include "../../mapprovider/postgis/PostGisProvider.h"
#include "../GraphBuilder.h"

// TURN RESTRICTION ////////////////////////////////////////////////////////////

SCENARIO ("Building graph of Mikhailovsk with turn restriction",
    "[graph][r_and_c][turn][mikhailovsk]")
{
    // block on node 1706164751 on way 158421713
    try
    {
        std::string orig_config_file("catchtest/testsettings/"
            "restrictions/mikhailovsk-original.json");
```

```
        ConfigurationReader orig_config_reader(orig_config_file);
        Configuration orig_config;
        orig_config_reader.fillConfiguration(orig_config);
        PostGisProvider orig_pgp(orig_config);
        Topology orig_topology;
        orig_pgp.getTopology(orig_topology);
        orig_pgp.setRestrictionsAndCosts(orig_topology);
        GraphBuilder orig_graph(orig_topology, orig_config);

        // ===================================================================
        GIVEN ("Configuration to build a Graph with turn restriction ")
        {
            std::string config_file("catchtest/testsettings/"
                "restrictions/mikhailovsk-turn_no_right.json");
            ConfigurationReader config_reader(config_file);
            Configuration config;
            config_reader.fillConfiguration(config);

            PostGisProvider pgp(config);

            Topology topology;
            pgp.getTopology(topology);
            pgp.setRestrictionsAndCosts(topology);

            GraphBuilder graph(topology, config);

            // ...............................................................
            WHEN ("Comparing original to graph with turn restrictions")
            {

                THEN ("there should be equally many vertices "
                      "in original and restricted")
                {
                    INFO ("  Original # Vertices: " << orig_graph.nrVertices());
                    INFO ("Restricted # Vertices: " << graph.nrVertices());
                    REQUIRE (orig_graph.nrVertices() == graph.nrVertices());
                }

                THEN ("there should be equally many edges "
                      "in original as in restricted")
                {
                    INFO ("  Original # Edges: " << orig_graph.nrEdges());
                    INFO ("Restricted # Edges: " << graph.nrEdges());
                    REQUIRE ((orig_graph.nrEdges()) == graph.nrEdges());
                }

                THEN ("there should be equally many nodes "
                      "in original as in restricted")
                {
                    INFO ("  Original # Nodes:    " << orig_graph.nrNodes());
                    INFO ("Restricted # Nodes:    " << graph.nrNodes());
                    REQUIRE ((orig_graph.nrNodes()) == graph.nrNodes());
                }

                THEN ("there should be 1 less line "
                      "in original than in restricted")
                {
```

```
                    // 1 right turn
                    INFO ("  Original # Lines:    " << orig_graph.nrLines());
                    INFO ("Restricted # Lines:    " << graph.nrLines());
                    REQUIRE ((orig_graph.nrLines() - 1) == graph.nrLines());
                }
            }
        }
    }
    catch (ConfigurationException& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (MapProviderException& dbe)
    {
        INFO(dbe.what());
        REQUIRE (false);    // force output of error and failure

    }
}

SCENARIO ("Building graph of Partille with turn restriction",
    "[graph][r_and_c][turn][partille]")
{
    // block on node 1706164751 on way 158421713
    try
    {
        std::string orig_config_file("catchtest/testsettings/"
            "restrictions/partille-original.json");
        ConfigurationReader orig_config_reader(orig_config_file);
        Configuration orig_config;
        orig_config_reader.fillConfiguration(orig_config);
        PostGisProvider orig_pgp(orig_config);
        Topology orig_topology;
        orig_pgp.getTopology(orig_topology);
        orig_pgp.setRestrictionsAndCosts(orig_topology);
        GraphBuilder orig_graph(orig_topology, orig_config);

        // ==============================================================
        GIVEN ("Configuration to build a Graph with turn restriction ")
        {
            std::string config_file("catchtest/testsettings/"
                "restrictions/partille-turn_no_left.json");
            ConfigurationReader config_reader(config_file);
            Configuration config;
            config_reader.fillConfiguration(config);

            PostGisProvider pgp(config);

            Topology topology;
            pgp.getTopology(topology);
            pgp.setRestrictionsAndCosts(topology);

            GraphBuilder graph(topology, config);

            // ..............................................................
            WHEN ("Comparing original to graph with turn restrictions")
```

```
            {

                THEN ("there should be equally many vertices "
                      "in original and restricted")
                {
                    INFO ("  Original # Vertices: " << orig_graph.nrVertices());
                    INFO ("Restricted # Vertices: " << graph.nrVertices());
                    REQUIRE (orig_graph.nrVertices() == graph.nrVertices());
                }

                THEN ("there should be equally many edges "
                      "in original as in restricted")
                {
                    INFO ("  Original # Edges: " << orig_graph.nrEdges());
                    INFO ("Restricted # Edges: " << graph.nrEdges());
                    REQUIRE ((orig_graph.nrEdges()) == graph.nrEdges());
                }

                THEN ("there should be equally many nodes "
                      "in original as in restricted")
                {
                    INFO ("  Original # Nodes:    " << orig_graph.nrNodes());
                    INFO ("Restricted # Nodes:    " << graph.nrNodes());
                    REQUIRE ((orig_graph.nrNodes()) == graph.nrNodes());
                }

                THEN ("there should be 1 less line "
                      "in original than in restricted")
                {
                    // 1 right turn
                    INFO ("  Original # Lines:    " << orig_graph.nrLines());
                    INFO ("Restricted # Lines:    " << graph.nrLines());
                    REQUIRE ((orig_graph.nrLines() - 1) == graph.nrLines());
                }
            }
        }
    }
    catch (ConfigurationException& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (MapProviderException& dbe)
    {
        INFO(dbe.what());
        REQUIRE (false);    // force output of error and failure

    }
}

// BARRIER BLOCK ///////////////////////////////////////////////////////////

SCENARIO ("Building graph of Mikhailovsk with barrier block",
    "[graph][r_and_c][block][mikhailovsk]")
{
    // block on node 1706164751 on way 158421713
    try
```

```
    {
        std::string orig_config_file("catchtest/testsettings/"
            "restrictions/mikhailovsk-original.json");
        ConfigurationReader orig_config_reader(orig_config_file);
        Configuration orig_config;
        orig_config_reader.fillConfiguration(orig_config);
        PostGisProvider orig_pgp(orig_config);
        Topology orig_topology;
        orig_pgp.getTopology(orig_topology);
        orig_pgp.setRestrictionsAndCosts(orig_topology);
        GraphBuilder orig_graph(orig_topology, orig_config);

        // ===================================================================
        GIVEN ("Configuration that restricts barrier block ")
        {
            std::string config_file("catchtest/testsettings/"
                "restrictions/mikhailovsk-barrier_block.json");
            ConfigurationReader config_reader(config_file);
            Configuration config;
            config_reader.fillConfiguration(config);

            PostGisProvider pgp(config);

            Topology topology;
            pgp.getTopology(topology);
            pgp.setRestrictionsAndCosts(topology);

            GraphBuilder graph(topology, config);

            // ...............................................................
            WHEN ("Comparing original to graph with barrier block")
            {

                THEN ("there should be equally many vertices "
                      "in original and restricted")
                {
                    INFO ("  Original # Vertices: " << orig_graph.nrVertices());
                    INFO ("Restricted # Vertices: " << graph.nrVertices());
                    REQUIRE (orig_graph.nrVertices() == graph.nrVertices());
                }

                THEN ("there should be 2 more edges "
                      "in original than in restricted")
                {
                    INFO ("  Original # Edges: " << orig_graph.nrEdges());
                    INFO ("Restricted # Edges: " << graph.nrEdges());
                    REQUIRE ((orig_graph.nrEdges() - 2) == graph.nrEdges());
                }

                THEN ("there should be 2 more nodes "
                      "in original than in restricted")
                {
                    INFO ("  Original # Nodes:    " << orig_graph.nrNodes());
                    INFO ("Restricted # Nodes:    " << graph.nrNodes());
                    REQUIRE ((orig_graph.nrNodes() - 2) == graph.nrNodes());
                }

```

```
                THEN ("there should be 10 more lines "
                      "in original than in restricted")
                {
                    // 4 bidirectional edges connecting = 8 lines
                    // 2 u-turns = 2 lines
                    INFO ("  Original # Lines:    " << orig_graph.nrLines());
                    INFO ("Restricted # Lines:    " << graph.nrLines());
                    REQUIRE ((orig_graph.nrLines() - 10) == graph.nrLines());
                }
            }
        }
    }
    catch (ConfigurationException& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (MapProviderException& dbe)
    {
        INFO(dbe.what());
        REQUIRE (false);    // force output of error and failure

    }
}

SCENARIO ("Building graph of Partille with barrier block",
    "[graph][r_and_c][block][partille]")
{
    // block on node 249292683 on way 28050664
    try
    {
        std::string orig_config_file("catchtest/testsettings/"
            "restrictions/partille-original.json");
        ConfigurationReader orig_config_reader(orig_config_file);
        Configuration orig_config;
        orig_config_reader.fillConfiguration(orig_config);
        PostGisProvider orig_pgp(orig_config);
        Topology orig_topology;
        orig_pgp.getTopology(orig_topology);
        orig_pgp.setRestrictionsAndCosts(orig_topology);
        GraphBuilder orig_graph(orig_topology, orig_config);

        // ===================================================================
        GIVEN ("Configuration that restricts barrier block ")
        {
            std::string config_file("catchtest/testsettings/"
                "restrictions/partille-barrier_block.json");
            ConfigurationReader config_reader(config_file);
            Configuration config;
            config_reader.fillConfiguration(config);

            PostGisProvider pgp(config);

            Topology topology;
            pgp.getTopology(topology);
            pgp.setRestrictionsAndCosts(topology);

```

```
            GraphBuilder graph(topology, config);

            // ................................................................
            WHEN ("Comparing original to graph with barrier block")
            {

                THEN ("there should be equally many vertices "
                      "in original and restricted")
                {
                    INFO ("  Original # Vertices: " << orig_graph.nrVertices());
                    INFO ("Restricted # Vertices: " << graph.nrVertices());
                    REQUIRE (orig_graph.nrVertices() == graph.nrVertices());
                }

                THEN ("there should be 2 more edges "
                      "in original than in restricted")
                {
                    INFO ("  Original # Edges: " << orig_graph.nrEdges());
                    INFO ("Restricted # Edges: " << graph.nrEdges());
                    REQUIRE ((orig_graph.nrEdges() - 2) == graph.nrEdges());
                }

                THEN ("there should be 2 more nodes "
                      "in original than in restricted")
                {
                    INFO ("  Original # Nodes:    " << orig_graph.nrNodes());
                    INFO ("Restricted # Nodes:    " << graph.nrNodes());
                    REQUIRE ((orig_graph.nrNodes() - 2) == graph.nrNodes());
                }

                THEN ("there should be 8 more lines "
                      "in original than in restricted")
                {
                    // 3 bidirectional edges connecting = 6 lines
                    // (1 cycleway = 0 lines)
                    // 2 u-turns = 2 lines
                    INFO ("  Original # Lines:    " << orig_graph.nrLines());
                    INFO ("Restricted # Lines:    " << graph.nrLines());
                    REQUIRE ((orig_graph.nrLines() - 8) == graph.nrLines());
                }
            }
        }
    }
    catch (ConfigurationException& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (MapProviderException& dbe)
    {
        INFO(dbe.what());
        REQUIRE (false);    // force output of error and failure

    }
}
```

```
// BARRIER BOLLARD /////////////////////////////////////////////////////////////

SCENARIO ("Building graph of Mikhailovsk with barrier bollard",
    "[graph][r_and_c][bollard][mikhailovsk]")
{
    // block on node 1706164751 on way 158421713
    try
    {
        std::string orig_config_file("catchtest/testsettings/"
            "restrictions/mikhailovsk-original.json");
        ConfigurationReader orig_config_reader(orig_config_file);
        Configuration orig_config;
        orig_config_reader.fillConfiguration(orig_config);
        PostGisProvider orig_pgp(orig_config);
        Topology orig_topology;
        orig_pgp.getTopology(orig_topology);
        orig_pgp.setRestrictionsAndCosts(orig_topology);
        GraphBuilder orig_graph(orig_topology, orig_config);

        // =================================================================
        GIVEN ("Configuration that restricts barrier bollard ")
        {
            std::string config_file("catchtest/testsettings/"
                "restrictions/mikhailovsk-barrier_bollard.json");
            ConfigurationReader config_reader(config_file);
            Configuration config;
            config_reader.fillConfiguration(config);

            PostGisProvider pgp(config);

            Topology topology;
            pgp.getTopology(topology);
            pgp.setRestrictionsAndCosts(topology);

            GraphBuilder graph(topology, config);

            // .............................................................
            WHEN ("Comparing original to graph with barrier bollard")
            {

                THEN ("there should be equally many vertices "
                      "in original and restricted")
                {
                    INFO ("  Original # Vertices: " << orig_graph.nrVertices());
                    INFO ("Restricted # Vertices: " << graph.nrVertices());
                    REQUIRE (orig_graph.nrVertices() == graph.nrVertices());
                }

                THEN ("there should be 2 more edges "
                      "in original than in restricted")
                {
                    INFO ("  Original # Edges: " << orig_graph.nrEdges());
                    INFO ("Restricted # Edges: " << graph.nrEdges());
                    REQUIRE ((orig_graph.nrEdges() - 2) == graph.nrEdges());
                }

                THEN ("there should be 2 more nodes "
```

```
                        "in original than in restricted")
                {
                    INFO ("  Original # Nodes:    " << orig_graph.nrNodes());
                    INFO ("Restricted # Nodes:    " << graph.nrNodes());
                    REQUIRE ((orig_graph.nrNodes() - 2) == graph.nrNodes());
                }

                THEN ("there should be 10 more lines "
                        "in original than in restricted")
                {
                    // 4 bidirectional edges connecting = 8 lines
                    // 2 u-turns = 2 lines
                    INFO ("  Original # Lines:    " << orig_graph.nrLines());
                    INFO ("Restricted # Lines:    " << graph.nrLines());
                    REQUIRE ((orig_graph.nrLines() - 10) == graph.nrLines());
                }
            }
        }
    }
    catch (ConfigurationException& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (MapProviderException& dbe)
    {
        INFO(dbe.what());
        REQUIRE (false);    // force output of error and failure

    }
}

SCENARIO ("Building graph of Partille with barrier bollard",
    "[graph][r_and_c][bollard][partille]")
{
    // block on node 249292683 on way 28050664
    try
    {
        std::string orig_config_file("catchtest/testsettings/"
            "restrictions/partille-original.json");
        ConfigurationReader orig_config_reader(orig_config_file);
        Configuration orig_config;
        orig_config_reader.fillConfiguration(orig_config);
        PostGisProvider orig_pgp(orig_config);
        Topology orig_topology;
        orig_pgp.getTopology(orig_topology);
        orig_pgp.setRestrictionsAndCosts(orig_topology);
        GraphBuilder orig_graph(orig_topology, orig_config);

        // ===================================================================
        GIVEN ("Configuration that restricts barrier bollard ")
        {
            std::string config_file("catchtest/testsettings/"
                "restrictions/partille-barrier_bollard.json");
            ConfigurationReader config_reader(config_file);
            Configuration config;
            config_reader.fillConfiguration(config);
```

```
            PostGisProvider pgp(config);

            Topology topology;
            pgp.getTopology(topology);
            pgp.setRestrictionsAndCosts(topology);

            GraphBuilder graph(topology, config);

            // .............................................................
            WHEN ("Comparing original to graph with barrier bollard")
            {

                THEN ("there should be equally many vertices "
                      "in original and restricted")
                {
                    INFO ("  Original # Vertices: " << orig_graph.nrVertices());
                    INFO ("Restricted # Vertices: " << graph.nrVertices());
                    REQUIRE (orig_graph.nrVertices() == graph.nrVertices());
                }

                THEN ("there should be 2 more edges "
                      "in original than in restricted")
                {
                    INFO ("  Original # Edges: " << orig_graph.nrEdges());
                    INFO ("Restricted # Edges: " << graph.nrEdges());
                    REQUIRE ((orig_graph.nrEdges() - 2) == graph.nrEdges());
                }

                THEN ("there should be 2 more nodes "
                      "in original than in restricted")
                {
                    INFO ("  Original # Nodes:    " << orig_graph.nrNodes());
                    INFO ("Restricted # Nodes:    " << graph.nrNodes());
                    REQUIRE ((orig_graph.nrNodes() - 2) == graph.nrNodes());
                }

                THEN ("there should be 8 more lines "
                      "in original than in restricted")
                {
                    // 3 bidirectional edges connecting = 6 lines
                    // (1 cycleway = 0 lines)
                    // 2 u-turns = 2 lines
                    INFO ("  Original # Lines:    " << orig_graph.nrLines());
                    INFO ("Restricted # Lines:    " << graph.nrLines());
                    REQUIRE ((orig_graph.nrLines() - 8) == graph.nrLines());
                }
            }
        }
    }
    catch (ConfigurationException& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (MapProviderException& dbe)
    {
```

```
        INFO(dbe.what());
        REQUIRE (false);    // force output of error and failure

    }
}

// BARRIER LIFT GATE /////////////////////////////////////////////////////

SCENARIO ("Building graph of Mikhailovsk with barrier lift gate",
    "[graph][r_and_c][lift_gate][mikhailovsk]")
{
    // block on node 1706164751 on way 158421713 (topo edge 649)
    try
    {
        std::string orig_config_file("catchtest/testsettings/"
            "restrictions/mikhailovsk-original.json");
        ConfigurationReader orig_config_reader(orig_config_file);
        Configuration orig_config;
        orig_config_reader.fillConfiguration(orig_config);
        PostGisProvider orig_pgp(orig_config);
        Topology orig_topology;
        orig_pgp.getTopology(orig_topology);
        orig_pgp.setRestrictionsAndCosts(orig_topology);
        GraphBuilder orig_graph(orig_topology, orig_config);

        // ==================================================================
        GIVEN ("Configuration that does not restrict barrier lift gate ")
        {
            std::string config_file("catchtest/testsettings/"
                "restrictions/mikhailovsk-barrier_lift_gate.json");
            ConfigurationReader config_reader(config_file);
            Configuration config;
            config_reader.fillConfiguration(config);

            PostGisProvider pgp(config);

            Topology topology;
            pgp.getTopology(topology);
            pgp.setRestrictionsAndCosts(topology);

            GraphBuilder graph(topology, config);

            // ..............................................................
            WHEN ("Comparing original to graph with barrier lift gate")
            {

                THEN ("there should be equally many vertices "
                      "in original and restricted")
                {
                    INFO ("  Original # Vertices: " << orig_graph.nrVertices());
                    INFO ("Restricted # Vertices: " << graph.nrVertices());
                    REQUIRE (orig_graph.nrVertices() == graph.nrVertices());
                }

                THEN ("there should be equally many edges "
                      "in original as in restricted")
                {
```

```
                    INFO ("  Original # Edges: " << orig_graph.nrEdges());
                    INFO ("Restricted # Edges: " << graph.nrEdges());
                    REQUIRE ((orig_graph.nrEdges()) == graph.nrEdges());
                }

                THEN ("there should be equally many nodes "
                      "in original as in restricted")
                {
                    INFO ("  Original # Nodes:    " << orig_graph.nrNodes());
                    INFO ("Restricted # Nodes:    " << graph.nrNodes());
                    REQUIRE ((orig_graph.nrNodes()) == graph.nrNodes());
                }

                THEN ("there should be equally many lines "
                      "in original as in restricted")
                {
                    INFO ("  Original # Lines:    " << orig_graph.nrLines());
                    INFO ("Restricted # Lines:    " << graph.nrLines());
                    REQUIRE ((orig_graph.nrLines()) == graph.nrLines());
                }

                THEN ("there should be an extra cost of 60 on edge 649")
                {
                    EdgeIdType id = 649;
                    Cost orig_cost = orig_topology.getEdge(id).cost();
                    Cost rest_cost = topology.getEdge(id).cost();
                    INFO ("  Original cost:    " << orig_cost);
                    INFO ("Restricted cost:    " << rest_cost);
                    REQUIRE (60.0 == Approx((rest_cost - orig_cost)));
                }
            }
        }
    }
    catch (ConfigurationException& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (MapProviderException& dbe)
    {
        INFO(dbe.what());
        REQUIRE (false);    // force output of error and failure

    }
}


SCENARIO ("Building graph of Partille with barrier lift gate",
    "[graph][r_and_c][lift_gate][partille]")
{
    // lift gate on node 249292683 on way 28050664 (topo edge 267)
    try
    {
        std::string orig_config_file("catchtest/testsettings/"
            "restrictions/partille-original.json");
        ConfigurationReader orig_config_reader(orig_config_file);
        Configuration orig_config;
```

```
        orig_config_reader.fillConfiguration(orig_config);
        PostGisProvider orig_pgp(orig_config);
        Topology orig_topology;
        orig_pgp.getTopology(orig_topology);
        orig_pgp.setRestrictionsAndCosts(orig_topology);
        GraphBuilder orig_graph(orig_topology, orig_config);

        // ===================================================================
        GIVEN ("Configuration that does not restrict barrier lift gate ")
        {
            std::string config_file("catchtest/testsettings/"
                "restrictions/partille-barrier_lift_gate.json");
            ConfigurationReader config_reader(config_file);
            Configuration config;
            config_reader.fillConfiguration(config);

            PostGisProvider pgp(config);

            Topology topology;
            pgp.getTopology(topology);
            pgp.setRestrictionsAndCosts(topology);

            GraphBuilder graph(topology, config);

            // ...............................................................
            WHEN ("Comparing original to graph with barrier lift gate")
            {

                THEN ("there should be equally many vertices "
                      "in original and restricted")
                {
                    INFO ("  Original # Vertices: " << orig_graph.nrVertices());
                    INFO ("Restricted # Vertices: " << graph.nrVertices());
                    REQUIRE (orig_graph.nrVertices() == graph.nrVertices());
                }

                THEN ("there should be equally many edges "
                      "in original as in restricted")
                {
                    INFO ("  Original # Edges: " << orig_graph.nrEdges());
                    INFO ("Restricted # Edges: " << graph.nrEdges());
                    REQUIRE ((orig_graph.nrEdges()) == graph.nrEdges());
                }

                THEN ("there should be equally many nodes "
                      "in original as in restricted")
                {
                    INFO ("  Original # Nodes:    " << orig_graph.nrNodes());
                    INFO ("Restricted # Nodes:    " << graph.nrNodes());
                    REQUIRE ((orig_graph.nrNodes()) == graph.nrNodes());
                }

                THEN ("there should be equally many lines "
                      "in original as in restricted")
                {
                    INFO ("  Original # Lines:    " << orig_graph.nrLines());
                    INFO ("Restricted # Lines:    " << graph.nrLines());
```

```
                        REQUIRE ((orig_graph.nrLines()) == graph.nrLines());
                    }

                    THEN ("there should be an extra cost of 60 on edge 267")
                    {
                        EdgeIdType id = 267;
                        Cost orig_cost = orig_topology.getEdge(id).cost();
                        Cost rest_cost = topology.getEdge(id).cost();
                        INFO ("  Original cost:    " << orig_cost);
                        INFO ("Restricted cost:    " << rest_cost);
                        REQUIRE (60.0 == Approx((rest_cost - orig_cost));
                    }
                }
            }
        }
        catch (ConfigurationException& e)
        {
            INFO(e.what());
            REQUIRE (false);    // force output of error and failure
        }
        catch (MapProviderException& dbe)
        {
            INFO(dbe.what());
            REQUIRE (false);    // force output of error and failure

        }
}

// TRAFFIC LIGHTS ///////////////////////////////////////////////////////////////

SCENARIO ("Building graph of Mikhailovsk with traffic signals",
    "[graph][r_and_c][traffic_signals][mikhailovsk]")
{
    // additional node on way 158421713 (topo edge id 649)
    try
    {
        std::string orig_config_file("catchtest/testsettings/"
            "restrictions/mikhailovsk-original.json");
        ConfigurationReader orig_config_reader(orig_config_file);
        Configuration orig_config;
        orig_config_reader.fillConfiguration(orig_config);
        PostGisProvider orig_pgp(orig_config);
        Topology orig_topology;
        orig_pgp.getTopology(orig_topology);
        orig_pgp.setRestrictionsAndCosts(orig_topology);
        GraphBuilder orig_graph(orig_topology, orig_config);

        // =================================================================
        GIVEN ("Configuration that has cost for traffic signals ")
        {
            std::string config_file("catchtest/testsettings/"
                "restrictions/mikhailovsk-highway_traffic_signals.json");
            ConfigurationReader config_reader(config_file);
            Configuration config;
            config_reader.fillConfiguration(config);

            PostGisProvider pgp(config);
```

```

            Topology topology;
            pgp.getTopology(topology);
            pgp.setRestrictionsAndCosts(topology);

            GraphBuilder graph(topology, config);

            // .............................................................
            WHEN ("Comparing original to graph with traffic lights")
            {

                THEN ("there should be equally many vertices "
                      "in original and restricted")
                {
                    INFO ("  Original # Vertices: " << orig_graph.nrVertices());
                    INFO ("Restricted # Vertices: " << graph.nrVertices());
                    REQUIRE (orig_graph.nrVertices() == graph.nrVertices());
                }

                THEN ("there should be equally many edges "
                      "in original as in restricted")
                {
                    INFO ("  Original # Edges: " << orig_graph.nrEdges());
                    INFO ("Restricted # Edges: " << graph.nrEdges());
                    REQUIRE ((orig_graph.nrEdges()) == graph.nrEdges());
                }

                THEN ("there should be equally many nodes "
                      "in original as in restricted")
                {
                    INFO ("  Original # Nodes:    " << orig_graph.nrNodes());
                    INFO ("Restricted # Nodes:    " << graph.nrNodes());
                    REQUIRE ((orig_graph.nrNodes()) == graph.nrNodes());
                }

                THEN ("there should be equally many lines "
                      "in original as in restricted")
                {
                    INFO ("  Original # Lines:    " << orig_graph.nrLines());
                    INFO ("Restricted # Lines:    " << graph.nrLines());
                    REQUIRE ((orig_graph.nrLines()) == graph.nrLines());
                }

                THEN ("there should be an extra cost of 30 on edge 649")
                {
                    EdgeIdType id = 649;
                    Cost orig_cost = orig_topology.getEdge(id).cost();
                    Cost rest_cost = topology.getEdge(id).cost();
                    INFO ("  Original cost:    " << orig_cost);
                    INFO ("Restricted cost:    " << rest_cost);
                    REQUIRE (30.0 == Approx((rest_cost - orig_cost)));
                }
            }
        }
    }
    catch (ConfigurationException& e)
    {
```

```
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (MapProviderException& dbe)
    {
        INFO(dbe.what());
        REQUIRE (false);    // force output of error and failure

    }
}

SCENARIO ("Building graph of Partille with traffic signals",
    "[graph][r_and_c][traffic_signals][partille]")
{
    // additional node on way 28050664 (topo edge id 267)
    try
    {
        std::string orig_config_file("catchtest/testsettings/"
            "restrictions/partille-original.json");
        ConfigurationReader orig_config_reader(orig_config_file);
        Configuration orig_config;
        orig_config_reader.fillConfiguration(orig_config);
        PostGisProvider orig_pgp(orig_config);
        Topology orig_topology;
        orig_pgp.getTopology(orig_topology);
        orig_pgp.setRestrictionsAndCosts(orig_topology);
        GraphBuilder orig_graph(orig_topology, orig_config);

        // ================================================================
        GIVEN ("Configuration that has cost for traffic signals ")
        {
            std::string config_file("catchtest/testsettings/"
                "restrictions/partille-highway_traffic_signals.json");
            ConfigurationReader config_reader(config_file);
            Configuration config;
            config_reader.fillConfiguration(config);

            PostGisProvider pgp(config);

            Topology topology;
            pgp.getTopology(topology);
            pgp.setRestrictionsAndCosts(topology);

            GraphBuilder graph(topology, config);

            // ..............................................................
            WHEN ("Comparing original to graph with traffic lights")
            {

                THEN ("there should be equally many vertices "
                      "in original and restricted")
                {
                    INFO ("  Original # Vertices: " << orig_graph.nrVertices());
                    INFO ("Restricted # Vertices: " << graph.nrVertices());
                    REQUIRE (orig_graph.nrVertices() == graph.nrVertices());
                }

```

```
                THEN ("there should be equally many edges "
                      "in original as in restricted")
                {
                    INFO ("  Original # Edges: " << orig_graph.nrEdges());
                    INFO ("Restricted # Edges: " << graph.nrEdges());
                    REQUIRE ((orig_graph.nrEdges()) == graph.nrEdges());
                }

                THEN ("there should be equally many nodes "
                      "in original as in restricted")
                {
                    INFO ("  Original # Nodes:    " << orig_graph.nrNodes());
                    INFO ("Restricted # Nodes:    " << graph.nrNodes());
                    REQUIRE ((orig_graph.nrNodes()) == graph.nrNodes());
                }

                THEN ("there should be equally many lines "
                      "in original as in restricted")
                {
                    INFO ("  Original # Lines:    " << orig_graph.nrLines());
                    INFO ("Restricted # Lines:    " << graph.nrLines());
                    REQUIRE ((orig_graph.nrLines()) == graph.nrLines());
                }

                THEN ("there should be an extra cost of 30 on edge 267")
                {
                    EdgeIdType id = 267;
                    Cost orig_cost = orig_topology.getEdge(id).cost();
                    Cost rest_cost = topology.getEdge(id).cost();
                    INFO ("  Original cost:    " << orig_cost);
                    INFO ("Restricted cost:    " << rest_cost);
                    REQUIRE (30.0 == Approx((rest_cost - orig_cost)));
                }
            }
        }
    }
    catch (ConfigurationException& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (MapProviderException& dbe)
    {
        INFO(dbe.what());
        REQUIRE (false);    // force output of error and failure

    }
}

// STOP NODE //////////////////////////////////////////////////////////////

SCENARIO ("Building graph of Mikhailovsk with stop at node before crossing",
    "[graph][r_and_c][stop_node][mikhailovsk]")
{
    // additional node on way 158421713 (topo edge id 649)
    try
    {
```

```
        std::string orig_config_file("catchtest/testsettings/"
            "restrictions/mikhailovsk-original.json");
        ConfigurationReader orig_config_reader(orig_config_file);
        Configuration orig_config;
        orig_config_reader.fillConfiguration(orig_config);
        PostGisProvider orig_pgp(orig_config);
        Topology orig_topology;
        orig_pgp.getTopology(orig_topology);
        orig_pgp.setRestrictionsAndCosts(orig_topology);
        GraphBuilder orig_graph(orig_topology, orig_config);

        // ===================================================================
        GIVEN ("Configuration that has cost for stops ")
        {
            std::string config_file("catchtest/testsettings/"
                "restrictions/mikhailovsk-highway_stop_node.json");
            ConfigurationReader config_reader(config_file);
            Configuration config;
            config_reader.fillConfiguration(config);

            PostGisProvider pgp(config);

            Topology topology;
            pgp.getTopology(topology);
            pgp.setRestrictionsAndCosts(topology);

            GraphBuilder graph(topology, config);

            // ...............................................................
            WHEN ("Comparing original to graph with stop signs")
            {
                THEN ("there should be equally many lines "
                      "in original as in restricted")
                {
                    INFO ("  Original # Lines:    " << orig_graph.nrLines());
                    INFO ("Restricted # Lines:    " << graph.nrLines());
                    REQUIRE ((orig_graph.nrLines()) == graph.nrLines());
                }

                THEN ("there should be an extra cost of 30 on edge 649")
                {
                    EdgeIdType id = 649;
                    Cost orig_cost = orig_topology.getEdge(id).cost();
                    Cost rest_cost = topology.getEdge(id).cost();
                    INFO ("  Original cost:    " << orig_cost);
                    INFO ("Restricted cost:    " << rest_cost);
                    REQUIRE (30.0 == Approx((rest_cost - orig_cost)));
                }
            }
        }
    }
    catch (ConfigurationException& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (MapProviderException& dbe)
```

```
    {
        INFO(dbe.what());
        REQUIRE (false);    // force output of error and failure

    }
}

SCENARIO ("Building graph of Partille with stop at node before crossing",
    "[graph][r_and_c][stop_node][partille]")
{
    // additional node on way 28050664 (topo edge id 267)
    try
    {
        std::string orig_config_file("catchtest/testsettings/"
            "restrictions/partille-original.json");
        ConfigurationReader orig_config_reader(orig_config_file);
        Configuration orig_config;
        orig_config_reader.fillConfiguration(orig_config);
        PostGisProvider orig_pgp(orig_config);
        Topology orig_topology;
        orig_pgp.getTopology(orig_topology);
        orig_pgp.setRestrictionsAndCosts(orig_topology);
        GraphBuilder orig_graph(orig_topology, orig_config);

        // =================================================================
        GIVEN ("Configuration that has cost for traffic signals ")
        {
            std::string config_file("catchtest/testsettings/"
                "restrictions/partille-highway_stop_node.json");
            ConfigurationReader config_reader(config_file);
            Configuration config;
            config_reader.fillConfiguration(config);

            PostGisProvider pgp(config);

            Topology topology;
            pgp.getTopology(topology);
            pgp.setRestrictionsAndCosts(topology);

            GraphBuilder graph(topology, config);

            // .............................................................
            WHEN ("Comparing original to graph with stop signs")
            {
                THEN ("there should be equally many lines "
                      "in original as in restricted")
                {
                    INFO ("  Original # Lines:    " << orig_graph.nrLines());
                    INFO ("Restricted # Lines:    " << graph.nrLines());
                    REQUIRE ((orig_graph.nrLines()) == graph.nrLines());
                }

                THEN ("there should be an extra cost of 30 on edge 267")
                {
                    EdgeIdType id = 267;
                    Cost orig_cost = orig_topology.getEdge(id).cost();
                    Cost rest_cost = topology.getEdge(id).cost();
```

```
                    INFO ("  Original cost:    " << orig_cost);
                    INFO ("Restricted cost:    " << rest_cost);
                    REQUIRE (30.0 == Approx((rest_cost - orig_cost)));
                }
            }
        }
    }
    catch (ConfigurationException& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (MapProviderException& dbe)
    {
        INFO(dbe.what());
        REQUIRE (false);    // force output of error and failure

    }
}

// STOP ALL ////////////////////////////////////////////////////////////////

SCENARIO ("Building graph of Mikhailovsk with stop for all at crossing",
    "[graph][r_and_c][stop_all][mikhailovsk]")
{
    // stop at vertex 1706164758 (topo 460)
    // affecting edges with topo id 611, 649, 661
    try
    {
        std::string orig_config_file("catchtest/testsettings/"
            "restrictions/mikhailovsk-original.json");
        ConfigurationReader orig_config_reader(orig_config_file);
        Configuration orig_config;
        orig_config_reader.fillConfiguration(orig_config);
        PostGisProvider orig_pgp(orig_config);
        Topology orig_topology;
        orig_pgp.getTopology(orig_topology);
        orig_pgp.setRestrictionsAndCosts(orig_topology);
        GraphBuilder orig_graph(orig_topology, orig_config);

        // ================================================================
        GIVEN ("Configuration that has cost for stops ")
        {
            std::string config_file("catchtest/testsettings/"
                "restrictions/mikhailovsk-highway_stop_all.json");
            ConfigurationReader config_reader(config_file);
            Configuration config;
            config_reader.fillConfiguration(config);

            PostGisProvider pgp(config);

            Topology topology;
            pgp.getTopology(topology);
            pgp.setRestrictionsAndCosts(topology);

            GraphBuilder graph(topology, config);

```

```
            // ...............................................................
            WHEN ("Comparing original to graph with stop signs")
            {
                THEN ("there should be equally many lines "
                      "in original as in restricted")
                {
                    INFO ("  Original # Lines:    " << orig_graph.nrLines());
                    INFO ("Restricted # Lines:    " << graph.nrLines());
                    REQUIRE ((orig_graph.nrLines()) == graph.nrLines());
                }

                THEN ("there should be an extra cost of 30 on edge 611")
                {
                    EdgeIdType id = 611;
                    Cost orig_cost = orig_topology.getEdge(id).cost();
                    Cost rest_cost = topology.getEdge(id).cost();
                    INFO ("  Original cost:    " << orig_cost);
                    INFO ("Restricted cost:    " << rest_cost);
                    REQUIRE (30.0 == Approx((rest_cost - orig_cost)));
                }

                THEN ("there should be an extra cost of 30 on edge 649")
                {
                    EdgeIdType id = 649;
                    Cost orig_cost = orig_topology.getEdge(id).cost();
                    Cost rest_cost = topology.getEdge(id).cost();
                    INFO ("  Original cost:    " << orig_cost);
                    INFO ("Restricted cost:    " << rest_cost);
                    REQUIRE (30.0 == Approx((rest_cost - orig_cost)));
                }

                THEN ("there should be an extra cost of 30 on edge 661")
                {
                    EdgeIdType id = 661;
                    Cost orig_cost = orig_topology.getEdge(id).cost();
                    Cost rest_cost = topology.getEdge(id).cost();
                    INFO ("  Original cost:    " << orig_cost);
                    INFO ("Restricted cost:    " << rest_cost);
                    REQUIRE (30.0 == Approx((rest_cost - orig_cost)));
                }
            }
        }
    }
    catch (ConfigurationException& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (MapProviderException& dbe)
    {
        INFO(dbe.what());
        REQUIRE (false);    // force output of error and failure

    }
}

SCENARIO ("Building graph of Partille with stop for all at crossing",
```

```
    "[graph][r_and_c][stop_all][partille]")
{
    // stop at vertex 308018343 (topo 229)
    // affecting edges with topo id 265, 266, 267
    try
    {
        std::string orig_config_file("catchtest/testsettings/"
            "restrictions/partille-original.json");
        ConfigurationReader orig_config_reader(orig_config_file);
        Configuration orig_config;
        orig_config_reader.fillConfiguration(orig_config);
        PostGisProvider orig_pgp(orig_config);
        Topology orig_topology;
        orig_pgp.getTopology(orig_topology);
        orig_pgp.setRestrictionsAndCosts(orig_topology);
        GraphBuilder orig_graph(orig_topology, orig_config);

        // ===================================================================
        GIVEN ("Configuration that has cost for traffic signals ")
        {
            std::string config_file("catchtest/testsettings/"
                "restrictions/partille-highway_stop_all.json");
            ConfigurationReader config_reader(config_file);
            Configuration config;
            config_reader.fillConfiguration(config);

            PostGisProvider pgp(config);

            Topology topology;
            pgp.getTopology(topology);
            pgp.setRestrictionsAndCosts(topology);

            GraphBuilder graph(topology, config);

            // ...............................................................
            WHEN ("Comparing original to graph with stop signs")
            {
                THEN ("there should be equally many lines "
                      "in original as in restricted")
                {
                    INFO ("  Original # Lines:    " << orig_graph.nrLines());
                    INFO ("Restricted # Lines:    " << graph.nrLines());
                    REQUIRE ((orig_graph.nrLines()) == graph.nrLines());
                }

                THEN ("there should be an extra cost of 30 on edge 265")
                {
                    EdgeIdType id = 265;
                    Cost orig_cost = orig_topology.getEdge(id).cost();
                    Cost rest_cost = topology.getEdge(id).cost();
                    INFO ("  Original cost:    " << orig_cost);
                    INFO ("Restricted cost:    " << rest_cost);
                    REQUIRE (30.0 == Approx((rest_cost - orig_cost)));
                }

                THEN ("there should be an extra cost of 30 on edge 266")
                {
```

```
                    EdgeIdType id = 266;
                    Cost orig_cost = orig_topology.getEdge(id).cost();
                    Cost rest_cost = topology.getEdge(id).cost();
                    INFO ("  Original cost:    " << orig_cost);
                    INFO ("Restricted cost:    " << rest_cost);
                    REQUIRE (30.0 == Approx((rest_cost - orig_cost)));
                }

                THEN ("there should be an extra cost of 30 on edge 267")
                {
                    EdgeIdType id = 267;
                    Cost orig_cost = orig_topology.getEdge(id).cost();
                    Cost rest_cost = topology.getEdge(id).cost();
                    INFO ("  Original cost:    " << orig_cost);
                    INFO ("Restricted cost:    " << rest_cost);
                    REQUIRE (30.0 == Approx((rest_cost - orig_cost)));
                }
            }
        }
    }
    catch (ConfigurationException& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (MapProviderException& dbe)
    {
        INFO(dbe.what());
        REQUIRE (false);    // force output of error and failure

    }
}

// TRAFFIC CALMING /////////////////////////////////////////////////////////////

SCENARIO ("Building graph of Mikhailovsk with speed bump at node",
    "[graph][r_and_c][traffic_calming][mikhailovsk]")
{
    // additional node on way 158421713 (topo edge id 649)
    try
    {
        std::string orig_config_file("catchtest/testsettings/"
            "restrictions/mikhailovsk-original.json");
        ConfigurationReader orig_config_reader(orig_config_file);
        Configuration orig_config;
        orig_config_reader.fillConfiguration(orig_config);
        PostGisProvider orig_pgp(orig_config);
        Topology orig_topology;
        orig_pgp.getTopology(orig_topology);
        orig_pgp.setRestrictionsAndCosts(orig_topology);
        GraphBuilder orig_graph(orig_topology, orig_config);

        // ===============================================================
        GIVEN ("Configuration that has cost for stops ")
        {
            std::string config_file("catchtest/testsettings/"
                "restrictions/mikhailovsk-traffic_calming_bump.json");
```

```
            ConfigurationReader config_reader(config_file);
            Configuration config;
            config_reader.fillConfiguration(config);

            PostGisProvider pgp(config);

            Topology topology;
            pgp.getTopology(topology);
            pgp.setRestrictionsAndCosts(topology);

            GraphBuilder graph(topology, config);

            // ..............................................................
            WHEN ("Comparing original to graph with stop signs")
            {
                THEN ("there should be equally many lines "
                      "in original as in restricted")
                {
                    INFO ("  Original # Lines:    " << orig_graph.nrLines());
                    INFO ("Restricted # Lines:    " << graph.nrLines());
                    REQUIRE ((orig_graph.nrLines()) == graph.nrLines());
                }

                THEN ("there should be an extra cost of 10 on edge 649")
                {
                    EdgeIdType id = 649;
                    Cost orig_cost = orig_topology.getEdge(id).cost();
                    Cost rest_cost = topology.getEdge(id).cost();
                    INFO ("  Original cost:    " << orig_cost);
                    INFO ("Restricted cost:    " << rest_cost);
                    REQUIRE (10.0 == Approx((rest_cost - orig_cost)));
                }
            }
        }
    }
    catch (ConfigurationException& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (MapProviderException& dbe)
    {
        INFO(dbe.what());
        REQUIRE (false);    // force output of error and failure

    }
}

SCENARIO ("Building graph of Partille with speed bump at node",
    "[graph][r_and_c][traffic_calming][partille]")
{
    // additional node on way 28050664 (topo edge id 267)
    try
    {
        std::string orig_config_file("catchtest/testsettings/"
            "restrictions/partille-original.json");
        ConfigurationReader orig_config_reader(orig_config_file);
```

```
        Configuration orig_config;
        orig_config_reader.fillConfiguration(orig_config);
        PostGisProvider orig_pgp(orig_config);
        Topology orig_topology;
        orig_pgp.getTopology(orig_topology);
        orig_pgp.setRestrictionsAndCosts(orig_topology);
        GraphBuilder orig_graph(orig_topology, orig_config);

        // ==============================================================
        GIVEN ("Configuration that has cost for stops ")
        {
            std::string config_file("catchtest/testsettings/"
                "restrictions/partille-traffic_calming_hump.json");
            ConfigurationReader config_reader(config_file);
            Configuration config;
            config_reader.fillConfiguration(config);

            PostGisProvider pgp(config);

            Topology topology;
            pgp.getTopology(topology);
            pgp.setRestrictionsAndCosts(topology);

            GraphBuilder graph(topology, config);

            // ..............................................................
            WHEN ("Comparing original to graph with stop signs")
            {
                THEN ("there should be equally many lines "
                      "in original as in restricted")
                {
                    INFO ("  Original # Lines:    " << orig_graph.nrLines());
                    INFO ("Restricted # Lines:    " << graph.nrLines());
                    REQUIRE ((orig_graph.nrLines()) == graph.nrLines());
                }

                THEN ("there should be an extra cost of 10 on edge 267")
                {
                    EdgeIdType id = 267;
                    Cost orig_cost = orig_topology.getEdge(id).cost();
                    Cost rest_cost = topology.getEdge(id).cost();
                    INFO ("  Original cost:    " << orig_cost);
                    INFO ("Restricted cost:    " << rest_cost);
                    REQUIRE (10.0 == Approx((rest_cost - orig_cost)));
                }
            }
        }
    }
    catch (ConfigurationException& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (MapProviderException& dbe)
    {
        INFO(dbe.what());
        REQUIRE (false);    // force output of error and failure
```

```
    }
}
```

D.7.24 Topology_test.cc

```
/*
 * Topology_test.cc
 *
 * @author  Jonas Bergman
 */

#include "../../catchtest/catch.hpp"

#include "../Topology.h"

SCENARIO ("Storing topology edges and vertices in Topology", "[topology]")
{
    OsmIdType osm_id(std::numeric_limits<OsmIdType>::max());
    // ----------------------------------------------------------------------
    GIVEN ("a Topology object and data for a vertex")
        {
        Topology topo;
        const VertexIdType id(1);
        const double x(2);
        const double y(3);
        const Point point(x, y);

        WHEN ("we try to add vertex to Topology")
        {
            const Vertex& r_vertex = topo.addVertex(id, point);

            THEN ("we should get a reference to a TopologyVertex object")
            {
                REQUIRE (r_vertex.id() == id);
                REQUIRE (r_vertex.point() == point);
                REQUIRE (r_vertex.point().x == Approx(x));
                REQUIRE (r_vertex.point().y == Approx(y));
            }
        }
        }

    // ----------------------------------------------------------------------
    GIVEN ("a Topology object and data for 2 vertices with same id")
    {
        Topology topo;
        const VertexIdType v1(1);
        const Point p1(2,3);
        const Point p2(4,5);

        WHEN ("we try to add second vertex to Topology")
        {
            const Vertex& r_v1 = topo.addVertex(v1, p1);
            const Vertex& r_v2 = topo.addVertex(v1, p2);

            THEN ("we should get a reference to first TopologyVertex object")
            {
                REQUIRE (r_v2 == r_v1);
```

```
            }
        }
    }

    // ---------------------------------------------------------------------
    GIVEN ("a Topology object and data for two vertices and an edge")
    {
        Topology topo;

        const VertexIdType v1(1);
        const Point p1(2,3);

        const VertexIdType v2(2);
        const Point p2(4,5);

        const EdgeIdType e1(1);

        WHEN ("we try to add edge to Topology with existing vertices")
        {
            const Vertex& r_v1 = topo.addVertex(v1, p1);
            const Vertex& r_v2 = topo.addVertex(v2, p2);

            const Edge& r_edge = topo.addEdge(e1, osm_id, v1, v2);

            THEN ("we should get a reference to a TopologyEdge object")
            {
                REQUIRE (r_edge.id() == e1);
                REQUIRE (r_edge.sourceId() == r_v1.id());
                REQUIRE (r_edge.targetId() == r_v2.id());
            }
        }
    }

    // ---------------------------------------------------------------------
    GIVEN ("a Topology object and data for two vertices and an edge")
    {
        Topology topo;

        const VertexIdType v1(1);
        const Point p1(2,3);

        const VertexIdType v2(2);
        const Point p2(4,5);

        const EdgeIdType e1(1);

        WHEN ("we try to add edge to Topology with non-existing vertices")
        {
            THEN ("we should get a TopologyException")
                  {
                REQUIRE_THROWS_AS(
                    const Edge& r_edge = topo.addEdge(e1, osm_id, v1, v2),
                    TopologyException&
                );
                  }
        }
    }
```

```

    // ---------------------------------------------------------------------------
    GIVEN ("Three points and to edges for a topology")
    {
        Topology topology;
        topology.addVertex(1, Point(0,0));
        topology.addVertex(2, Point(1,2));
        topology.addVertex(3, Point(3,4));
        topology.addEdge(1,osm_id,1,2);
        topology.addEdge(2,osm_id,2,3);

        // ..........................................................................
        WHEN ("counting nr of edges")
        {
            size_t nr_edges = topology.nrEdges();

            THEN ("we should get 2")
                {
                REQUIRE (nr_edges == 2);
                }
        }

        // ..........................................................................
        WHEN ("counting nr of vertices")
        {
            size_t nr_vertices = topology.nrVertices();

            THEN ("we should get 3")
            {
                REQUIRE (nr_vertices == 3);
            }
        }
    }
}
```

D.7.25 TurnCostCalculator_test.cc

```
/* Tests for TurnCostCalclulator class.
 *
 * To run these tests one needs to comment the
 * `private` label in the TurnCostCalculator.
 *
 * TurnCostCalculator_test.cc
 *
 *  @author  Jonas Bergman
 */

#include "../../catchtest/catch.hpp"

#include "../TurnCostCalculator.h"
#include "../Edge.h"
#include "../../config/ConfigurationReader.h"
#include "../../config/Configuration.h"

SCENARIO ("Keeping track of costs for Turn", "[turncost]")
{
    Edge source(1,1,1,1);
```

```
    Edge target(2,2,2,2);

    Edge::GeomData source_geom;
    Edge::GeomData target_geom;

    Edge::RoadData primary;
    Edge::RoadData secondary;

    primary.roadType = OsmHighway::HighwayType::PRIMARY;
    secondary.roadType = OsmHighway::HighwayType::SECONDARY;


    std::string config_file(
        "catchtest/testsettings/mikh_restr_0617-testsettings.json");
    ConfigurationReader config_reader(config_file);
    Configuration config;
    config_reader.fillConfiguration(config);

    GIVEN ("two edges and a configuration")
    {
        WHEN ("asking for turn cost for turn between bearing 80 and 350")
        {
            source_geom.targetBearing = 80;
            source.setGeomData(source_geom);
            source.setSpeed(90);

            target_geom.sourceBearing = 350;
            target.setGeomData(target_geom);
            target.setSpeed(60);

            THEN ("we should get a cost")
            {
                double cost =
                    TurnCostCalculator::getTurnCost(
                        source, target, config);
                INFO ("Turn cost " << cost);
                REQUIRE (cost > 0);
            }
        }

        /*********************************************************************
         * Must comment the `private` part of the TurnCostCalculator class
         * to run the following tests.
         ********************************************************************/
//        WHEN ("getting cost for lower priority source turning into"
//            " higher priority target")
//        {
//            source.setRoadData(secondary);
//            target.setRoadData(primary);
//
//            THEN ("we should get a cost")
//            {
//                double cost =
//                    TurnCostCalculator::giveWayToHigherRoadCategoryCost(
//                    source, target);
//                REQUIRE (cost > 0);
//            }
```

```
//        }
//
//        WHEN ("getting cost for higher priority source turning into"
//            " lower priority target")
//        {
//            source.setRoadData(primary);
//            target.setRoadData(secondary);
//
//            THEN ("we should not get a cost")
//            {
//                double cost =
//                    TurnCostCalculator::giveWayToHigherRoadCategoryCost(
//                    source, target);
//                REQUIRE (cost == Approx(0.0));
//            }
//        }
//
//        WHEN ("getting cost for source turning into"
//            " equal priority target")
//        {
//            source.setRoadData(primary);
//            target.setRoadData(primary);
//
//            THEN ("we should not get a cost")
//            {
//                double cost =
//                    TurnCostCalculator::giveWayToHigherRoadCategoryCost(
//                    source, target);
//                REQUIRE (cost == Approx(0.0));
//            }
//        }
//
//        WHEN ("asking for angle between bearing 80 and bearing 20")
//        {
//            source_geom.targetBearing = 80;
//            source.setGeomData(source_geom);
//
//            target_geom.sourceBearing = 20;
//            target.setGeomData(target_geom);
//
//            THEN ("we should get an angle of 60")
//            {
//                int angle = TurnCostCalculator::getTurnAngle(source, target);
//                INFO ("turn angle in 80, out 20 = " << angle);
//                REQUIRE (angle == 60);
//            }
//        }
//
//        WHEN ("asking for angle between bearing 80 and bearing 350")
//        {
//            source_geom.targetBearing = 80;
//            source.setGeomData(source_geom);
//
//            target_geom.sourceBearing = 350;
//            target.setGeomData(target_geom);
//
//            THEN ("we should get an angle of 90")
```

```
//            {
//                int angle = TurnCostCalculator::getTurnAngle(source, target);
//                INFO ("turn angle in 80, out 350 = " << angle);
//                REQUIRE (angle == 90);
//            }
//        }
//
//        WHEN ("asking for angle between bearing 80 and bearing 125")
//        {
//            source_geom.targetBearing = 80;
//            source.setGeomData(source_geom);
//
//            target_geom.sourceBearing = 125;
//            target.setGeomData(target_geom);
//
//            THEN ("we should get an angle of -45")
//            {
//                int angle = TurnCostCalculator::getTurnAngle(source, target);
//                INFO ("turn angle in 80, out 125 = " << angle);
//                REQUIRE (angle == -45);
//            }
//        }
//
//        WHEN ("asking for angle between bearing 80 and bearing 260")
//        {
//            source_geom.targetBearing = 80;
//            source.setGeomData(source_geom);
//
//            target_geom.sourceBearing = 260;
//            target.setGeomData(target_geom);
//
//            THEN ("we should get an angle of -180")
//            {
//                int angle = TurnCostCalculator::getTurnAngle(source, target);
//                INFO ("turn angle in 80, out 260 = " << angle);
//                REQUIRE (angle == -180);
//            }
//        }
//
//        WHEN ("asking for length penalty factor for length 4.5 at angle 35 degrees")
//        {
//            int angle = 35;
//
//            THEN ("we should get a factor of 1")
//            {
//                double len = 4.5;
//                double factor =
//                    TurnCostCalculator::calculateLengthPenaltyFactor(angle, len);
//                REQUIRE (factor == Approx(1.0));
//            }
//        }
//
//        WHEN ("asking for length penalty factor for length 6.0 at angle 35 degrees")
//        {
//            int angle = 35;
//
//            THEN ("we should get a factor less than 1")
```

```
//            {
//                double len = 6.0;
//                double factor =
//                    TurnCostCalculator::calculateLengthPenaltyFactor(angle, len);
//                REQUIRE (factor < 1.0);
//            }
//        }
//
//        WHEN ("asking for length penalty factor for length 6.0 at angle -130 degrees")
//        {
//            int angle = -130;
//
//            THEN ("we should get a factor equal to 1")
//            {
//                double len = 6.0;
//                double factor =
//                    TurnCostCalculator::calculateLengthPenaltyFactor(angle, len);
//                REQUIRE (factor == Approx(1.0));
//            }
//        }
//
//        WHEN ("asking for smallest speed of 20, 40, 60, 80")
//        {
//            THEN ("we should get 20")
//            {
//                Speed smallest = TurnCostCalculator::getSmallestSpeed({80,40,20,60});
//                REQUIRE (smallest == 20);
//            }
//        }
//
//        WHEN ("asking for angle speed between bearing 80 and 350")
//        {
//            source_geom.targetBearing = 80;
//            source.setGeomData(source_geom);
//            source.setSpeed(90);
//
//            target_geom.sourceBearing = 350;
//            target.setGeomData(target_geom);
//            target.setSpeed(60);
//
//            THEN ("we should get a speed")
//            {
//                Speed speed = TurnCostCalculator::getAngleSpeed(source, target);
//                INFO ("Angle speed " << speed);
//                REQUIRE ((speed > 0 && speed < 60));
//            }
//        }
//
//        WHEN ("asking for angle speed between bearing 80 and 260")
//        {
//            source_geom.targetBearing = 80;
//            source.setGeomData(source_geom);
//            source.setSpeed(90);
//
//            target_geom.sourceBearing = 260;
//            target.setGeomData(target_geom);
//            target.setSpeed(60);
```

```
//
//            THEN ("we should get a speed")
//            {
//                Speed speed = TurnCostCalculator::getAngleSpeed(source, target);
//                INFO ("Angle speed " << speed);
//                REQUIRE ((speed > 0 && speed < 60));
//            }
//        }
//
//        WHEN ("asking for angle speed between bearing 80 and 20")
//        {
//            source_geom.targetBearing = 80;
//            source.setGeomData(source_geom);
//            source.setSpeed(90);
//
//            target_geom.sourceBearing = 20;
//            target.setGeomData(target_geom);
//            target.setSpeed(60);
//
//            THEN ("we should get a speed")
//            {
//                Speed speed = TurnCostCalculator::getAngleSpeed(source, target);
//                INFO ("Angle speed " << speed);
//                REQUIRE ((speed > 0 && speed < 60));
//            }
//        }
//
//        WHEN ("asking for vehicle size speed between bearing 80 and 20, vehicle_length 4.5")
//        {
//            source_geom.targetBearing = 80;
//            source.setGeomData(source_geom);
//            source.setSpeed(90);
//
//            target_geom.sourceBearing = 20;
//            target.setGeomData(target_geom);
//            target.setSpeed(60);
//
//            double length {4.5};
//            int angle_speed {40};
//
//
//            THEN ("we should not get a speed reduction from angle speed")
//            {
//                Speed speed =
//                    TurnCostCalculator::getVehicleSizeSpeed(
//                        source, target, length, angle_speed);
//                INFO ("Vehicle size speed " << speed);
//                REQUIRE (speed == angle_speed);
//            }
//        }
//
//        WHEN ("asking for vehicle size speed between bearing 80 and 20, vehicle_length 8.5")
//        {
//            source_geom.targetBearing = 80;
//            source.setGeomData(source_geom);
//            source.setSpeed(90);
//
```

```
//            target_geom.sourceBearing = 20;
//            target.setGeomData(target_geom);
//            target.setSpeed(60);
//
//            double length {8.5};
//            int angle_speed {40};
//
//
//            THEN ("we should get a speed reduction from angle speed")
//            {
//                Speed speed =
//                    TurnCostCalculator::getVehicleSizeSpeed(
//                        source, target, length, angle_speed);
//                INFO ("Vehicle size speed " << speed);
//                REQUIRE (speed < angle_speed);
//            }
//        }
//
//        WHEN ("asking for turn cost for turn between bearing 80 and 20")
//        {
//            source_geom.targetBearing = 80;
//            source.setGeomData(source_geom);
//            source.setSpeed(90);
//
//            target_geom.sourceBearing = 20;
//            target.setGeomData(target_geom);
//            target.setSpeed(60);
//
//            THEN ("we should get a cost")
//            {
//                double cost =
//                    TurnCostCalculator::getTurnCost(
//                        source, target, config);
//                INFO ("Turn cost " << cost);
//                REQUIRE (cost > 0);
//            }
//        }
    }
}
```

D.8 lgu

D.8.1 README.md

```
Line Graph Utility
==================

The main class in this utility.

Given a configuration file it picks a `MapProvider` and fetches a `Topology`,
 ↪ which is passed to the `GraphBuilder` along with the `Configuration`. The
 ↪ goal is to fetch a linegraph that is built according to the data found in the
 ↪ database and the configuration settings found in the configuration file.
```

```
A requirement for this utility was to be able to update data in the database which means this utility can also be requested to re-read the topology if there has been a change to them, or the restrictions and costs if there has been a change to them.
```

D.8.2 LineGraphUtility.h

```
/**  The class to call to request a linegraph for routing.
 *
 * #include "LineGraphUtility.h"
 *
 * @author  Jonas Bergman
 */

#ifndef LGU_LINEGRAPHUTILITY_H_
#define LGU_LINEGRAPHUTILITY_H_

// SYSTEM INCLUDES
//

// PROJECT INCLUDES
//
#include <boost/graph/copy.hpp>

// LOCAL INCLUDES
//
#include "LineGraphUtilityException.h"
#include "../config/ConfigurationReader.h"
#include "../graph/GraphBuilder.h"
#include "../mapprovider/MapProvider.h"

// FORWARD REFERENCES
//


/** A class to run the fetching of data from database, through to complete
 * weighted linegraph.
 */
class LineGraphUtility
{
public:
// LIFECYCLE

    /** Default constructor.
     */
    LineGraphUtility() = delete;

    /** Constructor.
     * @param   rFilename   The path to the configuration file.
     */
    LineGraphUtility(const std::string& rFilename);

    /** Copy constructor.
     *
     * @param from The value to copy to this object.
     */
    LineGraphUtility(const LineGraphUtility& from) = delete;
```

```cpp


    /** Destructor.
     */
    ~LineGraphUtility(void);


// OPERATORS
// OPERATIONS
    /** Return a LineGraph
     */
    LineGraphType*   getLineGraph();

    /** Re-read the topology if there has been a change in the database.
     */
    void             updateTopology();

    /** Re-apply restrictions and costs on the topology fi there has been changes.
     */
    void             updateRestrictionsAndCosts();

    /** Save the LineGraph to storage.
     * This is a hack to be able to demo the line graph in PostGis and JOSM.
     */
    void             persistLineGraph();

    /** Output information about # vertices, edges, nodes, lines.
     */
    void             printGraphInformation(
                        std::string propmt,
                        std::ostream& os) const;

// ACCESS
// INQUIRY

protected:
private:
// HELPERS
    void    init();
    void    initConfiguration();
    void    initMapProvider();
    void    initTopology();
    void    initRestrictionsAndCosts();
    void    buildGraph();

// ATTRIBUTES
    const std::string&  mrSettingsfile;
    Configuration       mConfig;
    MapProvider*        mpMapProvider;
    Topology            mTopology;
    GraphBuilder*       mpGraphBuilder;

// CONSTANTS
};

// INLINE METHODS
//
```

```
// EXTERNAL REFERENCES
//

#endif /* LGU_LINEGRAPHUTILITY_H_ */
```

D.8.3 LineGraphUtility.cc

```
/*
 * LineGraphUtility.cc
 *
 * @author  Jonas Bergman
 */

#include "../mapprovider/postgis/PostGisProvider.h"
#include "../mapprovider/jsontest/JsonTestProvider.h"

#include <string>
#include "LineGraphUtility.h"  // class implemented

/////////////////////////////// PUBLIC ///////////////////////////////////////

//============================= LIFECYCLE ====================================
LineGraphUtility::LineGraphUtility(const std::string& rFilename)
    : mrSettingsfile(rFilename),
      mConfig(),
      mpMapProvider(nullptr),
      mTopology(),
      mpGraphBuilder(nullptr)
{
    try
    {
        init();
    }
    catch (const std::exception& e)
    {
        throw LineGraphUtilityException(
            std::string("Error initializing LineGraphUtility: ") + e.what());
    }
}

LineGraphUtility::~LineGraphUtility()
{
    delete mpMapProvider;
    delete mpGraphBuilder;
}

//============================= OPERATORS ====================================
//============================= OPERATIONS ===================================
LineGraphType*
LineGraphUtility::getLineGraph()
{
    LineGraphType& r_orig = mpGraphBuilder->getBoostLineGraph();
    LineGraphType* p_new = new LineGraphType();

    // make a copy of the old graph into a new
    boost::copy_graph(r_orig, *p_new);
```

```cpp

    return p_new;
}

void
LineGraphUtility::updateTopology()
{
    mTopology.clearTopology();
    initTopology();
    initRestrictionsAndCosts();
    buildGraph();
}

void
LineGraphUtility::updateRestrictionsAndCosts()
{
    mTopology.clearEdgeCostAndRestrictions();
    initRestrictionsAndCosts();
    buildGraph();
}

void
LineGraphUtility::persistLineGraph()
{
    try
    {
        mpMapProvider->persistLineGraph(*mpGraphBuilder);
    }
    catch(MapProviderException& mpe)
    {
        throw LineGraphUtilityException(mpe.what());
    }
}

void
LineGraphUtility::printGraphInformation(
    std::string prompt,
    std::ostream& os) const
{
    os << prompt;
    mpGraphBuilder->printGraphInformation(os);
}
//============================= ACESS      ===================================
//============================= INQUIRY    ===================================
/////////////////////////////// PROTECTED  ///////////////////////////////////

/////////////////////////////// PRIVATE    ///////////////////////////////////
void
LineGraphUtility::init()
{
    initConfiguration();
    initMapProvider();
    initTopology();
    initRestrictionsAndCosts();
    buildGraph();
}

```

```
void
LineGraphUtility::initConfiguration()
{
    try
    {
        ConfigurationReader config_reader(mrSettingsfile);
        config_reader.fillConfiguration(mConfig);
    }
    catch (ConfigurationException& ce)
    {
        delete mpMapProvider;
        delete mpGraphBuilder;
        throw LineGraphUtilityException(
            std::string("LineGraphUtility:initConfiguration: ") + ce.what());
    }
}

void
LineGraphUtility::initMapProvider()
{
    try
    {
        const TopologyConfig& r_topo_config = mConfig.getTopologyConfig();
        const std::string& r_provider_name = r_topo_config.providerName;

        if(r_provider_name == TopologyConfig::PROVIDER_POSTGIS)
        {
            mpMapProvider = new PostGisProvider(mConfig);
        }
        else if(r_provider_name == TopologyConfig::PROVIDER_JSONTEST)
        {
            mpMapProvider = new JsonTestProvider(mConfig);
        }
        else
        {
            throw MapProviderException("No valid MapProvider found");
        }
    }
    catch (MapProviderException& mpe)
    {
        delete mpMapProvider;
        delete mpGraphBuilder;

        throw LineGraphUtilityException(
            std::string("LineGraphUtility:initMapProvider: ") + mpe.what());
    }
}

void
LineGraphUtility::initTopology()
{
    try
    {
        mpMapProvider->getTopology(mTopology);
    }
    catch (MapProviderException& mpe)
    {
```

```cpp
            delete mpMapProvider;
            delete mpGraphBuilder;

            throw LineGraphUtilityException(
                std::string("LineGraphUtility:initTopology ") + mpe.what());
        }
}

void
LineGraphUtility::initRestrictionsAndCosts()
{
    try
    {
        mpMapProvider->setRestrictionsAndCosts(mTopology);
    }
    catch (MapProviderException& mpe)
    {
        delete mpMapProvider;
        delete mpGraphBuilder;

        throw LineGraphUtilityException(
            std::string("LineGraphUtility:initRestrictionsAndCosts ") + mpe.what());
    }
}

void
LineGraphUtility::buildGraph()
{
    try
    {
        delete mpGraphBuilder;
        mpGraphBuilder = new GraphBuilder(mTopology, mConfig);
    }
    catch (const std::exception& e)
    {
        delete mpMapProvider;
        delete mpGraphBuilder;

        throw LineGraphUtilityException(
            std::string("LineGraphUtility:buildGraph: ") + e.what());
    }
}
```

D.8.4 LineGraphUtilityException.h

```cpp
/** Exception thrown by the 'lgu' package.
 *
 * #include "LineGraphUtilityException.h"
 *
 * @author Jonas Bergman
 */

#ifndef LGU_LINEGRAPHUTILITYEXCEPTION_H_
#define LGU_LINEGRAPHUTILITYEXCEPTION_H_

// SYSTEM INCLUDES
//
```

```
#include <exception>
#include <string>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//

// FORWARD REFERENCES
//

/**
 * Exception to throw from the 'graph' package.
 * More information of the type of exception is given in the 'what()' message.
 */
class LineGraphUtilityException : public std::exception
{
public:
// LIFECYCLE
    /** Default constructor.
     */
    LineGraphUtilityException() = delete;

    /** Constructor taking a message to display.
     *
     * @param   message     The message to prepend when 'what()' is called.
     */
    LineGraphUtilityException(const std::string& rMessage) noexcept
        : std::exception(), mMessage(rMessage)
    {}

// OPERATORS
// OPERATIONS
// ACCESS
// INQUIRY
    const char* what() const noexcept
    { return (mMessage.c_str()); }

protected:
private:
// ATTRIBUTES
    std::string     mMessage;
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* LGU_LINEGRAPHUTILITYEXCEPTION_H_ */
```

D.8.5 LineGraphUtility_test.cc

```
/*
 * LineGraphUtility_test.cc
```

```
 *
 * @author  Jonas Bergman
 */

#include "../LineGraphUtility.h"

#include <chrono>

using namespace std::chrono;


#include "../../catchtest/catch.hpp"


SCENARIO ("LineGraphUtility construction", "[lgu][construction]")
{
    try
    {
        GIVEN ("a valid config file set up to use jsontest as provider")
        {
            std::string config_file(
                "catchtest/testsettings/mikh_restr_0617-testsettings.json");
            //.................................................................
            WHEN ("creating a LineGraphUtility")
            {
                THEN ("we should not receive an exception")
                {
                    REQUIRE_NOTHROW(LineGraphUtility lgu(config_file));
                }
            }
        }
    }
    catch (LineGraphUtilityException& lgue)
    {
        INFO(lgue.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (const std::exception& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
}

SCENARIO ("LineGraphUtility operation", "[lgu][operation]")
{
    try
    {
        GIVEN ("a valid config file")
        {
            std::string config_file(
                "catchtest/testsettings/mikh_restr_0617-testsettings.json");
            LineGraphUtility lgu(config_file);

            WHEN ("asking for a LineGraph")
            {
                LineGraphType* p_lg = lgu.getLineGraph();
```

```

            THEN ("we should have a line graph")
            {
                REQUIRE (p_lg != nullptr);
                REQUIRE (boost::num_edges(*p_lg) > 0);
            }

            THEN ("we should be able to print info for lines")
            {
                int i = 0;
                for(auto it = boost::edges(*p_lg);
                    (i < 10) && (it.first != it.second) ;
                    ++it.first, ++i)
                {
                    const auto& line = *(it.first);
                    NodeIdType lg_source_id =
                        boost::get(&LineGraphLine::lgSourceNodeId, *p_lg, line);
                    NodeIdType lg_target_id =
                        boost::get(&LineGraphLine::lgTargetNodeId, *p_lg, line);
                    VertexIdType topo_via_vertex_id =
                        boost::get(&LineGraphLine::topoViaVertexId, *p_lg, line);
                    double cost =
                        boost::get(&LineGraphLine::cost, *p_lg, line);

                    INFO ("LINE:   lg_source_id: " << lg_source_id
                        << ", lg_target_id: " << lg_target_id
                        << ", topo_via_vertex_id: " << topo_via_vertex_id
                        << ", cost: " << cost << "\n");
                    REQUIRE (true);

                }
            }

            THEN ("we can try to persist line graph")
            {
                try
                {
                    lgu.persistLineGraph();
                    INFO ("Persisted line graph");
                    REQUIRE (true);
                }
                catch (LineGraphUtilityException& lgue)
                {
                    INFO (lgue.what());
                    REQUIRE (false);
                }
            }

            delete p_lg;
        }

        WHEN ("asking to update restrictions and costs")
        {
            lgu.updateRestrictionsAndCosts();
            LineGraphType* p_lg {nullptr};
            p_lg = lgu.getLineGraph();

```

```
                THEN ("we should still be able to have a line graph")
                {
                    REQUIRE (p_lg != nullptr);
                    REQUIRE (boost::num_edges(*p_lg) > 0);
                }

                delete p_lg;
            }

            WHEN ("asking to update topology")
            {
                lgu.updateTopology();
                LineGraphType* p_lg {nullptr};
                p_lg = lgu.getLineGraph();

                THEN ("we should still be able to have a line graph")
                {
                    REQUIRE (p_lg != nullptr);
                    REQUIRE (boost::num_edges(*p_lg) > 0);
                }

                delete p_lg;
            }
        }
    }
    catch (LineGraphUtilityException& lgue)
    {
        INFO(lgue.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (const std::exception& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
}


SCENARIO ("LineGraphUtility timing", "[lgu][timing]")
{
    try
    {
        GIVEN ("a valid config file for Mikhailovsk without temporary topology")
        {
            std::string config_file(
                "catchtest/testsettings/mikhailovsk-original.json");

            WHEN ("when timing request for a LineGraph")
            {
                // start timing
                std::chrono::high_resolution_clock::time_point t1 =
                    std::chrono::high_resolution_clock::now();

                LineGraphType* p_lg;

                int nr_rounds = 10;
                for(int i = 0; i < nr_rounds; ++i)
```

```
                {
                    LineGraphUtility lgu(config_file);
                    p_lg = lgu.getLineGraph();
                    delete p_lg;
                    p_lg = nullptr;
                }

                // end timing;
                std::chrono::high_resolution_clock::time_point t2 =
                    std::chrono::high_resolution_clock::now();

                auto duration =
                    std::chrono::duration_cast<std::chrono::microseconds>
                        ( t2 - t1 ).count();


                THEN ("we should have an average timing")
                {
                    INFO ("Average duration over " << nr_rounds << " rounds for "
                        "Mikhailovsk without temporary topology: "
                        << duration / nr_rounds << " microseconds");
                    REQUIRE (true);
                }

                delete p_lg;
            }
        }

        GIVEN ("a valid config file for Mikhailovsk WITH temporary topology")
        {
            std::string config_file(
                "catchtest/testsettings/mikhailovsk-original-temp.json");

            WHEN ("when timing request for a LineGraph")
            {
                // start timing
                std::chrono::high_resolution_clock::time_point t1 =
                    std::chrono::high_resolution_clock::now();

                LineGraphType* p_lg;

                int nr_rounds = 10;
                for(int i = 0; i < nr_rounds; ++i)
                {
                    LineGraphUtility lgu(config_file);
                    p_lg = lgu.getLineGraph();
                    delete p_lg;
                    p_lg = nullptr;
                }

                // end timing;
                std::chrono::high_resolution_clock::time_point t2 =
                    std::chrono::high_resolution_clock::now();

                auto duration =
                    std::chrono::duration_cast<std::chrono::microseconds>
                        ( t2 - t1 ).count();
```

```


                THEN ("we should have an average timing")
                {
                    INFO ("Average duration over " << nr_rounds << " rounds for "
                        "Mikhailovsk WITH temporary topology: "
                        << duration / nr_rounds << " microseconds");
                    REQUIRE (true);
                }

                delete p_lg;
            }
        }


        GIVEN ("a valid config file for Partille without temporary topology")
        {
            std::string config_file(
                "catchtest/testsettings/partille-original.json");

            WHEN ("when timing request for a LineGraph")
            {
                // start timing
                std::chrono::high_resolution_clock::time_point t1 =
                    std::chrono::high_resolution_clock::now();

                LineGraphType* p_lg;

                int nr_rounds = 10;
                for(int i = 0; i < nr_rounds; ++i)
                {
                    LineGraphUtility lgu(config_file);
                    p_lg = lgu.getLineGraph();
                    delete p_lg;
                    p_lg = nullptr;
                }

                // end timing;
                std::chrono::high_resolution_clock::time_point t2 =
                    std::chrono::high_resolution_clock::now();

                auto duration =
                    std::chrono::duration_cast<std::chrono::microseconds>
                        ( t2 - t1 ).count();


                THEN ("we should have an average timing")
                {
                    INFO ("Average duration over " << nr_rounds << " rounds for "
                        "Partille without temporary topology: "
                        << duration / nr_rounds << " microseconds");
                    REQUIRE (true);
                }

                delete p_lg;
            }
        }
```

```
        GIVEN ("a valid config file for Partille WITH temporary topology")
        {
            std::string config_file(
                "catchtest/testsettings/partille-original-temp.json");

            WHEN ("when timing request for a LineGraph")
            {
                // start timing
                std::chrono::high_resolution_clock::time_point t1 =
                    std::chrono::high_resolution_clock::now();

                LineGraphType* p_lg;

                int nr_rounds = 10;
                for(int i = 0; i < nr_rounds; ++i)
                {
                    LineGraphUtility lgu(config_file);
                    p_lg = lgu.getLineGraph();
                    delete p_lg;
                    p_lg = nullptr;
                }

                // end timing;
                std::chrono::high_resolution_clock::time_point t2 =
                    std::chrono::high_resolution_clock::now();

                auto duration =
                    std::chrono::duration_cast<std::chrono::microseconds>
                        ( t2 - t1 ).count();

                THEN ("we should have an average timing")
                {
                    INFO ("Average duration over " << nr_rounds << " rounds for "
                        "Partille WITH temporary topology: "
                        << duration / nr_rounds << " microseconds");
                    REQUIRE (true);
                }

                delete p_lg;
            }
        }

    }
    catch (LineGraphUtilityException& lgue)
    {
        INFO(lgue.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (const std::exception& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
}

SCENARIO ("LineGraphUtility size and order", "[lgu][print_size]")
```

```
{
    try
    {
        GIVEN ("a valid config file for Mikhailovsk")
        {
            std::string config_file(
                "catchtest/testsettings/mikhailovsk-original.json");

            WHEN ("asking for size and order of graphs")
            {
                LineGraphUtility lgu(config_file);
                THEN ("we should get a print out of sizes")
                {
                    lgu.printGraphInformation("Mikhailovsk: ", std::cout);
                    REQUIRE(true);
                }
            }
        }

        GIVEN ("a valid config file for Partille")
        {
            std::string config_file(
                "catchtest/testsettings/partille-original.json");

            WHEN ("asking for size and order of graphs")
            {
                LineGraphUtility lgu(config_file);
                THEN ("we should get a print out of sizes")
                {
                    lgu.printGraphInformation("Partille: ", std::cout);
                    REQUIRE(true);
                }
            }
        }
    }
    catch (LineGraphUtilityException& lgue)
    {
        INFO(lgue.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (const std::exception& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
}
```

D.9 mapprovider

D.9.1 README.md

```
MapProvider
===========

The `mapprovider` package exists to implement different classes that can provide
 ↪ access to OpenStreetMap data.
```

```
#### Background
There exists several solutions to import OpenStreetMap data into a database, and the different solutions all creates different schemas and tables. To keep the flexibility to change how we get the OSM data, the `mapprovider` exists to provide an abstract interface that providers must implement.

#### `jsontest`
The `JsonTestProvider` is a small map provider that was implemented to be able to read in a small set of well-known edges and vertices (such as the [pgRouting sample data](http://docs.pgrouting.org/dev/doc/src/developer/sampledata.html)), to be used under development of the `Graph` class.

#### `postgis`
The `PostGisProvider` exists for working with topologies built with the `postgis_topology` extension. This is the course taken during development of the `LineGraphUtility` so far. (Using `pgRouting` seemed to be useful only for building topology and not to get access to the other map data, and `osm2po` is not open source. But with the `MapProvider` interface it is possible to implement another if desirable.)

The `PostGisProvider` uses classes `TopologyQueries` and `RestrictionQueries` for querying the database.

### Exceptions
Each subpackage throws its own exception: `MapProviderException`, and `PostGisProviderException`.
```

D.9.2 MapProvider.h

```cpp
/**  Abstract base class giving the interface for different sources of
 * topology map data.
 *
 * #include "MapProvider.h"
 *
 * @author  Jonas Bergman
 */

#ifndef MAPPROVIDER_MAPPROVIDER_H_
#define MAPPROVIDER_MAPPROVIDER_H_

// SYSTEM INCLUDES
//
#include <map>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//
#include "MapProviderException.h"
#include "../graph/Topology.h"
#include "../config/Configuration.h"
#include "../graph/Edge.h"
#include "../graph/GraphBuilder.h"
#include "../graph/Vertex.h"

```

```
// FORWARD REFERENCES
//

/**
 * Interface for getting map data from file or database.
 */
class MapProvider
{
public:
// LIFECYCLE

    /** Default constructor.
     */
    MapProvider() = delete;

    /** Constructor.
     * Construct a MapProvider based on the configurations given.
     */
    MapProvider(const Configuration& rConfig)
        : mBuildTempTopology(false), mrConfig(rConfig)
    {}

    /** Copy constructor.
     *
     * @param from The value to copy to this object.
     */
    MapProvider(const MapProvider& from) = delete;


    /** Destructor.
     */
    virtual ~MapProvider(void) {}


// OPERATORS
// OPERATIONS
    /** Fill the topology with data from the MapProvider.
     * @param   rTopology   The Topology to fill with data.
     * @throws  MapProviderException, TopologyException
     */
    virtual void    getTopology(Topology& rTopology) = 0;

    /** Read tags that might impose restrictions and costs and add them to
     * the edges in the topology.
     * @param   rTopology       The Topology with edges to get updated.
     * @throws  MapProviderException, RestrictionsException
     */
    virtual void    setRestrictionsAndCosts(Topology& rTopology) = 0;

    /** Save the line graph to persistent storage or throw exception if not
     * implemented.
     * @param   rGraph  The GraphBuilder with the LineGraph and topology to save.
     * @throws  MapProviderException
     */
    virtual void    persistLineGraph(const GraphBuilder& rGraph) = 0;

// ACCESS
```

```
// INQUIRY

protected:
    bool                 mBuildTempTopology;
    const Configuration& mrConfig;
private:
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* MAPPROVIDER_MAPPROVIDER_H_ */
```

D.9.3 MapProviderException.h

```
/**  Exception thrown by the MapProvider package.
 *
 * #include "MapProviderException.h"
 *
 * @author  Jonas Bergman
 */

#ifndef MAPPROVIDER_MAPPROVIDEREXCEPTION_H_
#define MAPPROVIDER_MAPPROVIDEREXCEPTION_H_

// SYSTEM INCLUDES
//
#include <exception>
#include <string>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//

// FORWARD REFERENCES
//

/**
 * Exception to throw from the 'MapProvider' package.
 * More information of the type of exception is given in the 'what()' message.
 */
class MapProviderException : public std::exception
{
public:
// LIFECYCLE

    /** Default constructor.
     */
    MapProviderException() = delete;

    /** Constructor taking a message to display.
     *
```

```
     * @param    message        The message to prepend when 'what()' is called.
     */
    MapProviderException(const std::string& rMessage) noexcept
        : std::exception(), mMessage(rMessage)
    {}

// OPERATORS
// OPERATIONS
// ACCESS
// INQUIRY
    const char* what() const noexcept
    { return mMessage.c_str(); }

protected:
private:
// ATTRIBUTES
    std::string        mMessage;
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* MAPPROVIDER_MAPPROVIDEREXCEPTION_H_ */
```

D.9.4 MapProvider_test.cc

```
/*
 * MapProvider_test.cc
 *
 * @author  Jonas Bergman
 */

#include "../MapProvider.h"

#include <string>
#include <map>

#include "../../catchtest/catch.hpp"
#include "../../config/ConfigurationReader.h"
#include "../../config/Configuration.h"
#include "../../config/DatabaseConfig.h"
#include "../postgis/PostGisProvider.h"
#include "../../graph/Edge.h"
#include "../../graph/Vertex.h"


SCENARIO ("MapProvider queries", "[mp-query]")
{
    try
    {
        std::string config_file("catchtest/testsettings/testsettings.json");
        ConfigurationReader config_reader(config_file);

        GIVEN ("a valid database configuration structure and "
```

```
                "name to existing topology")
        {
            Configuration config;
            config_reader.fillConfiguration(config);
            const DatabaseConfig& db_config = config.getDatabaseConfig();
            std::string topo_name("test");

            MapProvider* p_mp(nullptr);
            if(config.getTopologyConfig().providerName ==
                TopologyConfig::PROVIDER_POSTGIS)
            {
                p_mp = new PostGisProvider(config);
            }

            REQUIRE (p_mp != nullptr);

            // ..............................................................
            WHEN ("we try to fetch topology")
            {
                Topology topology;
                p_mp->getTopology(topology);

                THEN ("we should receive topology vertices and edges")
                {
                    REQUIRE (topology.nrVertices() > 0);
                    REQUIRE (topology.nrEdges() > 0);
                }
            }
            delete p_mp;
        }
    }
    catch (ConfigurationException& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (MapProviderException& dbe)
    {
        INFO(dbe.what());
        REQUIRE (false);    // force output of error and failure

    }
}
```

D.10 jsontest

D.10.1 README.md

```
JsonTest
========

This package exists for the sole purpose of testing in the initial stages of development, so that we easily could read in a small set of well-known edges and vertices.

But the test is now commented out, as it has not been updated to keep up with advances in the development.
```

```
Format of test file in json is:

```json
    {
        "vertices": [
            [1,2,0],
            [2,2,1]
        ],
        "edges": [
            [1,1,2,0]
        ]
    }
```

 where each row in `vertices` are `[id,x,y]` and each row in `edges` are `[id,
 ↪ source vertex id, target vertex id, direction]`. Values for `direction` is
 ↪ `0 = BOTH`, `1 = FROM_TO`, `2 = TO_FROM`.
```

D.10.2 JsonTestProvider.h

```cpp
/**  Read in sample topology from a json file.
 *
 * #include "JsonTestProvider.h"
 *
 * @author  Jonas Bergman
 */
#ifndef MAPPROVIDER_JSONTEST_JSONTESTPROVIDER_H_
#define MAPPROVIDER_JSONTEST_JSONTESTPROVIDER_H_

// SYSTEM INCLUDES
//

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//
#include "../MapProvider.h"

// FORWARD REFERENCES
//

/**
 * Interface for getting map data from file or database.
 */
class JsonTestProvider : public MapProvider
{
public:
// LIFECYCLE

    /** Default constructor.
     */
    JsonTestProvider() = delete;

    /** Constructor.
```

```
     * Construct a MapProvider based on the configurations given.
     */
    JsonTestProvider(const Configuration& rConfig);

    /** Copy constructor.
     *
     * @param from The value to copy to this object.
     */
    JsonTestProvider(const JsonTestProvider& from) = delete;


    /** Destructor.
     */
    virtual ~JsonTestProvider(void);


// OPERATORS
// OPERATIONS
    virtual void    getTopology(Topology& rTopology);

    virtual void    setRestrictionsAndCosts(Topology& rTopology);

    virtual void    persistLineGraph(const GraphBuilder& rGraph);

// ACCESS
// INQUIRY

protected:
private:
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* MAPPROVIDER_JSONTEST_JSONTESTPROVIDER_H_ */
```

D.10.3 JsonTestProvider.cc

```
/*
 * JsonTestProvider.cc
 *
 * @author  Jonas Bergman
 */

#include "JsonTestProvider.h"  // class implemented
#include "../../graph/Vertex.h"
#include "../../graph/Edge.h"

#include <string>
#include <boost/property_tree/ptree.hpp>
#include <boost/property_tree/json_parser.hpp>



```

```
/////////////////////////////// PUBLIC ///////////////////////////////////////

//============================= LIFECYCLE ====================================

JsonTestProvider::JsonTestProvider(const Configuration& rConfig)
    : MapProvider(rConfig)
{
}

JsonTestProvider::~JsonTestProvider()
{
}

//============================= OPERATORS ====================================

//============================= OPERATIONS ===================================
void
JsonTestProvider::getTopology(Topology& rTopology)
{
    using namespace boost::property_tree;

    const std::string& filename = mrConfig.getTopologyConfig().testFile;
    ptree              pt;

    try
    {
        read_json(filename, pt);

        // vertices
        int v_row[3];
        for(auto& row : pt.get_child("vertices"))
        {
            int i = 0;
            for(auto& item : row.second)
            {
                v_row[i] = item.second.get_value<int>();
                ++i;
            }
            rTopology.addVertex(v_row[0], Point(v_row[1], v_row[2]));
        }

        // edges
        int e_row[4];
        for(auto& row : pt.get_child("edges"))
        {
            int i = 0;
            for(auto& item : row.second)
            {
                e_row[i] = item.second.get_value<int>();
                ++i;
            }
            Edge::DirectionType direction;
            OsmIdType osm_id(std::numeric_limits<OsmIdType>::max());
            switch(e_row[3])
            {
                case 0:
                    direction = Edge::DirectionType::BOTH; break;
```

```
                case 1:
                    direction = Edge::DirectionType::FROM_TO; break;
                case 2:
                    direction = Edge::DirectionType::TO_FROM; break;
                default:
                    direction = Edge::DirectionType::BOTH;
            }
            Edge& e = rTopology.addEdge(e_row[0], osm_id, e_row[1], e_row[2]);
            Edge::RoadData rd;
            rd.direction = direction;
            e.setRoadData(rd);
        }
    }
    catch (boost::property_tree::ptree_error& e)
    {
        throw TopologyException("Could not read file " + filename);
    }
}

void
JsonTestProvider::setRestrictionsAndCosts(Topology& rTopology)
{
    //none
}

void
JsonTestProvider::persistLineGraph(const GraphBuilder& rGraph)
{
    throw MapProviderException("JsonTestProvider has not "
        "implemented persisting a Line graph");
}

//============================== ACESS      ===================================
//============================== INQUIRY    ===================================
/////////////////////////////// PROTECTED  ///////////////////////////////////

/////////////////////////////// PRIVATE    ///////////////////////////////////
```

D.10.4 JsonTestProvider_test.cc

```
///*
// * JsonTestProvider_test.cc
// *
// * @author  Jonas Bergman
// */
//
//#include "../JsonTestProvider.h"
//
//#include <string>
//#include <vector>
//
//#include "../../../catchtest/catch.hpp"
//#include "../../../config/ConfigurationReader.h"
//#include "../../../config/DatabaseConfig.h"
//#include "../../../graph/Edge.h"
//#include "../../../graph/Vertex.h"
//#include "../../../graph/Graph.h"
```

```
//
//SCENARIO ("JsonTest topology handling", "[jsontest]")
//{
//    try
//    {
//        // =================================================================
//        GIVEN ("a valid configuration structure with a jsontest filename")
//        {
//            std::string config_file("mapprovider/jsontest"
//                "/catchtest/jsontest-settings.json");
//            ConfigurationReader config_reader(config_file);
//            Configuration config;
//            config_reader.fillConfiguration(config);
//            JsonTestProvider* p_jt(nullptr);
//
//            // ..............................................................
//            WHEN ("we try to create topology")
//            {
//                THEN ("we should not receive an exception")
//                {
//                    REQUIRE_NOTHROW ( p_jt = new JsonTestProvider(config));
//                }
//            }
//
//            // ..............................................................
//            WHEN ("we try to fetch topology ")
//            {
//                Topology topology;
//                JsonTestProvider jtp(config);
//
//                // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
//                THEN ("we should not receive an exception")
//                {
//                    REQUIRE_NOTHROW (jtp.getTopology(topology));
//                }
//            }
//            // ..............................................................
//            WHEN ("using topology")
//            {
//                Topology topology;
//                JsonTestProvider jtp(config);
//                jtp.getTopology(topology);
//
//                size_t nr_vertices = 13;
//                size_t nr_edges = 16;
//
//                Configuration config;
//
//                // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
//                THEN ("we should have the right number of edges and vertices"
//                    " in the topology")
//                {
//                    REQUIRE (topology.nrVertices() == nr_vertices);
//                    REQUIRE (topology.nrEdges() == nr_edges);
//                }
//
//                // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
```

```
//              THEN ("we should be able to create a graph from topology")
//              {
//                  REQUIRE_NOTHROW (Graph graph(topology, config));
//              }
//
//              // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
//              THEN ("we should be able to create a graph from topology"
//                  " and print out the graph")
//              {
//                  Graph graph(topology, config);
//                  INFO (graph);
//                  REQUIRE (true);
//              }
//          }
//          delete p_jt;
//      }
//
//  }
//  catch (ConfigurationException& e) {
//      INFO(e.what());
//      REQUIRE (false);    // force output of error and failure
//  }
//}
//
//
```

D.10.5 jsontest-settings.json

```
{
    "database":
    {
        "host":         "127.0.0.1",
        "port":          5432,
        "username":     "tester",
        "password":     "tester",
        "database":     "mikh_style"
    },

    "topology":
    {
        "provider":       "jsontest",

        "postgis":
        {

            "topo_name":          "test",
            "roads_prefix":       "highways",
            "schema_prefix":      "topo",
            "build": {
                "temp_topo_name": "epoch_ms",
                "srid":            900913,
                "tolerance":       1.0
            },
            "edge":
            {
                "table":      "edge_data",
                "id_col":     "edge_id",
```

```
                "source_col": "start_node",
                "target_col": "end_node",
                "geom_col":   "geom"
            },
            "vertex":
            {
                "table":      "node",
                "id_col":     "node_id",
                "geom_col":   "geom"
            }
        },

        "pgrouting":
        {
        },

        "jsontest":
        {
            "test_file": "mapprovider/jsontest/catchtest/test-topology.json"
        }
    },

    "vehicle":
    {
        "category":   "motorcar",
        "motorcar":
        {
           "height":  1.6,
           "length":  4.5,
           "width":   1.9,
           "weight":  2.0,
           "maxspeed": 200
        }
    },

    "cost":
    {
       "default_speed":
       {
           "motorway":      {"high": 110, "low":  90},
           "motorway_link": {"high":  90, "low":  90},
           "trunk":         {"high":  90, "low":  60},
           "trunk_link":    {"high":  90, "low":  60},
           "primary":       {"high":  90, "low":  60},
           "primary_link":  {"high":  90, "low":  60},
           "secondary":     {"high":  90, "low":  60},
           "secondary_link":{"high":  90, "low":  60},
           "tertiary":      {"high":  90, "low":  60},
           "tertiary_link": {"high":  90, "low":  60},
           "unclassified":  {"high":  90, "low":  60},
           "residential":   {"high":  90, "low":  60},
           "living_street": {"high":  20, "low":  20}
       }
    }
}
```

D.10.6 test-topology.json

```
{
    "vertices": [
        [1,2,0],
        [2,2,1],
        [3,3,1],
        [4,4,1],
        [5,0,2],
        [6,1,2],
        [7,2,2],
        [8,3,2],
        [9,4,2],
        [10,2,3],
        [11,3,3],
        [12,4,3],
        [13,2,4]
    ],
    "edges": [
        [1,1,2,0],
        [2,2,3,1],
        [3,3,4,1],
        [4,2,7,0],
        [5,3,8,2],
        [6,5,6,0],
        [7,6,7,0],
        [8,7,8,0],
        [9,8,9,0],
        [10,7,10,0],
        [11,8,11,2],
        [12,10,11,2],
        [13,11,12,2],
        [14,10,13,0],
        [15,9,12,0],
        [16,4,9,0]
    ]
}
```

D.11 postgis

D.11.1 README.md

```
PostGisProvider
===============

This is a concrete class that implements the MapProviders interface. It fetches
↪ map data from a PostGis database where the OSM data has been imported with
↪ `osm2pgsql`, and topology has been built with the `postgis_topology`
↪ extension, converts it to valid `Vertex` and `Edge` types and stores them in
↪ a `Topology`.

The handling of the queries are factored out in different static classes:
- `CostQueries` for handling costs.
- `RestrictionQueries` for handling restrictions.
- `TopologyQueries` for handling topology related stuff.
- `LineGraphSaveQueries` for persisting a LineGraph back to the database.
```

D.11.2 PostGisProvider.h

```
/**  Handle connections with PostGis database to get map data.
 *
 * #include "PostGisProvider.h"
 *
 * @author  Jonas Bergman
 */

#ifndef MAPPROVIDER_POSTGIS_POSTGISPROVIDER_H_
#define MAPPROVIDER_POSTGIS_POSTGISPROVIDER_H_

// SYSTEM INCLUDES
//
#include <algorithm>
#include <sstream>
#include <string>
#include <vector>

// PROJECT INCLUDES
//
#include <pqxx/pqxx>        // link with -lpqxx -lpq
#include <boost/algorithm/string.hpp>

// LOCAL INCLUDES
//
#include "CostQueries.h"
#include "RestrictionQueries.h"
#include "TopologyQueries.h"
#include "../MapProvider.h"
#include "../MapProviderException.h"
#include "../../config/DatabaseConfig.h"
#include "../../graph/Edge.h"
#include "../../graph/Topology.h"
#include "../../graph/Vertex.h"
#include "../../graph/Speed.h"
#include "../../osm/OsmAccess.h"
#include "../../osm/OsmHighway.h"
#include "../../osm/OsmVehicle.h"
#include "../../util/TimeToStringMaker.h"
#include "LineGraphSaveQueries.h"

// FORWARD REFERENCES
//

/**
 * A class to handle the reading of data from the PostGis database.
 * The configurations for the connection is given.
 */
class PostGisProvider : public MapProvider
{
public:
// LIFECYCLE

    /** Default constructor.
     */
    PostGisProvider() = delete;

```

```
    /** Constructor.
     * Establish connection to database.
     *
     * @param       rConfig                             The configuration.
     * @throws       MapProviderException    If connection could not be established.
     */
    PostGisProvider(const Configuration& rConfig);

    /** Destructor.
     * Close connection to database
     */
    virtual ~PostGisProvider();


// OPERATORS
// OPERATIONS
    virtual void    getTopology(Topology& rTopology);

    virtual void    setRestrictionsAndCosts(Topology& rTopology);

    virtual void    persistLineGraph(const GraphBuilder& rGraph);

// INQUIRY

protected:

private:
// HELPERS

    /** Get edges from database.
     * @param    rEdgeResult   Result of db query for Edges.
     * @throws        MapProviderException
     */
    void    getTopologyEdges(pqxx::result& rEdgeResult);

    /** Add edges to topology.
     * @param    rEdgeResult    Result of db query for edges.
     * @param    rTopology      Topology to fill with edges.
     * @throws   TopologyException
     */
    void    addEdgeResultToTopology(const pqxx::result& rEdgeResult,
                                    Topology& rTopology);

    /** Get vertices from database.
     * @throws        MapProviderException
     */
    void    getTopologyVertices(pqxx::result& rVertexResult);

    /** Add vertices to topology.
     * @throws   TopologyException
     */
    void    addVertexResultToTopology(
                const pqxx::result& rResult,
                Topology&           rTopology);

    // Helpers for constructor
    /** Set the base name for the topology, either a string from config
```

```
     * or a timestamp.
     */
    void    setTopoBaseName(std::string& rTopoBaseName);

    /** Build a PostGIS topology with name given in constructor.
     * @param       srid                The SRID for the projection to use
     * @param       tolerance       The distance to look for merging vertices, unit of srid.
     * @throws       MapProviderException
     */
    void        buildTopology(int srid, double tolerance);

    /** Remove PostGIS topology (tables and schema) from the database.
     * @throws       MapProviderException
     */
    void        removeTopology();


    // Restriction helpers ---------------------------------------------------

    /** Add restrictions to edges.
     * @param   rTopology  Adding EdgeRestricion to Edges in topology.
     * @throw   MapProviderException
     */
    void    addEdgeRestrictions(Topology& rTopology);

    /** Get VehicleProperty restrictions
     * Helper for 'getEdgeRestrictions()'
     * @param   rResult     Store the result of query in here.
     * @throw   MapProviderException
     */
    void    getVehiclePropertyEdgeRestrictions(pqxx::result& rResult);

    /** Add the result of the query for vehicle properties to Edge's restrictions.
     * Helper for 'getEdgeRestrictions()'
     * @param   rResult         The results of the query
     * @param   rTopology       Update affected edges in the topology.
     * @throw   MapProviderException
     */
    void    addVehiclePropertyRestrictionsToEdge(
                const pqxx::result&    rResult,
                Topology&              rTopology);

    /** Get Access restrictions to edge.
     * Helper for 'getEdgeRestrictions()'
     * @param   rResult     Store the result of query in here.
     * @throw   MapProviderException
     */
    void    getAccessRestrictions(pqxx::result& rResult);

    /** Add the result of the query for Access to restrictions.
     * Helper for 'getEdgeRestrictions()'
     * @param   rResult         The results of the query
     * @param   rTopology       Update affected edges in the topology.
     * @throw   MapProviderException
     */
    void    addAccessRestrictionsToEdge(
                const pqxx::result&    rResult,
```

```
                Topology&              rTopology);

    /** Get Turning restrictions for traveling from edge.
     * Helper for 'getEdgeRestrictions()'.
     * Turning restrictions are relations and not easily handled with
     * osm2pgsql. Therefore we must use 'slim' mode when converting OSM to
     * PostGis, and use the table 'planet_osm_rels' and column 'tags' to look
     * for a 'restriction'. If we find one we have to parse the 'members'
     * column ourselves.
     * @param   rResult     Store the result of query in here.
     * @throw   MapProviderException
     */
    void    getTurningRestrictions(pqxx::result& rResult);

    /** Add the result of the query for Turning restrictions.
     * Helper for 'getEdgeRestrictions()'
     * @param   rResult         The results of the query
     * @param   rTopology       Update affected edges in the topology.
     * @throw   MapProviderException
     */
    void    addTurningRestrictionsToEdge(
                const pqxx::result&    rResult,
                Topology&              rTopology);

    /** Get restrictions defined at points but applicable to edges,
     * such as barriers and railway crossings.
     * @param   rResult     Store the result of query in here.
     * @throw   MapProviderException
     */
    void    getEdgePointRestrictions(pqxx::result& rResult);

    /** Add the result of the query for Point restrictions on Edges .
     * Helper for 'getEdgeRestrictions()'
     * @param   rResult         The results of the query
     * @param   rTopology       Update affected edges in the topology.
     * @throw   MapProviderException
     */
    void    addPointRestrictionsToEdge(
                const pqxx::result&    rResult,
                Topology&              rTopology);

    // Costs ---------------------------------------------------------------
    /** Add costs to the edge.
     * @param   The Topology with Edges to add cost to.
     */
    void    addEdgeCosts(Topology& rTopology);

    /** Get costs for the travel time along an edge.
     * The length is constant in the topology but we need to find out if
     * there is a max speed restriction or if there is a bad surface.
     * If no such restrictions are in the database then the default speed
     * for the road category is used.
     * @param   rResult     Store the result of the query here.
     * @throw   MapProviderException
     */
    void    getTravelTimeCosts(pqxx::result& rResult);

```

```
    /** Add costs for travel time along the edge.
     * First set the speed of those with explicit restrictions in database,
     * then set the default speed for those without explicit speeds.
     * @param   rResult     The results of the query.
     * @param   rTopology   The topology with edges to set cost for.
     * @throw   MapProviderException
     */
    void    addTravelTimeCosts(const pqxx::result& rResult, Topology& rTopology);

    /** Get other costs for the edge other than speed and barriers, those
     * include slowdown at stop and yield signs, zebra crossings, railway
     * crossings, bus stops, speed bumps, traffic lights...
     * @param   rResult     Store the result of the query here.
     */
    void    getOtherEdgeCosts(pqxx::result& rResult);

    /** Add costs for speed bumps and such to affected edges.
     * @param   rResult     The results of the query.
     * @param   rTopology   The topology with edges to set cost for.
     * @throw   MapProviderException
     */
    void    addOtherCosts(const pqxx::result& rResult, Topology& rTopology);

    // LineGraph persistence -------------------------------------------------
    /** Set up the schema and tables needed to persist the line graph.
     */
    void    setUpSchemaAndTables();

    /** Create a new schema in the database
     * @throw   MapProviderException
     */
    void    createLineGraphSchema();

    /** Create the needed tables in the database
     * @throw   MapProviderException
     */
    void    createLineGraphTables();

    /** Insert the data in the database.
     * @param   rGraph      The graph with data.
     * @throw   MapProviderException
     */
    void    insertData(const GraphBuilder& rGraph);

    /** Prepare the LineGraph data for inserting into the database.
     * @param   rTrans      The transaction to operate within.
     * @param   rGraph      The graph data
     * @throw   MapProviderException
     */
    void    prepareLineGraphData(
                pqxx::transaction_base& rTrans,
                const GraphBuilder&           rGraph);

    // Generic helpers to clean up the code some ----------------------------
    /** Check that the connection with the database is up, or throw exception.
     * @throw   MapProviderException
     */
```

```
    void    testConnection();

// ATTRIBUTES
    const Configuration&    mConfig;
    const DatabaseConfig&   mDbConfig;
    const TopologyConfig&   mTopoConfig;
    pqxx::connection        mConnection;
    std::string             mOsmEdgeTable;
    std::string             mPointTableName;
    std::string             mSchemaName;
    std::string             mTopoEdgeTable;
    std::string             mEdgeIdCol;
    std::string             mSourceCol;
    std::string             mTargetCol;
    std::string             mEdgeGeomCol;
    std::string             mTopoVertexTable;
    std::string             mVertexIdCol;
    std::string             mVertexGeomCol;
    std::string             mLineGraphSchema;
    std::string             mLineGraphNodeTable;
    std::string             mLineGraphLineTable;

// CONSTANTS
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* MAPPROVIDER_POSTGIS_POSTGISPROVIDER_H_ */
```

D.11.3 PostGisProvider.cc

```
/*
 * PostGisProvider.cc
 *
 * @author  Jonas Bergman
 */

#include "PostGisProvider.h"  // class implemented
#include "TopologyQueries.h"
#include "RestrictionQueries.h"
#include "CostQueries.h"

#include "../../graph/Edge.h"
#include "../../osm/OsmId.h"
#include "../../graph/EdgeRestriction.h"
#include "../../graph/EdgeCost.h"



/////////////////////////////// PUBLIC ///////////////////////////////////////

//============================= LIFECYCLE ====================================
PostGisProvider::PostGisProvider(const Configuration& rConfig)
```

```
try
    : MapProvider(rConfig),
      mConfig(rConfig),
      mDbConfig(rConfig.getDatabaseConfig()),
      mTopoConfig(rConfig.getTopologyConfig()),
      mConnection(mDbConfig.getConnectionString())
{
    try
    {
        testConnection();

        std::string topoBaseName;
        setTopoBaseName(topoBaseName);

        if(topoBaseName == "")
        {
            throw MapProviderException("No topology specified.");
        }

        pqxx::nontransaction nt(mConnection);
        mOsmEdgeTable       = nt.esc(mTopoConfig.roadsPrefix + "_" +
                                topoBaseName);
        mPointTableName     = nt.esc("planet_osm_point");
        mSchemaName         = nt.esc(mTopoConfig.topologySchemaPrefix + "_" +
                                topoBaseName);
        mTopoEdgeTable      = nt.esc(mSchemaName + "." +
                                mTopoConfig.edgeTableName);
        mEdgeIdCol          = nt.esc(mSchemaName + "." +
                                mTopoConfig.edgeIdColumnName);
        mSourceCol          = nt.esc(mSchemaName + "." +
                                mTopoConfig.sourceColumnName);
        mTargetCol          = nt.esc(mSchemaName + "." +
                                mTopoConfig.targetColumnName);
        mEdgeGeomCol        = nt.esc(mSchemaName + "." +
                                mTopoConfig.edgeGeomColumnName);
        mTopoVertexTable    = nt.esc(mSchemaName + "." +
                                mTopoConfig.vertexTableName);
        mVertexIdCol        = nt.esc(mSchemaName + "." +
                                mTopoConfig.vertexIdColumnName);
        mVertexGeomCol      = nt.esc(mSchemaName + "." +
                                mTopoConfig.vertexGeomColumnName);
        mLineGraphSchema    = nt.esc("line_graph_generated");
        mLineGraphNodeTable = nt.esc(mLineGraphSchema + ".node");
        mLineGraphLineTable = nt.esc(mLineGraphSchema + ".line");
        nt.abort();

        if(mBuildTempTopology)
        {
            buildTopology(mTopoConfig.srid, mTopoConfig.tolerance);
        }
    }
    catch(const std::exception& e)
    {
        throw MapProviderException(
            std::string("PostGisProvider:ctor(in): ") + e.what());
    }
}
```

```
// catch error in initializer list (opening connection)
catch(const std::exception& e)
{
        throw MapProviderException(
            std::string("PostGisProvider:ctor(out): ") + e.what());
}

PostGisProvider::~PostGisProvider()
{
    try
    {
        if(mBuildTempTopology)
        {
            removeTopology();
        }
        if(mConnection.is_open())
        {
            mConnection.disconnect();
        }
    }
    catch(const std::exception& e)
    {
        throw MapProviderException(
            std::string("PostGisProvider:dtor: ") + e.what());
    }
}

//============================= OPERATORS ====================================

//============================= OPERATIONS ===================================
void
PostGisProvider::getTopology(Topology& rTopology)
{
    pqxx::result vertex_result;
    getTopologyVertices(vertex_result);
    addVertexResultToTopology(vertex_result, rTopology);

    pqxx::result edge_result;
    getTopologyEdges(edge_result);
    addEdgeResultToTopology(edge_result, rTopology);
}

void
PostGisProvider::setRestrictionsAndCosts(Topology& rTopology)
{
    addEdgeRestrictions(rTopology);
    addEdgeCosts(rTopology);
}

void
PostGisProvider::persistLineGraph(const GraphBuilder& rGraph)
{
    setUpSchemaAndTables();
    insertData(rGraph);
}

//============================= ACESS      ===================================
```

```
//============================ INQUIRY    ===================================
////////////////////////////// PROTECTED  ///////////////////////////////////
////////////////////////////// PRIVATE    ///////////////////////////////////
void
PostGisProvider::getTopologyVertices(pqxx::result& rVertexResult)
{
    try
    {
        testConnection();

        // NON-TRANSACTION START
        pqxx::nontransaction transaction(mConnection);

        TopologyQueries::getTopologyVertices(
            transaction,
            rVertexResult,
            mTopoVertexTable);
    }
    catch(const std::exception& e)
    {
        throw MapProviderException(
            std::string("PostGisProvider:getTopologyVertices: ") + e.what());
    }
}

void
PostGisProvider::addVertexResultToTopology(
    const pqxx::result& rResult,
    Topology&           rTopology)
{
    TopologyQueries::addVertexResultToTopology(rResult, rTopology);
}


void
PostGisProvider::getTopologyEdges(pqxx::result& rEdgeResult)
{
    try
    {
        testConnection();

        // NON-TRANSACTION START
        pqxx::nontransaction transaction(mConnection);

        TopologyQueries::getTopologyEdges(
            transaction,
            rEdgeResult,
            mTopoEdgeTable,
            mSchemaName,
            mOsmEdgeTable);
    }
    catch(const std::exception& e)
    {
        throw MapProviderException(
            std::string("PostGisProvider:getTopoEdges: ") + e.what());
    }
}
```

```

void
PostGisProvider::addEdgeResultToTopology(
    const pqxx::result& rResult,
    Topology&           rTopology)
{
    TopologyQueries::addEdgeResultToTopology(rResult, rTopology);
}

void
PostGisProvider::buildTopology(int srid, double tolerance)
{
    try
    {
        testConnection();

        // TRANSACTION START
        pqxx::work transaction(mConnection);

        try
        {
            TopologyQueries::installPostgisTopology(transaction);
            TopologyQueries::setSearchPath(transaction);
            TopologyQueries::createTemporaryTable(transaction, mOsmEdgeTable);
            TopologyQueries::createTemporarySchema(
                transaction, mSchemaName, srid);
            TopologyQueries::addTopoGeometryColumn(
                transaction, mSchemaName, mOsmEdgeTable);
            TopologyQueries::fillTopoGeometryColumn(
                transaction, mSchemaName, mOsmEdgeTable, tolerance);

            // TRANSACTION END
            transaction.commit();
        }
        catch (const std::exception& e)
        {
            transaction.abort();
            throw e;
        }
    }
    catch(const std::exception& e)
    {
        throw MapProviderException(
            std::string("PostGisProvider:buildTopology: ") + e.what());
    }
}


void
PostGisProvider::removeTopology()
{
    try
    {
        testConnection();

        // TRANSACTION START
        pqxx::work transaction(mConnection);
```

```

        try
        {
            TopologyQueries::dropTemporaryTable(transaction, mOsmEdgeTable);
            TopologyQueries::dropTemporarySchema(transaction, mSchemaName);
            TopologyQueries::deleteTemporaryLayerRecord(transaction, mOsmEdgeTable);
            TopologyQueries::deleteTemporaryTopoRecord(transaction, mSchemaName);

            // TRANSACTION END
            transaction.commit();
        }
        catch (const std::exception& e)
        {
            transaction.abort();
            throw e;
        }
    }
    catch(const std::exception& e)
    {
        throw MapProviderException(std::string(
            "PostGisProvider:removeTopology: ") + e.what());
    }
}

void
PostGisProvider::setTopoBaseName(std::string& rTopoBaseName)
{
    if(mTopoConfig.tempTopoName == TopologyConfig::TEMP_TOPO_NAMEBASE)
    {
        rTopoBaseName = TimeToStringMaker::getEpochMsTimeString();
        mBuildTempTopology = true;
    }
    else
    {
        rTopoBaseName = mTopoConfig.topoName;
    }
}

// Restrictions -------------------------------------------------------------
void
PostGisProvider::addEdgeRestrictions(Topology& rTopology)
{
    pqxx::result result;

    getVehiclePropertyEdgeRestrictions(result);
    addVehiclePropertyRestrictionsToEdge(result, rTopology);

    result.clear();
    getAccessRestrictions(result);
    addAccessRestrictionsToEdge(result, rTopology);

    result.clear();
    getTurningRestrictions(result);
    addTurningRestrictionsToEdge(result, rTopology);

    result.clear();
    getEdgePointRestrictions(result);
```

```
    addPointRestrictionsToEdge(result, rTopology);
}

void
PostGisProvider::getVehiclePropertyEdgeRestrictions(pqxx::result& rResult)
{
    try
    {
        testConnection();

        // NON-TRANSACTION START
        pqxx::nontransaction transaction(mConnection);

        RestrictionQueries::getVehiclePropertyEdgeRestrictions(
            transaction,
            rResult,
            mTopoEdgeTable,
            mOsmEdgeTable,
            mSchemaName
        );
    }
    catch(const std::exception& e)
    {
        throw MapProviderException(
            std::string("PostGisProvider:getVehiclePropertyEdgeRestrictions: ")
                        + e.what());
    }
}

void
PostGisProvider::addVehiclePropertyRestrictionsToEdge(
    const pqxx::result& rResult,
    Topology&           rTopology)
{
    RestrictionQueries::addVehiclePropertyRestrictionsToEdge(rResult, rTopology);
}

void
PostGisProvider::getAccessRestrictions(pqxx::result& rResult)
{
    try
    {
        testConnection();

        // NON-TRANSACTION START
        pqxx::nontransaction transaction(mConnection);

        RestrictionQueries::getAccessRestrictions(
            transaction,
            rResult,
            mTopoEdgeTable,
            mOsmEdgeTable,
            mSchemaName);
    }
    catch(const std::exception& e)
    {
        throw MapProviderException(
```

```
            std::string("PostGisProvider:getAccessRestrictions: ")
        + e.what());
    }
}

void
PostGisProvider::addAccessRestrictionsToEdge(
    const pqxx::result& rResult,
    Topology&           rTopology)
{
    RestrictionQueries::addAccessRestrictionsToEdge(rResult, rTopology, mConfig);
}

void
PostGisProvider::getTurningRestrictions(pqxx::result& rResult)
{
    try
    {
        testConnection();

        // TRANSACTION START
        pqxx::nontransaction transaction(mConnection);

        try
        {
            RestrictionQueries::dropCreateTurningRestrictionsTable(transaction);
            RestrictionQueries::identifyTurningRestrictions(
                transaction,
                mOsmEdgeTable,
                mTopoEdgeTable);
            RestrictionQueries::getTurningRestrictions(transaction, rResult);
        }
        catch (const std::exception& e)
        {
            transaction.abort();
            throw e;
        }
    }
    catch(const std::exception& e)
    {
        throw MapProviderException(
            std::string("PostGisProvider:getTurningRestrictions: ")
        + e.what());
    }
}

void
PostGisProvider::addTurningRestrictionsToEdge(
    const pqxx::result&     rResult,
    Topology&               rTopology)
{
    RestrictionQueries::addTurningRestrictionsToEdge(rResult, rTopology);
}

void
PostGisProvider::getEdgePointRestrictions(pqxx::result& rResult)
{
```

```
    try
    {
        testConnection();

        // NON-TRANSACTION START
        pqxx::nontransaction transaction(mConnection);

        RestrictionQueries::getEdgePointRestrictions(
            transaction,
            rResult,
            mPointTableName,
            mTopoEdgeTable,
            mOsmEdgeTable,
            mSchemaName);
    }
    catch(const std::exception& e)
    {
        throw MapProviderException(
            std::string("PostGisProvider:getEdgePointRestrictions: ") + e.what());
    }
}

void
PostGisProvider::addPointRestrictionsToEdge(
    const pqxx::result&     rResult,
    Topology&               rTopology)
{
    RestrictionQueries::addPointRestrictionsToEdge(rResult, rTopology, mConfig);
}

// Costs ---------------------------------------------------------------------
void
PostGisProvider::addEdgeCosts(Topology& rTopology)
{
    pqxx::result result;

    getTravelTimeCosts(result);
    addTravelTimeCosts(result, rTopology);

    // barrier costs are added while looking for restrictions

    result.clear();
    getOtherEdgeCosts(result);
    addOtherCosts(result, rTopology);
}

void
PostGisProvider::getTravelTimeCosts(pqxx::result& rResult)
{
    try
    {
        testConnection();

        // NON-TRANSACTION START
        pqxx::nontransaction transaction(mConnection);

        CostQueries::getTravelTimeEdgeCosts(
```

```
                transaction,
                rResult,
                mTopoEdgeTable,
                mOsmEdgeTable,
                mSchemaName
            );
        }
        catch(const std::exception& e)
        {
            throw MapProviderException(
                std::string("PostGisProvider:getTravelTimCost: ")
                            + e.what());
        }
}

void
PostGisProvider::addTravelTimeCosts(
    const pqxx::result& rResult,
    Topology&           rTopology)
{
    CostQueries::addTravelTimeCosts(rResult, rTopology, mConfig);
}

void
PostGisProvider::getOtherEdgeCosts(pqxx::result& rResult)
{
    try
    {
        testConnection();

        // NON-TRANSACTION START
        pqxx::nontransaction transaction(mConnection);

        CostQueries::getOtherCosts(
            transaction,
            rResult,
            mPointTableName,
            mTopoEdgeTable,
            mOsmEdgeTable,
            mSchemaName);
    }
    catch(const std::exception& e)
    {
        throw MapProviderException(
            std::string("PostGisProvider:getOtherEdgeCosts: ") + e.what());
    }
}

void
PostGisProvider::addOtherCosts(
    const pqxx::result&     rResult,
    Topology&               rTopology)
{
    CostQueries::addOtherCosts(rResult, rTopology, mConfig);
}

//  LineGraph persistence ----------------------------------------------------
```

```
void
PostGisProvider::setUpSchemaAndTables()
{
    createLineGraphSchema();
    createLineGraphTables();
}

void
PostGisProvider::createLineGraphSchema()
{
    try
    {
        testConnection();

        // NON-TRANSACTION START
        pqxx::nontransaction transaction(mConnection);

        LineGraphSaveQueries::dropCreateSchema(transaction, mLineGraphSchema);
    }
    catch(const std::exception& e)
    {
        throw MapProviderException(
            std::string("PostGisProvider:createLineGraphSchema: ") + e.what());
    }
}

void
PostGisProvider::createLineGraphTables()
{
    try
    {
        testConnection();

        // NON-TRANSACTION START
        pqxx::nontransaction transaction(mConnection);

        LineGraphSaveQueries::dropCreateLineTable(transaction, mLineGraphLineTable);
        LineGraphSaveQueries::dropCreateNodeTable(transaction, mLineGraphNodeTable);
    }
    catch(const std::exception& e)
    {
        throw MapProviderException(
            std::string("PostGisProvider:createLineGraphTables: ") + e.what());
    }
}

void
PostGisProvider::insertData(const GraphBuilder& rGraph)
{
    try
    {
        testConnection();

        pqxx::work transaction(mConnection);

        try
        {
```

```
            prepareLineGraphData(transaction, rGraph);

            // TRANSACTION END
            transaction.commit();
        }
        catch (const std::exception& e)
        {
            transaction.abort();
            throw e;
        }
    }
    catch(const std::exception& e)
    {
        throw MapProviderException(
            std::string("PostGisProvider:insertData: ") + e.what());
    }
}

void
PostGisProvider::prepareLineGraphData(
    pqxx::transaction_base& rTrans,
    const GraphBuilder&            rGraph)
{
    const LineGraphType& rLineGraph = rGraph.getBoostLineGraph();
    const Topology&      rTopology  = rGraph.getTopology();

    for(auto line_it = boost::edges(rLineGraph);
        line_it.first != line_it.second;
        ++line_it.first)
    {
        const LineType& line = *(line_it.first);

        NodeIdType source_node_id = rLineGraph[line].lgSourceNodeId;
        NodeIdType target_node_id = rLineGraph[line].lgTargetNodeId;

        Cost cost = rLineGraph[line].cost;

        const NodeType& source_node = rGraph.getLineGraphNode(source_node_id);
        const NodeType& target_node = rGraph.getLineGraphNode(target_node_id);

        EdgeIdType source_edge_id = rLineGraph[source_node].topoEdgeId;
        EdgeIdType target_edge_id = rLineGraph[target_node].topoEdgeId;


        const Edge& sourceEdge = rTopology.getEdge(source_edge_id);
        const Edge& targetEdge = rTopology.getEdge(target_edge_id);

        const Point& sourcePoint = sourceEdge.geomData().centerPoint;
        const Point& targetPoint = targetEdge.geomData().centerPoint;

        std::string sourceWKT = "POINT(" + std::to_string(sourcePoint.x) + " "
                                + std::to_string(sourcePoint.y) + ")";
        std::string targetWKT = "POINT(" + std::to_string(targetPoint.x) + " "
                                + std::to_string(targetPoint.y) + ")";
        std::string lineWKT = "LINESTRING(" + std::to_string(sourcePoint.x) + " " +
                                      std::to_string(sourcePoint.y) + ", " +
                                      std::to_string(targetPoint.x) + " " +
```

```
                                                std::to_string(targetPoint.y) + ")";

        LineGraphSaveQueries::insertNode(
            rTrans,
            mLineGraphNodeTable,
            source_edge_id,
            sourceWKT);

        LineGraphSaveQueries::insertNode(
            rTrans,
            mLineGraphNodeTable,
            target_edge_id,
            targetWKT);

        LineGraphSaveQueries::insertLine(
            rTrans,
            mLineGraphLineTable,
            cost,
            lineWKT);
    }
}

void
PostGisProvider::testConnection()
{
    if(!mConnection.is_open())
    {
        throw MapProviderException(
            std::string("Could not open ") + mDbConfig.database);
    }
}
```

D.11.4 CostQueries.h

```
/**  Queries for PostGisProvider to find costs.
 *
 * #include "CostQueries.h"
 *
 * @author  Jonas Bergman
 */
#ifndef MAPPROVIDER_POSTGIS_COSTQUERIES_H_
#define MAPPROVIDER_POSTGIS_COSTQUERIES_H_

// SYSTEM INCLUDES
//
#include <string>
#include <sstream>
#include <vector>

// PROJECT INCLUDES
//
#include <boost/algorithm/string.hpp>
#include <pqxx/pqxx>

// LOCAL INCLUDES
//
#include "../../osm/OsmHighway.h"
```

```
#include "../../graph/Edge.h"
#include "../../graph/EdgeRestriction.h"
#include "../../graph/Topology.h"
#include "../../graph/Vertex.h"
#include "../MapProviderException.h"

/** Class for holding static queries about costs,
 * needed by the PostGisProvider.
 */
class CostQueries
{
public:
// TYPES

    struct TravelTimeCostResult
    {
        enum Columns
        {
            EDGE_ID,
            ELEMENT_ID,
            MAXSPEED,
            SURFACE
        };
    };

    struct OtherCostResult
    {
        enum Columns
        {
            OSM_ID,
            HIGHWAY,
            RAILWAY,
            PUBLIC_TRANSPORT,
            TRAFFIC_CALMING,
            EDGE_ID
        };
    };

// LIFECYCLE
    CostQueries() = delete;
    CostQueries(const CostQueries& from) = delete;
    ~CostQueries() = default;

// OPERATORS
// OPERATIONS
    /** Query for costs for travel time, meaning we need to find the speed.
     * @param   rTrans          Transaction to perform query in.
     * @param   rResult         Store the result of query here.
     * @param   rTopoEdgeTable  Name of table with topology edges.
     * @param   rOsmEdgeTable   Name of table with OSM edges.
     * @param   rSchemaName     Name of the topology schema.
     * @throw   pqxx::pqxx_exception
     */
    static void     getTravelTimeEdgeCosts(
                        pqxx::transaction_base& rTrans,
                        pqxx::result&           rResult,
                        const std::string&      rTopoEdgeTable,
```

```
                            const std::string&      rOsmEdgeTable,
                            const std::string&      rSchemaName);

    /** Add costs for travel time along the edge.
     * First set the speed of those with explicit restrictions in database,
     * then set the default speed for those without explicit speeds.
     * @param   rResult     The results of the query.
     * @param   rTopology   The topology with edges to set cost for.
     * @param   rConfig         Configuration
     * @throw   MapProviderException
     */
    static void     addTravelTimeCosts(
                            const pqxx::result&     rResult,
                            Topology&               rTopology,
                            const Configuration&    rConfig);

    /** Add cost relating to the maxspeed of the edge.
     * The cost is the number of seconds to travel the edge.
     * @param   rEdge           The edge to add cost to
     * @param   speed           The speed for the edge found in the database.
     * @param   surfaceString   The surface as string or empty if not specified.
     * @param   rConfig         Configuration
     * @throw   MapProviderException
     */
    static void     addTravelTimeCostToEdge(
                            Edge&                   rEdge,
                            Speed                   speed,
                            std::string&            surfaceString,
                            const Configuration&    rConfig);

    /** If the speed in the db was not set we must fetch the default
     * for this road category from the configuration.
     * @param   rEdge   The edge to find the default speed for
     * @param   rConfig         Configuration
     * @return  The default speed for this type of highway.
     */
    static Speed    getDefaultSpeedForEdge(
                            const Edge&             rEdge,
                            const Configuration&    rConfig);

    /** Query for costs under the highway and railway tags:
     * Highway:
     *  - bus_stop
     *  - crossing
     *  - give_way
     *  - mini_roundabout
     *  - stop
     *  - traffic_signals
     * Railway:
     *  - level_crossing
     * Traffic calming
     *
     * @param   rTrans          Transaction to perform query in
     * @param   rResult         Store the result of the query here
     * @param   rOsmPointTable  Name of table with OSM points (nodes)
     * @param   rTopoEdgeTable  Name of table with topology edges.
     * @param   rOsmEdgeTable   Name of table with OSM edges.
```

```
     * @param   rSchemaName     The name of the schema with topology info.
     */
    static void     getOtherCosts(
                        pqxx::transaction_base&     rTrans,
                        pqxx::result&               rResult,
                        const std::string&          rOsmPointTable,
                        const std::string&          rTopoEdgeTable,
                        const std::string&          rOsmEdgeTable,
                        const std::string&          rSchemaName);

    /** Add costs for speed bumps and such to affected edges.
     * @param   rResult     The results of the query.
     * @param   rTopology   The topology with edges to set cost for.
     * @param   rConfig     The Configuration
     * @throw   MapProviderException
     */
    static void     addOtherCosts(
                        const pqxx::result&     rResult,
                        Topology&               rTopology,
                        const Configuration&    rConfig);

    /** Add a cost of an other type to the edge.
     * Look up the value in the configuration.
     * @param   rEdge   The Edge to add a cost to.
     * @param   key     The type of cost as a string
     * @param   rConfig     The Configuration
     */
    static void     addOtherCostToEdge(
                        Edge&                   rEdge,
                        const std::string&      key,
                        const Configuration&    rConfig);

    /** While looking for restrictions and we come across barriers,
     * add the costs for barriers if they incur costs.
     * @param   edge    The edge with a barrier.
     * @param   type    The type of barrier.
     * @param   rConfig The Configuration to use for the cost.
     */
    static void     addBarrierCostToEdge(
                        Edge&                   rEdge,
                        OsmBarrier::BarrierType type,
                        const Configuration&    rConfig);

// ACCESS
// INQUIRY

protected:
private:
    /** SELECT FROM JOIN */
    static std::string  startOfQuery(const std::string& rTopoEdgeTable);

    /** Which columns to pick */
    static std::string  queryColumns(const std::vector<std::string>& rCols);

    /** FROM JOIN ON WHERE */
    static std::string  midOfQuery(
        const std::string& rSchemaName,
```

```
        const std::string& rOsmEdgeTable);

    /** Make sure only to pick rows with content in some column. */
    static std::string  notNullColumns(const std::vector<std::string>& rCols);

    /** AS ON ORDER BY */
    static std::string  endOfQuery();
};

#endif /* MAPPROVIDER_POSTGIS_COSTQUERIES_H_ */
```

D.11.5 CostQueries.cc

```
/*
 * CostQueries.cc
 *
 * @author  Jonas Bergman
 */

#include "CostQueries.h"  // class implemented

/////////////////////////////// PUBLIC ///////////////////////////////////////

//============================= LIFECYCLE ====================================
//============================= OPERATORS ====================================
//============================= OPERATIONS ===================================
//static
void
CostQueries::getTravelTimeEdgeCosts(
    pqxx::transaction_base& rTrans,
    pqxx::result&           rResult,
    const std::string&      rTopoEdgeTable,
    const std::string&      rOsmEdgeTable,
    const std::string&      rSchemaName)
{
    std::vector<std::string> columns {
        "maxspeed",
        "surface"
    };

    rResult = rTrans.exec(
        startOfQuery(rTopoEdgeTable) +
        queryColumns(columns) +
        midOfQuery(rSchemaName, rOsmEdgeTable) +
//        notNullColumns(columns) +
        endOfQuery()
    );
}

// static
void
CostQueries::addTravelTimeCosts(
    const pqxx::result&     rResult,
    Topology&               rTopology,
    const Configuration&    rConfig)
{
    try
```

```
    {
        for(const pqxx::tuple& row : rResult)
        {
            // throw exception if no edgeId
            EdgeIdType edgeId =
                row[TravelTimeCostResult::EDGE_ID].as<EdgeIdType>();

            Edge& edge = rTopology.getEdge(edgeId);

            Speed speed =
                row[TravelTimeCostResult::MAXSPEED].as<Speed>(
                    EdgeRestriction::VehicleProperties::DEFAULT_SPEED_MAX);
            std::string surface_string =
                row[TravelTimeCostResult::SURFACE].as<std::string>("");

            addTravelTimeCostToEdge(edge, speed, surface_string, rConfig);
        }
    }
    catch (std::exception& e)
    {
        throw MapProviderException(
            std::string("PostGisProvider:addTravelTimeCost: ") + e.what());
    }
}

//static
void
CostQueries::addTravelTimeCostToEdge(
    Edge&                   rEdge,
    Speed                   speed,
    std::string&            surfaceString,
    const Configuration&    rConfig)
{
    bool hasMaxSpeed =
        (speed != EdgeRestriction::VehicleProperties::DEFAULT_SPEED_MAX);
    bool hasSurface = surfaceString.length() > 0;
    if(!(hasMaxSpeed || hasSurface))
    {
        speed = getDefaultSpeedForEdge(rEdge, rConfig);
    }
    // look if surface restricts speed
    else if(hasSurface)
    {
        try
        {
            OsmHighway::SurfaceType surface =
                OsmHighway::parseSurfaceString(surfaceString);
            Speed surfaceSpeed =
                rConfig.getCostConfig().surfaceMaxSpeed.getSurfaceMaxSpeed(surface);
            if(surfaceSpeed < speed)
            {
                speed = surfaceSpeed;
            }
        }
        catch (OsmException& e)
        {
            throw MapProviderException(
```

```
                std::string("CostQueries:addTravelTime... ") +
                "could not parse surface " + surfaceString);
            }
        }
        double speed_mps = speed / 3.6;
        double travel_time = rEdge.geomData().length/ speed_mps;
        rEdge.edgeCost().addCost(EdgeCost::TRAVEL_TIME, travel_time);
        rEdge.setSpeed(speed);
}

// static
Speed
CostQueries::getDefaultSpeedForEdge(
    const Edge&             rEdge,
    const Configuration&    rConfig)
{
    OsmHighway::HighwayType type = rEdge.roadData().roadType;
    const CostConfig& costConfig = rConfig.getCostConfig();
    Speed speed=
        costConfig.defaultSpeed.getDefaultSpeed(type, CostConfig::DefaultSpeed::LOW);
    return speed;
}

// static
void
CostQueries::getOtherCosts(
    pqxx::transaction_base& rTrans,
    pqxx::result&           rResult,
    const std::string&      rOsmPointTable,
    const std::string&      rTopoEdgeTable,
    const std::string&      rOsmEdgeTable,
    const std::string&      rSchemaName)
{
    rResult = rTrans.exec(
        "SELECT p.osm_id, "
        "       p.highway, "
        "       p.railway, "
        "       p.public_transport, "
        "       p.traffic_calming, "
        "       t.edge_id "
        "FROM  " + rOsmPointTable + " p, "
        "      " + rTopoEdgeTable + " t, "
        "      " + rOsmEdgeTable + " o, "
        "      " + rSchemaName + ".relation r "
        "WHERE  r.topogeo_id = (topo_geom).id "
        "AND    r.element_id = t.edge_id "
        "AND   (p.highway = 'bus_stop' OR "
        "       p.highway = 'crossing' OR "
        "       p.highway = 'give_way' OR"
        "       p.highway = 'mini_roundabout' OR"
        "       p.highway = 'stop' OR"
        "       p.highway = 'traffic_signals' OR"
        "       p.railway = 'level_crossing' OR"
        "       p.public_transport = 'stop_position' OR"
        "       p.traffic_calming = 'yes' OR"
        "       p.traffic_calming = 'bump' OR"
        "       p.traffic_calming = 'hump' OR"
```

```
        "       p.traffic_calming = 'table' OR"
        "       p.traffic_calming = 'cushion' OR"
        "       p.traffic_calming = 'rumble_strip' OR"
        "       p.traffic_calming = 'chicane' OR"
        "       p.traffic_calming = 'choker' OR"
        "       p.traffic_calming = 'island' "
        ")"
        "AND    ST_Intersects(p.way, t.geom) "
        "AND    o.highway IN " + OsmHighway::typesAsCommaSeparatedString()
    );
}

// static
void
CostQueries::addOtherCosts(
    const pqxx::result&     rResult,
    Topology&               rTopology,
    const Configuration&    rConfig)
{
    try
    {
        for(const pqxx::tuple& row : rResult)
        {
            // throw exception if no edgeId
            EdgeIdType edgeId =
                row[OtherCostResult::EDGE_ID].as<EdgeIdType>();

            Edge& edge = rTopology.getEdge(edgeId);

            std::string type_string = "highway=" +
                row[OtherCostResult::HIGHWAY].as<std::string>("");
            addOtherCostToEdge(edge, type_string, rConfig);

            type_string = "railway=" +
                row[OtherCostResult::RAILWAY].as<std::string>("");
            addOtherCostToEdge(edge, type_string, rConfig);

            type_string = "public_transport=" +
                row[OtherCostResult::PUBLIC_TRANSPORT].as<std::string>("");
            addOtherCostToEdge(edge, type_string, rConfig);

            type_string = "traffic_calming=" +
                row[OtherCostResult::TRAFFIC_CALMING].as<std::string>("");
            addOtherCostToEdge(edge, type_string, rConfig);
        }
    }
    catch (std::exception& e)
    {
        throw MapProviderException(
            std::string("CostQueries:addOtherCosts..: ") + e.what());
    }
}

// static
void
CostQueries::addOtherCostToEdge(
    Edge&                   rEdge,
```

```
    const std::string&      key,
    const Configuration&    rConfig)
{
    size_t eq_char = key.find('=');
    if((eq_char == std::string::npos) || (eq_char == key.length() - 1))
    {
        return;
    }

    Cost cost = rConfig.getCostConfig().otherEdgeCosts.getOtherCost(key);
    rEdge.edgeCost().addCost(EdgeCost::OTHER, cost);
}

// static
void
CostQueries::addBarrierCostToEdge(
    Edge&                   rEdge,
    OsmBarrier::BarrierType type,
    const Configuration&    rConfig)
{
    if(rConfig.getBarrierCostsRule().costsToPass(type))
    {
        Cost cost = rConfig.getBarrierCostsRule().getCost(type);
        rEdge.edgeCost().addCost(EdgeCost::BARRIER, cost);
    }
}

//============================== ACESS      ===================================
//============================== INQUIRY    ===================================
////////////////////////////// PROTECTED  ///////////////////////////////////
////////////////////////////// PRIVATE    ///////////////////////////////////
//static
std::string
CostQueries::startOfQuery(const std::string& rTopoEdgeTable)
{
    return (
        "SELECT     edge_id, "
        //-- osm data about original edge
        "           osm.* "
        "FROM      " + rTopoEdgeTable +
        " JOIN ( "
        "   SELECT  element_id "
    );
}

//static
std::string
CostQueries::queryColumns(const std::vector<std::string>& rCols)
{
    std::ostringstream oss;
    for(const std::string& col : rCols)
    {
        oss << ", " << col;
    }
    return oss.str();
}
```

```
//static
std::string
CostQueries::midOfQuery(
    const std::string& rSchemaName,
    const std::string& rOsmEdgeTable)
{
    return (
        "   FROM    " + rSchemaName + ".relation "
        "   JOIN    " + rOsmEdgeTable +
        "   ON      topogeo_id = (topo_geom).id "
        "   WHERE   highway in " + OsmHighway::typesAsCommaSeparatedString()
    );
}

//static
std::string
CostQueries::notNullColumns(const std::vector<std::string>& rCols)
{
    std::ostringstream oss;
    oss << " AND (";
    size_t i = 0;
    for(const std::string& col : rCols)
    {
        oss << col << " IS NOT NULL ";
        if(i < (rCols.size() - 1))
        {
            oss << " OR ";
        }
        ++i;
    }
    oss << ") ";
    return oss.str();
}

//static
std::string
CostQueries::endOfQuery()
{
    return (
        ") AS osm "
        "ON edge_id = element_id "
        "ORDER BY edge_id ASC;"
    );
}
```

D.11.6 LineGraphSaveQueries.h

```
/** Queries for saving the LineGraph to database
 *
 * #include "LineGraphSaveQueries.h"
 *
 * @author  Jonas Bergman
 */

#ifndef MAPPROVIDER_POSTGIS_LINEGRAPHSAVEQUERIES_H_
#define MAPPROVIDER_POSTGIS_LINEGRAPHSAVEQUERIES_H_

```

```
// SYSTEM INCLUDES
//
#include <string>
#include <sstream>
#include <vector>

// PROJECT INCLUDES
//
#include <boost/algorithm/string.hpp>
#include <pqxx/pqxx>

// LOCAL INCLUDES
//
#include "../../graph/GraphBuilder.h"
#include "../../graph/Topology.h"

/** Class for holding static queries for saving the line graph to database
 */
class LineGraphSaveQueries
{
public:
// TYPES

// LIFECYCLE
    LineGraphSaveQueries() = delete;
    LineGraphSaveQueries(const LineGraphSaveQueries& from) = delete;
    ~LineGraphSaveQueries() = default;

// OPERATORS
// OPERATIONS

    /** Create a new schema, dropping any existing with the same name first.
     * @param   rTrans      Transaction to perform the query in.
     * @param   rSchemaName Name of the schema to create.
     * @throw   pqxx::pqxx_exception
     */
    static void dropCreateSchema(
        pqxx::transaction_base& rTrans,
        const std::string&      rSchemaName);

    /** Create a new table for lines, dropping any existing with the same name.
     * @param   rTrans          Transaction to perform the query in.
     * @param   rTableName      Name of the table to create.
     * @throw   pqxx::pqxx_exception
     */
    static void dropCreateLineTable(
        pqxx::transaction_base& rTrans,
        const std::string&      rTableName);

    /** Create a new table for nodes, dropping any existing with the same name.
     * @param   rTrans          Transaction to perform the query in.
     * @param   rTableName      Name of the table to create.
     * @throw   pqxx::pqxx_exception
     */
    static void dropCreateNodeTable(
        pqxx::transaction_base& rTrans,
        const std::string&      rTableName);
```

```

    /** Insert a node into the database
     * @param   rTrans      Transaction to perform the query in.
     * @param   id          The id of the node's corresponding edge in the topology.
     * @param   rGeomString WKT (well-known text) representation of the node
     * @throw   pqxx::pqxx_exception
     */
    static void insertNode(
        pqxx::transaction_base& rTrans,
        const std::string&      rTableName,
        EdgeIdType              id,
        const std::string&      rGeomString);

    /** Insert a line into the database
     * @param   rTrans      Transaction to perform the query in.
     * @param   cost        The cost of traveling the edge
     * @param   rGeomString WKT (well-known text) representation of the line
     * @throw   pqxx::pqxx_exception
     */
    static void insertLine(
        pqxx::transaction_base& rTrans,
        const std::string&      rTableName,
        Cost                    cost,
        const std::string&      rGeomString);

// ACCESS
// INQUIRY

protected:
private:
};

#endif /* MAPPROVIDER_POSTGIS_LINEGRAPHSAVEQUERIES_H_ */
```

D.11.7 LineGraphSaveQueries.cc

```
/*
 * LineGraphSaveQueries.cc
 *
 * @author  Jonas Bergman
 */

#include "LineGraphSaveQueries.h"

/////////////////////////////// PUBLIC ///////////////////////////////////////

//============================= LIFECYCLE ====================================
//============================= OPERATORS ====================================
//============================= OPERATIONS ===================================
//static
void
LineGraphSaveQueries::dropCreateSchema(
    pqxx::transaction_base& rTrans,
    const std::string&      rSchemaName)
{
    rTrans.exec(
            "DROP SCHEMA IF EXISTS " + rSchemaName + " CASCADE; "
```

```
            "CREATE SCHEMA " + rSchemaName
    );
}

//static
void
LineGraphSaveQueries::dropCreateLineTable(
    pqxx::transaction_base& rTrans,
    const std::string&      rTableName)
{
    rTrans.exec(
            "DROP TABLE IF EXISTS " + rTableName + " CASCADE; "
            "CREATE TABLE " + rTableName + " ( "
            "    cost     double precision, "
            "    geom     geometry(LineString, 900913) "
            "); "
    );
}

//static
void
LineGraphSaveQueries::dropCreateNodeTable(
    pqxx::transaction_base& rTrans,
    const std::string&      rTableName)
{
    rTrans.exec(
            "DROP TABLE IF EXISTS " + rTableName + " CASCADE; "
            "CREATE TABLE " + rTableName + " ( "
            "    topo_id  bigint unique, "
            "    geom     geometry(Point, 900913) "
            "); "
    );
}

//static
void
LineGraphSaveQueries::insertNode(
    pqxx::transaction_base& rTrans,
    const std::string&      rTableName,
    EdgeIdType              id,
    const std::string&      rGeomString)
{

    rTrans.exec(
        "INSERT INTO " + rTableName +" (topo_id, geom) "
        "SELECT " + std::to_string(id) +
        ", ST_GeomFromText('" + rGeomString + "', 900913) "
        "WHERE NOT EXISTS ("
        "    SELECT topo_id FROM " + rTableName +
        "    WHERE topo_id = " + std::to_string(id) + " );"
    );
}

//static
void
LineGraphSaveQueries::insertLine(
    pqxx::transaction_base& rTrans,
```

```
    const std::string&      rTableName,
    Cost                    cost,
    const std::string&      rGeomString)
{
    rTrans.exec(
        "INSERT INTO " + rTableName +" (cost, geom) "
        "VALUES (" + std::to_string(cost) +
        ", ST_GeomFromText('" + rGeomString + "', 900913)); "
    );
}
//============================= ACESS      ===================================
//============================= INQUIRY    ===================================
/////////////////////////////// PROTECTED  ///////////////////////////////////
/////////////////////////////// PRIVATE    ///////////////////////////////////
```

D.11.8 RestrictionQueries.h

```
/**  Queries for PostGisProvider to find restrictions.
 *
 * #include "RestrictionQueries.h"
 *
 * @author  Jonas Bergman
 */

#ifndef MAPPROVIDER_POSTGIS_RESTRICTIONQUERIES_H_
#define MAPPROVIDER_POSTGIS_RESTRICTIONQUERIES_H_

// SYSTEM INCLUDES
//
#include <string>
#include <sstream>
#include <vector>

// PROJECT INCLUDES
//
#include <boost/algorithm/string.hpp>
#include <pqxx/pqxx>

// LOCAL INCLUDES
//
#include "CostQueries.h"
#include "../../osm/OsmHighway.h"
#include "../../osm/OsmTurningRestriction.h"
#include "../../graph/Edge.h"
#include "../../graph/EdgeRestriction.h"
#include "../../graph/Topology.h"
#include "../../graph/Vertex.h"
#include "../MapProviderException.h"

/** Class for holding static queries about restrictions,
 * needed by the PostGisProvider.
 */
class RestrictionQueries
{
public:
// TYPES

```

```
    /** Columns used in query for Vehicle Properties restrictions. */
    struct VehiclePropertiesRestrictions
    {
        enum Columns
        {
            EDGE_ID,
            ELEMENT_ID,
            MAXHEIGHT,
            MAXLENGTH,
            MAXWEIGHT,
            MAXWIDTH,
            MAXSPEED,
            MINSPEED,
        };
    };

    /** Columns used in query for Access restrictions. */
    struct AccessRestrictions
    {
        enum Columns
        {
            EDGE_ID,
            ELEMENT_ID,
            ACCESS,
            BARRIER,
            DISUSED,
            NOEXIT,
            MOTORCAR,
            GOODS,
            HGV,
            PSV,
            LHV,
            MOTOR_VEHICLE,
            VEHICLE,
        };
    };

    /** Columns used in query for Turning restrictions. */
    struct TurningRestrictions
    {
        enum Columns
        {
            FROM_OSM_ID,
            TO_OSM_ID,
            VIA_OSM,
            EDGE_IDS,
            RESTRICTION_TYPE
        };

        /** Results from queries are handled by these functions. */
        struct Results
        {
            /** Parse a row in the results from turning restrictions.
             * @param   rRow        The row with results.
             * @param   rTopology   The topology that needs to be queried.
             * @return  A turning restriction object.
             * @throw   pqxx::pqxx_exception
```

```
             * @throw   TopologyException
             * @throw   MapProviderException
             */
            static OsmTurningRestriction* parseTurningRestrictionResultRow(
                const pqxx::tuple&    rRow,
                Topology&             rTopology);


            /** Try to parse the column 'edge_ids' from the 'turning_restrictions'.
             * Split the string of edge ids and convert them to a vector of EdgeIds.
             * @param   rEdgeIds    A string like "{123, 456}". Gets trimmed of {}.
             * @return  A vector of the EdgeIds separately.
             * @throw   std::invalid_argument
             * @throw   std::out_of_range
             */
            static std::vector<EdgeIdType> parseEdgeIdsString(
                std::string& rEdgeIds);

            /** Find the Edge that matches the OsmId in turning restriction.
             * @param   osmId       The original edge osm id.
             * @param   rEdgeIds    The candidate edges that are near restriction.
             * @param   rTopology   The Topology to ask for edges.
             */
            static Edge&        findEdgeMatchingOsmId(
                OsmIdType                       osmId,
                const std::vector<EdgeIdType>&  rEdgeIds,
                Topology&                       rTopology);
        };
    };

    /** Columns used in query for EdgePoint restrictions. */
    struct EdgePointRestrictions
    {
        enum Columns
        {
            POINT_OSM_ID,
            BARRIER,
            ACCESS,
            GOODS,
            HGV,
            LHV,
            MOTORCAR,
            MOTOR_VEHICLE,
            PSV,
            VEHICLE,
            EDGE_ID,
        };
    };


// LIFECYCLE
    RestrictionQueries() = delete;
    RestrictionQueries(const RestrictionQueries& from) = delete;
    ~RestrictionQueries() = default;

// OPERATORS
// OPERATIONS
```

```
    /** Query for restrictions based on Vehicle Properties.
     * @param   rTrans          Transaction to perform query in.
     * @param   rResult         Store the result of query here.
     * @param   rTopoEdgeTable  Name of table with topology edges.
     * @param   rOsmEdgeTable   Name of table with OSM edges.
     * @param   rSchemaName     Name of the topology schema.
     * @throw   pqxx::pqxx_exception
     */
    static void         getVehiclePropertyEdgeRestrictions(
                            pqxx::transaction_base& rTrans,
                            pqxx::result&           rResult,
                            const std::string&      rTopoEdgeTable,
                            const std::string&      rOsmEdgeTable,
                            const std::string&      rSchemaName);

    /** Add the result of the query for vehicle properties to Edge's restrictions.
     * @param   rResult         The results of the query
     * @param   rTopology       Update affected edges in the topology.
     * @throw   MapProviderException
     */
    static void         addVehiclePropertyRestrictionsToEdge(
                            const pqxx::result&    rResult,
                            Topology&              rTopology);

    /** Query for general access restrictions.
     * @param   rTrans          Transaction to perform query in.
     * @param   rResult         Store the result of query here.
     * @param   rTopoEdgeTable  Name of table with topology edges.
     * @param   rOsmEdgeTable   Name of table with OSM edges.
     * @param   rSchemaName     Name of the topology schema.
     * @throw   pqxx::pqxx_exception
     */
    static void         getAccessRestrictions(
                            pqxx::transaction_base& rTrans,
                            pqxx::result&           rResult,
                            const std::string&      rTopoEdgeTable,
                            const std::string&      rOsmEdgeTable,
                            const std::string&      rSchemaName);

    /** Add the result of the query for Access to restrictions.
     * @param   rResult         The results of the query
     * @param   rTopology       Update affected edges in the topology.
     * @param   rConfig         Configuration
     * @throw   MapProviderException
     */
    static void         addAccessRestrictionsToEdge(
                            const pqxx::result&    rResult,
                            Topology&              rTopology,
                            const Configuration&   rConfig);

    /** Drop and create the table 'turning_restrictions'.
     * @param   rTrans          Transaction to perform query in.
     * @throw   pqxx::pqxx_exception
     */
    static void         dropCreateTurningRestrictionsTable(
                            pqxx::transaction_base& rTrans);

```

```
    /** Populate the table 'turning_restrictions'.
     * @param   rTrans          The transaction to execute within.
     * @param   rOsmEdgeTable   The name of the table with original osm edges.
     * @param   rTopoEdgeTable  The name of the table with topology edges.
     * @throw   pqxx::pqxx_exception
     */
    static void         identifyTurningRestrictions(
                            pqxx::transaction_base& rTrans,
                            const std::string&      rOsmEdgeTable,
                            const std::string&      rTopoEdgeTable);

    /** Get the restrictions from the 'turning_restrictions' table.
     * @param   rTrans          Transaction to perform query in.
     * @param   rResult         Store the result of query here.
     * @throw   pqxx::pqxx_exception
     */
    static void         getTurningRestrictions(
                            pqxx::transaction_base& rTrans,
                            pqxx::result&           rResult);

    /** Add the result of the query for Turning restrictions.
     * @param   rResult         The results of the query
     * @param   rTopology       Update affected edges in the topology.
     * @throw   MapProviderException
     */
    static void         addTurningRestrictionsToEdge(
                            const pqxx::result&    rResult,
                            Topology&              rTopology);

    /** Get the restrictions from the 'planet_osm_point' that relates to edges.
     * @param   rTrans          Transaction to perform query in.
     * @param   rResult         Store the result of query here.
     * @param   rOsmPointTabl   The name of the table with original osm points.
     * @param   rTopoEdgeTable  The name of the table with topology edges.
     * @param   rOsmEdgeTable   The name of the table with OSM edges.
     * @param   rSchemaName     The name of the schema with topology info.
     * @throw   pqxx::pqxx_exception
     */
    static void         getEdgePointRestrictions(
                            pqxx::transaction_base& rTrans,
                            pqxx::result&           rResult,
                            const std::string&      rOsmPointTable,
                            const std::string&      rTopoEdgeTable,
                            const std::string&      rOsmEdgeTable,
                            const std::string&      rSchemaName);

    /** Add the result of the query for Point restrictions on Edges .
     * @param   rResult         The results of the query
     * @param   rTopology       Update affected edges in the topology.
     * @param   rConfig         Configuration
     * @throw   MapProviderException
     */
    static void         addPointRestrictionsToEdge(
                            const pqxx::result&    rResult,
                            Topology&              rTopology,
                            const Configuration&   rConfig);
// ACCESS
```

```
// INQUIRY

protected:
private:
    /** SELECT FROM JOIN */
    static std::string  startOfQuery(const std::string& rTopoEdgeTable);

    /** Which columns to pick */
    static std::string  queryColumns(const std::vector<std::string>& rCols);

    /** FROM JOIN ON WHERE */
    static std::string  midOfQuery(
        const std::string& rSchemaName,
        const std::string& rOsmEdgeTable);

    /** Make sure only to pick rows with content in some column. */
    static std::string  notNullColumns(const std::vector<std::string>& rCols);

    /** AS ON ORDER BY */
    static std::string  endOfQuery();
};

#endif /* MAPPROVIDER_POSTGIS_RESTRICTIONQUERIES_H_ */
```

D.11.9 RestrictionQueries.cc

```
/*
 * RestrictionQueries.cc
 *
 * @author  Jonas Bergman
 */

#include "RestrictionQueries.h"  // class implemented

// Result ------------------------------------------------------------------
//static
OsmTurningRestriction*
RestrictionQueries::TurningRestrictions::
    Results::parseTurningRestrictionResultRow(
    const pqxx::tuple&    rRow,
    Topology&             rTopology)
{
    OsmIdType fromOsmId =
        rRow[RestrictionQueries::TurningRestrictions::FROM_OSM_ID].as<OsmIdType>();
    OsmIdType toOsmId =
        rRow[RestrictionQueries::TurningRestrictions::TO_OSM_ID].as<OsmIdType>();
    std::string typeString =
        rRow[RestrictionQueries::TurningRestrictions::RESTRICTION_TYPE].as<std::string>();
    OsmTurningRestriction::TurningRestrictionType type =
        OsmTurningRestriction::parseString(typeString);
    std::string edgeIdsString =
        rRow[RestrictionQueries::TurningRestrictions::EDGE_IDS].as<std::string>();
    std::string viaOsmIdsString =
        rRow[RestrictionQueries::TurningRestrictions::VIA_OSM].as<std::string>("");

    std::vector<EdgeIdType> edgeIds = parseEdgeIdsString(edgeIdsString);

```

```
    Edge& fromEdge = findEdgeMatchingOsmId(fromOsmId, edgeIds, rTopology);
    Edge& toEdge   = findEdgeMatchingOsmId(toOsmId, edgeIds, rTopology);

    OsmTurningRestriction* p_restriction {nullptr};

    // VIA WAY
    if(fromEdge.targetId() != toEdge.sourceId())
    {
        p_restriction = new OsmTurningRestriction(
            type,
            fromEdge.id(),
            viaOsmIdsString,
            toEdge.id());
    }
    // VIA NODE
    else
    {
        VertexIdType vertexId = fromEdge.targetId();
        p_restriction = new OsmTurningRestriction(
            type,
            fromEdge.id(),
            vertexId,
            toEdge.id());
    }
    return p_restriction;
}

//static
std::vector<EdgeIdType>
RestrictionQueries::TurningRestrictions::Results::parseEdgeIdsString(
    std::string& rEdgeIds)
{
    boost::trim_if(rEdgeIds, boost::is_any_of("{}"));
    std::vector<std::string> idStrings;
    boost::split(idStrings, rEdgeIds, boost::is_any_of(","));

    std::vector<EdgeIdType> edgeIds;
    for(const std::string& idStr : idStrings)
    {
        edgeIds.push_back(Edge::parse(idStr));
    }
    return edgeIds;
}

//static
Edge&
RestrictionQueries::TurningRestrictions::Results::findEdgeMatchingOsmId(
    OsmIdType                       osmId,
    const std::vector<EdgeIdType>&  rEdgeIds,
    Topology&                       rTopology)
{
    for(EdgeIdType id : rEdgeIds)
    {
        Edge& edge = rTopology.getEdge(id);
        if(edge.osmId() == osmId)
        {
            return edge;
```

```
        }
    }
    throw MapProviderException(
        "PostGisRestrictionQueries:Result:findEdgeMatchingOsmId: "
        "No edges matching osm_id: " + std::to_string(osmId));
}

/////////////////////////////// PUBLIC ///////////////////////////////////////

//============================= LIFECYCLE ====================================
//============================= OPERATORS ====================================
//============================= OPERATIONS ===================================
//static
void
RestrictionQueries::getVehiclePropertyEdgeRestrictions(
    pqxx::transaction_base& rTrans,
    pqxx::result&           rResult,
    const std::string&      rTopoEdgeTable,
    const std::string&      rOsmEdgeTable,
    const std::string&      rSchemaName)
{
    std::vector<std::string> columns {
        "maxheight",
        "maxlength",
        "maxweight",
        "maxwidth",
        "maxspeed",
        "minspeed"
    };

    rResult = rTrans.exec(
        startOfQuery(rTopoEdgeTable) +
        queryColumns(columns) +
        midOfQuery(rSchemaName, rOsmEdgeTable) +
        notNullColumns(columns) +
        endOfQuery()
    );
}

// static
void
RestrictionQueries::addVehiclePropertyRestrictionsToEdge(
    const pqxx::result& rResult,
    Topology&           rTopology)
{
    try
    {
        for(const pqxx::tuple& row : rResult)
        {
            // throw exception if no edgeId
            EdgeIdType edgeId =
                row[VehiclePropertiesRestrictions::EDGE_ID].as<EdgeIdType>();

            Edge& edge = rTopology.getEdge(edgeId);
            EdgeRestriction& r_restrictions = edge.restrictions();

            EdgeRestriction::VehicleProperties* p_vp =
```

```
                new EdgeRestriction::VehicleProperties();

            p_vp->maxHeight =
                row[VehiclePropertiesRestrictions::MAXHEIGHT].as<double>
                (EdgeRestriction::VehicleProperties::DEFAULT_DIMENSION_MAX);
            p_vp->maxLength =
                row[VehiclePropertiesRestrictions::MAXLENGTH].as<double>
                (EdgeRestriction::VehicleProperties::DEFAULT_DIMENSION_MAX);
            p_vp->maxWeight =
                row[VehiclePropertiesRestrictions::MAXWEIGHT].as<double>
                (EdgeRestriction::VehicleProperties::DEFAULT_DIMENSION_MAX);
            p_vp->maxWidth =
                row[VehiclePropertiesRestrictions::MAXWIDTH].as<double>
                (EdgeRestriction::VehicleProperties::DEFAULT_DIMENSION_MAX);
            p_vp->maxSpeed =
                row[VehiclePropertiesRestrictions::MAXSPEED].as<unsigned>
                (EdgeRestriction::VehicleProperties::DEFAULT_SPEED_MAX);
            p_vp->minSpeed =
                row[VehiclePropertiesRestrictions::MINSPEED].as<unsigned>
                (EdgeRestriction::VehicleProperties::DEFAULT_SPEED_MIN);

            r_restrictions.setVehiclePropertyRestriction(p_vp);
        }
    }
    catch (std::exception& e)
    {
        throw MapProviderException(
            std::string("RestrictionQueries:addVehicleProp..ToEdge..: ")
                        + e.what());
    }
}

//static
void
RestrictionQueries::getAccessRestrictions(
    pqxx::transaction_base& rTrans,
    pqxx::result&           rResult,
    const std::string&      rTopoEdgeTable,
    const std::string&      rOsmEdgeTable,
    const std::string&      rSchemaName)
{
    std::vector<std::string> columns {
        "access",
        "barrier",
        "disused",
        "noexit",
        "motorcar",
        "goods",
        "hgv",
        "psv",
        "lhv",
        "motor_vehicle",
        "vehicle"
    };

    rResult = rTrans.exec(
        startOfQuery(rTopoEdgeTable) +
```

```
        queryColumns(columns) +
        midOfQuery(rSchemaName, rOsmEdgeTable) +
        notNullColumns(columns) +
        endOfQuery()
    );
}

// static
void
RestrictionQueries::addAccessRestrictionsToEdge(
    const pqxx::result&     rResult,
    Topology&               rTopology,
    const Configuration&    rConfig)
{
    try
    {
        for(const pqxx::tuple& row : rResult)
        {
            // throw exception if no edgeId
            EdgeIdType edgeId =
                row[AccessRestrictions::EDGE_ID].as<EdgeIdType>();

            Edge& edge = rTopology.getEdge(edgeId);
            EdgeRestriction& r_restrictions = edge.restrictions();

            std::string colString;
            colString = row[AccessRestrictions::ACCESS].as<std::string>("");
            if(colString != "")
            {
                OsmAccess::AccessType type = OsmAccess::parseString(colString);
                r_restrictions.setGeneralAccessRestriction(type);
            }

            colString = row[AccessRestrictions::MOTORCAR].as<std::string>("");
            if(colString != "")
            {
                OsmAccess::AccessType type = OsmAccess::parseString(colString);
                r_restrictions.addVehicleTypeAccessRestriction(
                    OsmVehicle::MOTORCAR,
                    type
                );
            }

            colString = row[AccessRestrictions::GOODS].as<std::string>("");
            if(colString != "")
            {
                OsmAccess::AccessType type = OsmAccess::parseString(colString);
                r_restrictions.addVehicleTypeAccessRestriction(
                    OsmVehicle::GOODS,
                    type
                );
            }

            colString = row[AccessRestrictions::HGV].as<std::string>("");
            if(colString != "")
            {
                OsmAccess::AccessType type = OsmAccess::parseString(colString);
```

```
                r_restrictions.addVehicleTypeAccessRestriction(
                    OsmVehicle::HGV,
                    type
                );
            }

            colString = row[AccessRestrictions::PSV].as<std::string>("");
            if(colString != "")
            {
                OsmAccess::AccessType type = OsmAccess::parseString(colString);
                r_restrictions.addVehicleTypeAccessRestriction(
                    OsmVehicle::PSV,
                    type
                );
            }

            colString = row[AccessRestrictions::LHV].as<std::string>("");
            if(colString != "")
            {
                OsmAccess::AccessType type = OsmAccess::parseString(colString);
                r_restrictions.addVehicleTypeAccessRestriction(
                    OsmVehicle::LHV,
                    type
                );
            }

            colString = row[AccessRestrictions::MOTOR_VEHICLE].as<std::string>("");
            if(colString != "")
            {
                OsmAccess::AccessType type = OsmAccess::parseString(colString);
                r_restrictions.addVehicleTypeAccessRestriction(
                    OsmVehicle::MOTOR_VEHICLE,
                    type
                );
            }

            colString = row[AccessRestrictions::VEHICLE].as<std::string>("");
            if(colString != "")
            {
                OsmAccess::AccessType type = OsmAccess::parseString(colString);
                r_restrictions.addVehicleTypeAccessRestriction(
                    OsmVehicle::VEHICLE,
                    type
                );
            }

            colString = row[AccessRestrictions::BARRIER].as<std::string>("");
            if(colString != "")
            {
                OsmBarrier::BarrierType type = OsmBarrier::parseString(colString);
                r_restrictions.setBarrierRestriction(type);
                CostQueries::addBarrierCostToEdge(edge, type, rConfig);
            }

            colString = row[AccessRestrictions::DISUSED].as<std::string>("");
            if(colString == "yes")
            {
```

```
                r_restrictions.setDisusedRestriction();
            }

            colString = row[AccessRestrictions::NOEXIT].as<std::string>("");
            if(colString == "yes")
            {
                r_restrictions.setNoExitRestriction();
            }
        }
    }
    catch (std::exception& e)
    {
        throw MapProviderException(
            std::string("RestrictionQueries:addAccessResultToEdge..: ") + e.what());
    }
}

//static
void
RestrictionQueries::dropCreateTurningRestrictionsTable(
    pqxx::transaction_base& rTrans)
{
    rTrans.exec(
        "DROP TABLE IF EXISTS turning_restrictions; "
        "CREATE TABLE turning_restrictions( "
        "   from_osm_id       bigint, "
        "   to_osm_id         bigint, "
        "   via_osm           varchar, "
        "   edge_ids          integer[], "
        "   restriction_type  varchar)"
    );
}

//static
void
RestrictionQueries::identifyTurningRestrictions(
    pqxx::transaction_base& rTrans,
    const std::string&      rOsmEdgeTable,
    const std::string&      rTopoEdgeTable)
{
    rTrans.exec(
        "SELECT * FROM find_osm_turning_restrictions('"
        + rOsmEdgeTable + "', '" + rTopoEdgeTable + "')"
    );
}

//static
void
RestrictionQueries::getTurningRestrictions(
    pqxx::transaction_base& rTrans,
    pqxx::result&           rResult)
{
    rResult = rTrans.exec(
        "SELECT * FROM turning_restrictions"
    );
}

```

```
// static
void
RestrictionQueries::addTurningRestrictionsToEdge(
    const pqxx::result&     rResult,
    Topology&               rTopology)
{
    try
    {
        for(const pqxx::tuple& row : rResult)
        {
            OsmTurningRestriction* p_turn =
                TurningRestrictions::Results::
                    parseTurningRestrictionResultRow(row, rTopology);

            // mark edge as having a restriction
            Edge& edge = rTopology.getEdge(p_turn->fromEdgeId());
            EdgeRestriction& r_restrictions = edge.restrictions();
            r_restrictions.addTurningRestriction(p_turn);

            // explicit mark "VIA WAY"
            if(p_turn->viaType() == OsmTurningRestriction::VIA_WAY)
            {
                r_restrictions.setViaWayRestriction();
            }
        }
    }
    catch (std::exception& e)
    {
        throw MapProviderException(
            std::string("RestrictionQueries:addTurningResultToEdge..: ") + e.what());
    }
}

//static
void
RestrictionQueries::getEdgePointRestrictions(
    pqxx::transaction_base& rTrans,
    pqxx::result&           rResult,
    const std::string&      rOsmPointTable,
    const std::string&      rTopoEdgeTable,
    const std::string&      rOsmEdgeTable,
    const std::string&      rSchemaName)
{
    rResult = rTrans.exec(
        "SELECT p.osm_id, "
        "       p.barrier, "
        "       p.access, "
        "       p.goods, "
        "       p.hgv, "
        "       p.lhv, "
        "       p.motorcar, "
        "       p.motor_vehicle, "
        "       p.psv, "
        "       p.vehicle, "
        "       t.edge_id "
        "FROM  " + rOsmPointTable + " p, "
        "      " + rTopoEdgeTable + " t, "
```

```
        "      " + rOsmEdgeTable + " o, "
        "      " + rSchemaName + ".relation r "
        "WHERE  r.topogeo_id = (topo_geom).id "
        "AND    r.element_id = t.edge_id "
        "AND    p.barrier IS NOT NULL "
        "AND    ST_Intersects(p.way, t.geom) "
        "AND    o.highway IN " + OsmHighway::typesAsCommaSeparatedString()
    );
}
// static
void
RestrictionQueries::addPointRestrictionsToEdge(
    const pqxx::result&     rResult,
    Topology&               rTopology,
    const Configuration&   rConfig)
{
    try
    {
        for(const pqxx::tuple& row : rResult)
        {
            // throw exception if no edgeId
            EdgeIdType edgeId =
                row[EdgePointRestrictions::EDGE_ID].as<EdgeIdType>();

            Edge& edge = rTopology.getEdge(edgeId);
            EdgeRestriction& r_restrictions = edge.restrictions();

            std::string barrierTypeString =
                row[EdgePointRestrictions::BARRIER].as<std::string>();
            OsmBarrier::BarrierType barrierType =
                OsmBarrier::parseString(barrierTypeString);
            r_restrictions.setBarrierRestriction(barrierType);
            CostQueries::addBarrierCostToEdge(edge, barrierType, rConfig);

            std::string colString;
            colString = row[EdgePointRestrictions::ACCESS].as<std::string>("");
            if(colString != "")
            {
                OsmAccess::AccessType type = OsmAccess::parseString(colString);
                r_restrictions.setGeneralAccessRestriction(type);
            }

            colString = row[EdgePointRestrictions::GOODS].as<std::string>("");
            if(colString != "")
            {
                OsmAccess::AccessType type = OsmAccess::parseString(colString);
                r_restrictions.addVehicleTypeAccessRestriction(
                    OsmVehicle::GOODS,
                    type);
            }

            colString = row[EdgePointRestrictions::HGV].as<std::string>("");
            if(colString != "")
            {
                OsmAccess::AccessType type = OsmAccess::parseString(colString);
                r_restrictions.addVehicleTypeAccessRestriction(
                    OsmVehicle::HGV,
```

```
                    type);
            }

            colString = row[EdgePointRestrictions::LHV].as<std::string>("");
            if(colString != "")
            {
                OsmAccess::AccessType type = OsmAccess::parseString(colString);
                r_restrictions.addVehicleTypeAccessRestriction(
                    OsmVehicle::LHV,
                    type);
            }

            colString = row[EdgePointRestrictions::MOTORCAR].as<std::string>("");
            if(colString != "")
            {
                OsmAccess::AccessType type = OsmAccess::parseString(colString);
                r_restrictions.addVehicleTypeAccessRestriction(
                    OsmVehicle::MOTORCAR,
                    type);
            }

            colString = row[EdgePointRestrictions::MOTOR_VEHICLE].as<std::string>("");
            if(colString != "")
            {
                OsmAccess::AccessType type = OsmAccess::parseString(colString);
                r_restrictions.addVehicleTypeAccessRestriction(
                    OsmVehicle::MOTOR_VEHICLE,
                    type);
            }

            colString = row[EdgePointRestrictions::PSV].as<std::string>("");
            if(colString != "")
            {
                OsmAccess::AccessType type = OsmAccess::parseString(colString);
                r_restrictions.addVehicleTypeAccessRestriction(
                    OsmVehicle::PSV,
                    type);
            }

            colString = row[EdgePointRestrictions::VEHICLE].as<std::string>("");
            if(colString != "")
            {
                OsmAccess::AccessType type = OsmAccess::parseString(colString);
                r_restrictions.addVehicleTypeAccessRestriction(
                    OsmVehicle::VEHICLE,
                    type);
            }
        }
    }
    catch (std::exception& e)
    {
        throw MapProviderException(
            std::string("RestrictionQueries:addPointResultToEdge..: ") + e.what());
    }
}
//============================= ACESS      ===================================
//============================= INQUIRY    ===================================
```

```
/////////////////////////////// PROTECTED  ///////////////////////////////////
/////////////////////////////// PRIVATE    ///////////////////////////////////
//static
std::string
RestrictionQueries::startOfQuery(const std::string& rTopoEdgeTable)
{
    return (
        "SELECT     edge_id, "
        //-- osm data about original edge
        "           osm.* "
        "FROM      " + rTopoEdgeTable +
        " JOIN ( "
        "   SELECT  element_id "
    );
}

//static
std::string
RestrictionQueries::queryColumns(const std::vector<std::string>& rCols)
{
    std::ostringstream oss;
    for(const std::string& col : rCols)
    {
        oss << ", " << col;
    }
    return oss.str();
}

//static
std::string
RestrictionQueries::midOfQuery(
    const std::string& rSchemaName,
    const std::string& rOsmEdgeTable)
{
    return (
        "   FROM    " + rSchemaName + ".relation "
        "   JOIN    " + rOsmEdgeTable +
        "   ON      topogeo_id = (topo_geom).id "
        "   WHERE   highway in " + OsmHighway::typesAsCommaSeparatedString()
    );
}

//static
std::string
RestrictionQueries::notNullColumns(const std::vector<std::string>& rCols)
{
    std::ostringstream oss;
    oss << " AND (";
    size_t i = 0;
    for(const std::string& col : rCols)
    {
        oss << col << " IS NOT NULL ";
        if(i < (rCols.size() - 1))
        {
            oss << " OR ";
        }
        ++i;
```

```
    }
    oss << ") ";
    return oss.str();
}

//static
std::string
RestrictionQueries::endOfQuery()
{
    return (
        ") AS osm "
        "ON edge_id = element_id "
        "ORDER BY edge_id ASC;"
    );
}
```

D.11.10 TopologyQueries.h

```
/**  A class for holding static queries about the topology
 * for the PostGisProvider.
 *
 * #include "TopologyQueries.h"
 *
 * @author  Jonas Bergman
 */

#ifndef MAPPROVIDER_POSTGIS_TOPOLOGYQUERIES_H_
#define MAPPROVIDER_POSTGIS_TOPOLOGYQUERIES_H_

// SYSTEM INCLUDES
//
#include <string>

// PROJECT INCLUDES
//
#include <pqxx/pqxx>

// LOCAL INCLUDES
//
#include "../../graph/Topology.h"
#include "../../graph/TopologyException.h"
#include "../MapProviderException.h"

// FORWARD REFERENCES
//

/** This class holds static queries about the topology to be used by the
 * PostGisProvider. It also holds types for handling the results.
 */
class TopologyQueries
{
public:
// TYPES

    /** Columns used in queries for Vertices. */
    struct VertexResult
    {
```

```
        enum Columns
        {
            NODE_ID,
            X,
            Y
        };
    };

    /** Columns used in queries for Edges. */
    struct EdgeResult
    {
        enum Columns
        {
            EDGE_ID,
            START_NODE,
            END_NODE,
            EDGE_LENGTH,
            CENTER_X,
            CENTER_Y,
            SOURCE_BEARING,
            TARGET_BEARING,
            OSM_ID,
            ELEMENT_ID,     // NOT USED: same as EDGE_ID_COL

            // road data
            HIGHWAY,
            JUNCTION,
            LANES,
            ONEWAY,

            // access
            ACCESS,
            MOTORCAR,
            GOODS,
            HGV,
            PSV,
            LHV,
            MOTOR_VEHICLE,
            VEHICLE,
        };
    };



// LIFECYCLE
    /** Constructor. */
    TopologyQueries() = delete;
    /** Copy constructor. */
    TopologyQueries(const TopologyQueries& from) = delete;

// OPERATORS
// OPERATIONS
    /** Fetch the vertices for the topology.
     * @param   rTrans          Transaction to perform query in.
     * @param   rResult         Store the result of query here.
     * @param   rVertexTable    Name of table to fetch topology vertices from.
     */
```

```
    static void     getTopologyVertices(
                        pqxx::transaction_base& rTrans,
                        pqxx::result&           rResult,
                        const std::string&      rVertexTable);

    /** Add vertices to topology.
     * @throws  TopologyException
     */
    static void     addVertexResultToTopology(
                        const pqxx::result&     rResult,
                        Topology&               rTopology);

    /** Fetch the edges for the topology.
     * @param   rTrans          Transaction to perform query in.
     * @param   rResult         Store the result of query here.
     * @param   rTopoEdgeTable  Name of table to fetch topology edges from.
     * @param   rSchemaName     Name of topology schema.
     * @param   rOsmEdgeTable   Name of table with original OSM edge data.
     */
    static void     getTopologyEdges(
                        pqxx::transaction_base& rTrans,
                        pqxx::result&           rResult,
                        const std::string&      rTopoEdgeTable,
                        const std::string&      rSchemaName,
                        const std::string&      rOsmEdgeTable);

    /** Add edges to topology.
     * @param   rEdgeResult    Result of db query for edges.
     * @param   rTopology      Topology to fill with edges.
     * @throws  TopologyException
     */
    static void     addEdgeResultToTopology(
                        const pqxx::result&     rResult,
                        Topology&               rTopology);

    /** Helper to add basic data from db to Edge.
     * @param   rRow        Row with data for an Edge.
     * @param   rTopology   Topology to add edge to.
     * @return  Reference to the newly added Edge.
     * @throws  TopologyException
     */
    static Edge&    addBasicResultToEdge(
                        const pqxx::tuple&  rRow,
                        Topology&           rTopology);

    /** Add geometric result from query to an Edge.
     * @param   rEdge   Reference to Edge to set Geom data on.
     * @param   rRow    Reference to Row with Geom data in it.
     */
    static void     addGeomDataResultToEdge(
                        Edge&               rEdge,
                        const pqxx::tuple&  rRow);

    /** Add road related result from query to an Edge.
     * @param   rEdge   Reference to Edge to set road data on.
     * @param   rRow    Reference to Row with road data in it.
     */
```

```
    static void     addRoadDataResultToEdge(
                        Edge&               rEdge,
                        const pqxx::tuple&  rRow);

    /** Extract highway type from database result and store in RoadData.
     * @param   rRoadData   The RoadData to store in.
     * @param   rRow    Reference to Row with road data in it.
     * @throw   MapProviderException
     */
    static void     addHighwayTypeToEdgeRoadData(
                        Edge::RoadData&     rRoadData,
                        const pqxx::tuple&  rRow);

    /** Make sure the 'postgis_topology' extension is installed.
     * @param   rTrans          Transaction to perform query in.
     */
    static void     installPostgisTopology(pqxx::transaction_base& rTrans);

    /** Set schema search path for queries.
     * @param   rTrans          Transaction to perform query in.
     */
    static void     setSearchPath(pqxx::transaction_base& rTrans);

    /** Create the temporary table for topologies.
     * @param   rTrans          Transaction to perform query in.
     * @param   rTableName      Name of the temporary topology table.
     */
    static void     createTemporaryTable(
                        pqxx::transaction_base& rTrans,
                        const std::string&      rTableName);

    /** Create a schema for the temporary postgis topology.
     * @param   rTrans          Transaction to perform query in.
     * @param   rSchemaName     Name of the temporary schema.
     * @param   srid            The projection to use.
     */
    static void     createTemporarySchema(
                        pqxx::transaction_base& rTrans,
                        const std::string&      rSchemaName,
                        int                     srid);

    /** Add a column for geometry in the table with Osm Edges.
     * @param   rTrans          Transaction to perform query in.
     * @param   rSchemaName     Name of the temporary schema.
     * @param   rOsmeEdgeTable  Name of the table with OSM edges.
     */
    static void     addTopoGeometryColumn(
                        pqxx::transaction_base& rTrans,
                        const std::string&      rSchemaName,
                        const std::string&      rOsmEdgeTable);

    /** Fill geometry in the table with Osm Edges, using a tolerance for
     * merging nodes near one another.
     * @param   rTrans          Transaction to perform query in.
     * @param   rSchemaName     Name of the temporary schema.
     * @param   rOsmeEdgeTable  Name of the table with OSM edges.
     * @param   tolerance       Tolerance in unit of projection.
```

```
     */
    static void     fillTopoGeometryColumn(
                        pqxx::transaction_base& rTrans,
                        const std::string&      rSchemaName,
                        const std::string&      rOsmEdgeTable,
                        double                  tolerance);

    /** Drop the temporary table for topologies.
     * @param   rTrans          Transaction to perform query in.
     * @param   rTableName      Name of the temporary topology table.
     */
    static void     dropTemporaryTable(
                        pqxx::transaction_base& rTrans,
                        const std::string&      rTableName);

    /** Drop the temporary schema for topologies.
     * @param   rTrans          Transaction to perform query in.
     * @param   rSchemaName     Name of the temporary topology scheam.
     */
    static void     dropTemporarySchema(
                        pqxx::transaction_base& rTrans,
                        const std::string&      rTableName);

    /** Clean up in records for postgis topologies.
     * @param   rTrans          Transaction to perform query in.
     * @param   rTableName      Name of the temporary topology table.
     */
    static void     deleteTemporaryLayerRecord(
                        pqxx::transaction_base& rTrans,
                        const std::string&      rTableName);

    /** Clean up in records for postgis topologies.
     * @param   rTrans          Transaction to perform query in.
     * @param   rTableName      Name of the temporary topology table.
     */
    static void     deleteTemporaryTopoRecord(
                        pqxx::transaction_base& rTrans,
                        const std::string&      rSchemaName);

// ACCESS
// INQUIRY

protected:
private:
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* MAPPROVIDER_POSTGIS_TOPOLOGYQUERIES_H_ */
```

D.11.11 TopologyQueries.cc

```
/*
 * TopologyQueries.cc
 *
 * @author  Jonas Bergman
 */

#include "TopologyQueries.h"  // class implemented

#include "../../osm/OsmHighway.h"


/////////////////////////////// PUBLIC ///////////////////////////////////////

//============================= LIFECYCLE ====================================
//============================= OPERATORS ====================================
//============================= OPERATIONS ===================================
//static
void
TopologyQueries::getTopologyVertices(
    pqxx::transaction_base& rTrans,
    pqxx::result&           rResult,
    const std::string&      rVertexTable)
{
        rResult = rTrans.exec(
                "SELECT node_id, ST_X(geom) AS x, ST_Y(geom) AS y "
                " FROM " + rVertexTable +
                " ORDER BY node_id ASC;"
        );
}

// static
void
TopologyQueries::addVertexResultToTopology(
    const pqxx::result& rResult,
    Topology&           rTopology)
{
    for(size_t row = 0; row < rResult.size(); ++row)
    {
        VertexIdType id(rResult[row][VertexResult::NODE_ID].as<int>());
        Point point(rResult[row][VertexResult::X].as<double>(),
                    rResult[row][VertexResult::Y].as<double>());
        rTopology.addVertex(id, point);
    }
}

//static
void
TopologyQueries::getTopologyEdges(
    pqxx::transaction_base& rTrans,
    pqxx::result&           rResult,
    const std::string&      rTopoEdgeTable,
    const std::string&      rSchemaName,
    const std::string&      rOsmEdgeTable)
{
    std::string sql(
        "SELECT     edge_id, "
```

```
        "           start_node, "
        "           end_node, "
        //-- geom data about edge
        "           ST_Length(geom) AS edge_length, "
        "           ST_X(ST_LineInterpolatePoint(geom, 0.5)) AS center_x, "
        "           ST_Y(ST_LineInterpolatePoint(geom, 0.5)) AS center_y, "
        "           (ST_Azimuth("
        "               ST_PointN(geom,1), "
        "               ST_PointN(geom,2))/(2*pi())*360)::int "
        "               AS source_bearing, "
        "           (ST_Azimuth("
        "               ST_PointN(geom,ST_NPoints(geom)-1), "
        "               ST_PointN(geom,ST_NPoints(geom)))/(2*pi())*360)::int "
        "               AS target_bearing, "
        //-- osm data about original edge
        "           osm.* "
        "FROM      " + rTopoEdgeTable +
        " JOIN ( "
        "   SELECT  osm_id, element_id "
        //-- road data
        "           , highway "
        "           , junction "
        "           , lanes "
        "           , oneway "
        "   FROM    " + rSchemaName + ".relation "
        "   JOIN    " + rOsmEdgeTable+
        "   ON      topogeo_id = (topo_geom).id "
        "   WHERE   highway in " + OsmHighway::typesAsCommaSeparatedString() +
        ") AS osm "
        "ON edge_id = element_id "
        "ORDER BY edge_id ASC;"
    );
    rResult = rTrans.exec(sql);
}

// static
void
TopologyQueries::addEdgeResultToTopology(
    const pqxx::result& rResult,
    Topology&           rTopology)
{
    for(const pqxx::tuple& row : rResult)
    {
        Edge& edge = addBasicResultToEdge(row, rTopology);
        addGeomDataResultToEdge(edge, row);
        addRoadDataResultToEdge(edge, row);
    }
}

// static
Edge&
TopologyQueries::addBasicResultToEdge(
    const pqxx::tuple&  rRow,
    Topology&           rTopology)
{
    EdgeIdType
        edge_id(rRow[EdgeResult::EDGE_ID].as<EdgeIdType>(Edge::MAX_ID));
```

```
    OsmIdType
        osm_id(rRow[EdgeResult::OSM_ID].as<OsmIdType>(Osm::MAX_ID));
    VertexIdType
        source_id(rRow[EdgeResult::START_NODE].as<int>(Vertex::MAX_ID));
    VertexIdType
        target_id(rRow[EdgeResult::END_NODE].as<int>(Vertex::MAX_ID));

    Edge& edge = rTopology.addEdge(edge_id, osm_id, source_id, target_id);

    return edge;
}

// static
void
TopologyQueries::addGeomDataResultToEdge(Edge& rEdge, const pqxx::tuple& rRow)
{
    Edge::GeomData gd(
            rRow[EdgeResult::EDGE_LENGTH].as<double>(0),
            Point(rRow[EdgeResult::CENTER_X].as<double>(0),
                  rRow[EdgeResult::CENTER_Y].as<double>(0)),
            rRow[EdgeResult::SOURCE_BEARING].as<int>(0),
            rRow[EdgeResult::TARGET_BEARING].as<int>(0));
    rEdge.setGeomData(gd);
}

// static
void
TopologyQueries::addRoadDataResultToEdge(Edge& rEdge, const pqxx::tuple& rRow)
{
    Edge::RoadData rd;
    std::string
        onewayStr(rRow[EdgeResult::ONEWAY].as<std::string>("no"));

    if(rRow[EdgeResult::JUNCTION].as<std::string>("") ==
        OsmHighway::JUNCTION_ROUNDABOUT)
    {
        onewayStr = "yes";
    }
    if(onewayStr == "yes")
    {
        rd.direction = Edge::DirectionType::FROM_TO;
    }
    else if(onewayStr == "-1")
    {
        rd.direction = Edge::DirectionType::TO_FROM;
    }

    rd.nrLanes = rRow[EdgeResult::LANES].as<size_t>(1);

    addHighwayTypeToEdgeRoadData(rd, rRow);

    rEdge.setRoadData(rd);
}

// static
void
TopologyQueries::addHighwayTypeToEdgeRoadData(Edge::RoadData& rRoadData,
```

```
                                             const pqxx::tuple& rRow)
{
    std::string roadTypeStr( rRow[EdgeResult::HIGHWAY].as<std::string>("road"));
    try
    {
        rRoadData.roadType = OsmHighway::parseString(roadTypeStr);
    }
    catch (OsmException& oe)
    {
        throw MapProviderException(
            std::string("TopologyQueries:addHighwayTypeToEdgeRoadData:")
            + oe.what());
    }
}

//static
void
TopologyQueries::installPostgisTopology(pqxx::transaction_base& rTrans)
{
    rTrans.exec(
            "CREATE EXTENSION IF NOT EXISTS postgis_topology"
    );
}

//static
void
TopologyQueries::setSearchPath(pqxx::transaction_base& rTrans)
{
    rTrans.exec(
            "SET search_path = topology, public"
    );
}

//static
void
TopologyQueries::createTemporaryTable(pqxx::transaction_base& rTrans,
                                      const std::string& rTableName)
{
    rTrans.exec(
            "CREATE TABLE public." + rTableName + " " +
            "AS SELECT * "
            "FROM planet_osm_line "
            "WHERE highway IS NOT NULL"
    );
}

//static
void
TopologyQueries::createTemporarySchema(pqxx::transaction_base& rTrans,
                                      const std::string& rSchemaName, int srid)
{
    rTrans.exec(
            "SELECT topology.CreateTopology('" +
            rSchemaName + "','" +
            rTrans.quote(srid) + ")"
    );
}
```

```
//static
void
TopologyQueries::addTopoGeometryColumn(pqxx::transaction_base& rTrans,
                                       const std::string& rSchemaName,
                                       const std::string& rOsmEdgeTable)
{
    rTrans.exec(
            "SELECT topology.AddTopoGeometryColumn('" +
            rSchemaName + "', " +
            "'public', '" +
            rOsmEdgeTable + "', " +
            "'topo_geom', 'LINESTRING')"
    );
}

//static
void
TopologyQueries::fillTopoGeometryColumn(pqxx::transaction_base& rTrans,
                                        const std::string& rSchemaName,
                                        const std::string& rOsmEdgeTable,
                                        double tolerance)
{
    rTrans.exec(
            "UPDATE public." +
            rOsmEdgeTable + " " +
            "SET topo_geom = topology.toTopoGeom(way, '" +
            rSchemaName +
            "', 1, " +
            rTrans.quote(tolerance) + ")"
    );
}

//static
void
TopologyQueries::dropTemporaryTable(pqxx::transaction_base& rTrans,
                                    const std::string& rTableName)
{
    rTrans.exec(
            "DROP TABLE IF EXISTS public." + rTableName
    );
}

//static
void
TopologyQueries::dropTemporarySchema(pqxx::transaction_base& rTrans,
                                     const std::string& rSchemaName)
{
    rTrans.exec(
            "DROP SCHEMA IF EXISTS " + rSchemaName + " CASCADE"
    );
}

//static
void
TopologyQueries::deleteTemporaryLayerRecord(pqxx::transaction_base& rTrans,
                                            const std::string& rTableName)
```

```
{
    rTrans.exec(
            "DELETE FROM topology.layer "
            "WHERE table_name = " + rTrans.quote(rTableName)
    );
}

//static
void
TopologyQueries::deleteTemporaryTopoRecord(pqxx::transaction_base& rTrans,
                                              const std::string& rSchemaName)
{
    rTrans.exec(
            "DELETE FROM topology.topology "
            "WHERE name = " + rTrans.quote(rSchemaName)
    );
}
//============================== ACESS      ===================================
//============================== INQUIRY    ===================================
////////////////////////////// PROTECTED  /////////////////////////////////////

////////////////////////////// PRIVATE    /////////////////////////////////////
```

D.11.12 PostGisProvider_test.cc

```
/*
 * DatabaseHandler_test.cc
 *
 * @author  Jonas Bergman
 */

#include "../../postgis/PostGisProvider.h"
#include "../../postgis/RestrictionQueries.h"

#include <iostream>
#include <string>
#include <sstream>
#include <vector>

#include "../../../catchtest/catch.hpp"
#include "../../../config/ConfigurationReader.h"
#include "../../../config/DatabaseConfig.h"
#include "../../../util/TimeToStringMaker.h"
#include "../../../graph/Edge.h"
#include "../../../graph/Vertex.h"
#include "../../../graph/GraphBuilder.h"

SCENARIO ("PostGis topology handling", "[postgis][topology]")
{
    try
    {
        // =================================================================
        GIVEN ("a configuration file with NO topology name")
        {
            WHEN ("we try to read in topology")
            {
                std::string config_file(
```

```
                    "catchtest/testsettings/missing-topo-testsettings.json");
                ConfigurationReader config_reader(config_file);
                Configuration config;
                config_reader.fillConfiguration(config);

                THEN ("we should get an exception")
                {
                    REQUIRE_THROWS_AS (PostGisProvider pgp(config),
                        MapProviderException&);
                }
            }
        }
    }
    catch (ConfigurationException& e) {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
}


SCENARIO ("PostGis queries", "[postgis][query]")
{
    try
    {

        // ===================================================================
        GIVEN ("a valid database configuration structure and "
                "name to existing topology")
        {
            std::string config_file(
                "catchtest/testsettings/mikh0522-testsettings.json");
            ConfigurationReader config_reader(config_file);
            Configuration config;
            config_reader.fillConfiguration(config);

            PostGisProvider db_handler(config);

            // ...............................................................
            WHEN ("we try to fetch a topology")
            {
                Topology topology;

                THEN ("we should not get an exception")
                {
                    REQUIRE_NOTHROW (db_handler.getTopology(topology););
                }
            }

            // ...............................................................
            WHEN ("we try to fetch topology ")
            {
                Topology topology;
                db_handler.getTopology(topology);

                THEN ("we should receive a vertices and edges")
                {
                    REQUIRE (topology.nrVertices() > 0);
```

```
                    REQUIRE (topology.nrEdges() > 0);
                }
            }

            // ...............................................................
            WHEN ("we try to build a graph ")
            {
                Topology topology;
                db_handler.getTopology(topology);
                Configuration config;
                GraphBuilder graph(topology, config);
                std::ostringstream oss;

                THEN ("we should be able to print some information")
                {
                    graph.printGraphInformation(oss);
                    INFO(oss.str());
                    REQUIRE (true);
                }
            }

            // ...............................................................
            WHEN ("fetching an edge from topology")
            {
                Topology topology;
                db_handler.getTopology(topology);
                const Edge& edge = topology.getEdge(1);

                THEN ("we should be able to print it out")
                {
                    INFO (edge);
                    REQUIRE (true);

                    /* Information matches this query:
$ psql -U jonas -d mikh_0522 -c
"SELECT edge_id, osm_id, start_node, end_node, lanes, highway
 FROM   topo_test.edge_data
 JOIN (
 SELECT osm_id, element_id, highway, lanes
 FROM topo_test.relation
 JOIN highways_test
 ON topogeo_id = (topo_geom).id )
 AS osm
 ON edge_id = element_id
 WHERE edge_id = 1;"

 edge_id |  osm_id   | start_node | end_node | lanes |   highway
---------+-----------+------------+----------+-------+-------------
       1 | 124227193 |          1 |       54 |       | residential
(1 row)

                     */
                }
            }

        }
    }
```

```
    catch (ConfigurationException& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (MapProviderException& dbe)
    {
        INFO(dbe.what());
        REQUIRE (false);    // force output of error and failure

    }
}

SCENARIO ("Set costs on Edges", "[postgis][cost]")
{
    try
    {
        // =================================================================
        GIVEN ("a valid database configuration structure and "
                "name to existing topology")
        {
            std::string config_file(
                "catchtest/testsettings/mikh_restr_0617-testsettings.json");
            ConfigurationReader config_reader(config_file);
            Configuration config;
            config_reader.fillConfiguration(config);

            PostGisProvider pgp(config);

            Topology topology;
            pgp.getTopology(topology);

            // ...............................................................
            WHEN ("we try to set restrictions and costs on topology")
            {
                pgp.setRestrictionsAndCosts(topology);

                THEN ("we should be able to read travel time cost on edges")
                {
                    EdgeIdType id = 1;
                    const Edge& edge = topology.getEdge(id);
                    INFO ("edge " << id
                        << ", length: " << edge.geomData().length
                        << ", travel time: "
                        << edge.edgeCost().getCost(EdgeCost::TRAVEL_TIME)
                        << ", total cost: " << edge.cost());
                    INFO ("edge " << edge);
                    REQUIRE (edge.cost() > 0);
                }

                THEN ("we should be able to find cost for barriers")
                {
                    EdgeIdType id = 869;
                    const Edge& edge = topology.getEdge(id);
                    INFO ("edge " << id
                        << ", length: " << edge.geomData().length
                        << ", travel time: "
```

```
                        << edge.edgeCost().getCost(EdgeCost::TRAVEL_TIME)
                        << ", barrier cost: "
                        << edge.edgeCost().getCost(EdgeCost::BARRIER)
                        << ", total cost: " << edge.cost());
                    REQUIRE (edge.cost() > 0);
                }

                THEN ("we should be able to find cost for other hindrances")
                {
                    EdgeIdType id = 869;
                    const Edge& edge = topology.getEdge(id);
                    INFO ("edge " << id
                        << ", length: " << edge.geomData().length
                        << ", travel time: "
                        << edge.edgeCost().getCost(EdgeCost::TRAVEL_TIME)
                        << ", barrier cost: "
                        << edge.edgeCost().getCost(EdgeCost::BARRIER)
                        << ", other cost: "
                        << edge.edgeCost().getCost(EdgeCost::OTHER)
                        << ", total cost: " << edge.cost());
                    REQUIRE (edge.cost() > 0);
                }
            }
        }
    }
    catch (ConfigurationException& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (MapProviderException& dbe)
    {
        INFO(dbe.what());
        REQUIRE (false);    // force output of error and failure
    }

}
```

D.12 osm

D.12.1 README.md

```
OSM
===

OpenStreetMap related classes and constants are placed in this package.


Relations
---------

There is no really easy way to get to relations if data has been imported with
↪ osm2pgsql. Best chance is to import in "slim mode" (with flag `-s`) and look
↪ through table `planet_osm_rel` and search the column `tags` for
↪ `restriction`. Then parse the `members` column for members of the relation
↪ and their roles.
```

```
The TurnRestriction class could be smarter with handling turn either via `nodes`
↪  or `ways` but it is not implemented yet.
```

D.12.2 OsmAccess.h

```cpp
/**  Access to `Access` data from the OSM file.
 *
 * #include "OsmAccess.h"
 *
 * @author  Jonas Bergman
 */

#ifndef OSM_OSMACCESS_H_
#define OSM_OSMACCESS_H_

// SYSTEM INCLUDES
//
#include <algorithm>
#include <initializer_list>
#include <string>
#include <vector>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//
#include "OsmException.h"

// FORWARD REFERENCES
//

/**
 * Class to represent OSM key `access`.
 */
class OsmAccess
{
public:
// TYPES
    enum AccessType
    {
        YES,
        PRIVATE,
        NO,
        PERMISSIVE,
        AGRICULTURAL,
        USE_SIDEPATH,
        DELIVERY,
        DESIGNATED,
        DISMOUNT,
        DISCOURAGED,
        FORESTRY,
        DESTINATION,
        CUSTOMERS,

        NR_ACCESS_TYPES
    };
```

```


    /** Allow access to the types in the 'allowAccessTypes', deny all other.
     */
    struct AccessRule
    {
        AccessRule() = default;
        AccessRule(const AccessRule& from) = default;
        AccessRule(std::initializer_list<AccessType> allowedTypes);

        bool    hasAccess(AccessType type) const;

        std::vector<AccessType> allowAccessToTypes;
    };

// LIFECYCLE
    OsmAccess() = delete;
    OsmAccess(AccessType type);
    OsmAccess(const OsmAccess& from) = default;
    ~OsmAccess() = default;

// OPERATORS
// OPERATIONS
    /** Attempt to parse a string to a AccessType
     * @param   rTypeString     String which could contain a Access type
     * @return  A valid AccessType
     * @throw   OsmException if invalid string.
     */
    static AccessType   parseString(const std::string& rTypeString);

    /** Convert a Access Type to a string representation.
     * @param   accessType     The type to convert.
     * @return  string representation of the type.
     * @throw   OsmException if unknown vehicle type (out of bounds).
     */
    static std::string  toString(AccessType accessType);

    /** Convert this AccessType to a string.
     * @return  string representation of this VehicleType.
     */
    std::string         toString() const;

    /** See if this Access type permits access according to rule;
     * @param   AccessRule
     * @return  true if access is allowed, false if not
     */
    bool                allowsAccess(AccessRule rule) const;

// ACCESS
    /**
     * @return  The access type.
     */
    AccessType          accessType() const;
// INQUIRY
protected:
private:
    AccessType               mType {YES};
```

```
    static const std::string sTypeStrings[];
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* OSM_OSMACCESS_H_ */
```

D.12.3 OsmAccess.cc

```
/*
 * OsmAccess.cc
 *
 * @author  Jonas Bergman
 */


#include "OsmAccess.h"  // class implemented

// AccessRule ---------------------------------------------------------------
OsmAccess::AccessRule::AccessRule(
    std::initializer_list<OsmAccess::AccessType> allowedTypes)
    : allowAccessToTypes(allowedTypes)
{
}

bool
OsmAccess::AccessRule::hasAccess(OsmAccess::AccessType type) const
{
    auto it = std::find(allowAccessToTypes.begin(),
        allowAccessToTypes.end(),
        type);
    return it != allowAccessToTypes.end();
}


/////////////////////////////// PUBLIC ///////////////////////////////////////

//============================= LIFECYCLE ====================================
OsmAccess::OsmAccess(OsmAccess::AccessType type)
    : mType(type)
{}

//============================= OPERATORS ====================================

//static
OsmAccess::AccessType
OsmAccess::parseString(const std::string& rTypeString)
{
    for(size_t i = 0; i < NR_ACCESS_TYPES; ++i)
    {
        if(rTypeString == OsmAccess::sTypeStrings[i])
        {
            return static_cast<AccessType>(i);
```

```
        }
    }
    throw OsmException("OsmAccess:parseString: Unknown Access Type string.");
}

//static
std::string
OsmAccess::toString(OsmAccess::AccessType accessType)
{
    if(accessType >= NR_ACCESS_TYPES)
    {
        throw OsmException("OsmAccess:toString: Unknown Access Type");
    }
    return OsmAccess::sTypeStrings[accessType];
}

std::string
OsmAccess::toString() const
{
    return sTypeStrings[this->mType];
}

bool
OsmAccess::allowsAccess(OsmAccess::AccessRule rule) const
{
    return rule.hasAccess(mType);
}

OsmAccess::AccessType
OsmAccess::accessType() const
{
    return mType;
}

//============================= OPERATIONS ===================================
//============================= ACESS      ===================================
//============================= INQUIRY    ===================================
/////////////////////////////// PROTECTED  ///////////////////////////////////

/////////////////////////////// PRIVATE    ///////////////////////////////////
const std::string OsmAccess::sTypeStrings[] =
{
    "yes",
    "private",
    "no",
    "permissive",
    "agricultural",
    "use_sidepath",
    "delivery",
    "designated",
    "dismount",
    "discouraged",
    "forestry",
    "destination",
    "customers"
};
```

D.12.4 OsmBarrier.h

```
/**  Access to `Barrier` data from the OSM file.
 *
 * #include "OsmBarrier.h"
 *
 * @author  Jonas Bergman
 */

#ifndef OSM_OSMBARRIER_H_
#define OSM_OSMBARRIER_H_

// SYSTEM INCLUDES
//
#include <algorithm>
#include <initializer_list>
#include <map>
#include <string>
#include <vector>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//
#include "OsmException.h"
#include "../graph/Cost.h"

// FORWARD REFERENCES
//


/**
 * Class to represent OSM key `barrier`.
 */
class OsmBarrier
{
public:
// TYPES
    enum BarrierType
    {
        NONE,
        BLOCK,
        BOLLARD,
        BORDER_CONTROL,
        BUMP_GATE,
        BUS_TRAP,
        CATTLE_GRID,
        CHAIN,
        CYCLE_BARRIER,
        DEBRIS,
        ENTRANCE,
        FULLHEIGHT_TURNSTILE,
        GATE,
        HAMPSHIRE_GATE,
        HEIGHT_RESTRICTOR,
        HORSE_STILE,
        JERSEY_BARRIER,
```

```
        KENT_CARRIAGE_GAP,
        KISSING_GATE,
        LIFT_GATE,
        LOG,
        MOTORCYCLE_BARRIER,
        ROPE,
        SALLY_PORT,
        SPIKES,
        STILE,
        SUMP_BUSTER,
        SWING_GATE,
        TOLL_BOOTH,
        TURNSTILE,
        YES,

        NR_BARRIER_TYPES
    };

    /** Barriers which imposes restrictions on access.
     */
    struct RestrictionsRule
    {
        RestrictionsRule() = default;
        RestrictionsRule(const RestrictionsRule& from) = default;
        RestrictionsRule(std::initializer_list<BarrierType> restrictionTypes);

        bool    restrictsAccess(BarrierType type) const;

        std::vector<BarrierType> restrictionTypes;
    };

    /** Barriers which infer costs.
     */
    struct CostsRule
    {
        CostsRule() = default;
        CostsRule(const CostsRule& from) = default;

        bool        costsToPass(BarrierType type) const;
        Cost        getCost(BarrierType type) const;
        void        addCost(BarrierType type, Cost cost);

        std::map<BarrierType, Cost> costs;
    };

// LIFECYCLE
    OsmBarrier() = delete;
    OsmBarrier(BarrierType type);
    OsmBarrier(const OsmBarrier& from) = default;
    ~OsmBarrier() = default;

// OPERATORS
// OPERATIONS
    /** Attempt to parse a string to a BarrierType
     * @param   rTypeString     String which could contain a Barrier type
     * @return  A valid BarrierType
     * @throw   OsmException if invalid string.
```

```
     */
    static BarrierType  parseString(const std::string& rTypeString);

    /** Convert a Barrier Type to a string representation.
     * @param   barrierType     The type to convert.
     * @return  string representation of the type.
     * @throw   OsmException if unknown barrier type (out of bounds).
     */
    static std::string  toString(BarrierType barrierType);

    /** Convert this BarrierType to a string.
     * @return  string representation of this VehicleType.
     */
    std::string         toString() const;

    /** See if this Barrier type permits access according to rule;
     * @param   RestrictionRule
     * @return  true if access is allowed, false if not
     */
    bool                restrictsAccess(RestrictionsRule rule) const;

    /** See if this Barrier type costs to pass according to rule;
     * @param   RestrictionRule
     * @return  true if access is allowed, false if not
     */
    bool                costsToPass(CostsRule rule) const;

// ACCESS
// INQUIRY
protected:
private:
    BarrierType                           mType {NONE};
    static const std::vector<std::string> sTypeStrings;
    static const std::vector<std::string> sDisregardedTypes;
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* OSM_OSMBARRIER_H_ */
```

D.12.5 OsmBarrier.cc

```
/*
 * OsmBarrier.cc
 *
 * @author  Jonas Bergman
 */


#include "OsmBarrier.h"  // class implemented

// RestrictionsRule -----------------------------------------------------------
OsmBarrier::RestrictionsRule::RestrictionsRule(
```

```
    std::initializer_list<OsmBarrier::BarrierType> restrictionTypes)
    : restrictionTypes(restrictionTypes)
{ }

bool
OsmBarrier::RestrictionsRule::restrictsAccess(OsmBarrier::BarrierType type) const
{
    auto it = std::find(restrictionTypes.begin(), restrictionTypes.end(), type);
    return it != restrictionTypes.end();
}

bool
OsmBarrier::CostsRule::costsToPass(OsmBarrier::BarrierType type) const
{
    const auto& it  = costs.find(type);
    return it != costs.end();
}

Cost
OsmBarrier::CostsRule::getCost(OsmBarrier::BarrierType type) const
{
    const auto& it  = costs.find(type);
    if(it != costs.end())
    {
        return it->second;
    }
    else
    {
        return 0;
    }
}

void
OsmBarrier::CostsRule::addCost(OsmBarrier::BarrierType type, Cost cost)
{
    costs.erase(type);
    costs.insert({type, cost});
}

/////////////////////////////// PUBLIC ///////////////////////////////////////

//============================= LIFECYCLE ====================================
OsmBarrier::OsmBarrier(OsmBarrier::BarrierType type)
    : mType(type)
{}

//============================= OPERATORS ====================================

//static
OsmBarrier::BarrierType
OsmBarrier::parseString(const std::string& rTypeString)
{
    for(size_t i = 0; i < sTypeStrings.size(); ++i)
    {
        if(rTypeString == OsmBarrier::sTypeStrings[i])
        {
            return static_cast<BarrierType>(i);
```

```
        }
    }
    // no match in types. Look if it is disregarded or unknown.
    auto it = std::find(sDisregardedTypes.begin(),
                        sDisregardedTypes.end(),
                        rTypeString);
    if(it != sDisregardedTypes.end())
    {
        return BarrierType::NONE;
    }
    throw OsmException("OsmBarrier:parseString: Unknown Barrier Type string: "
        + rTypeString);
}

//static
std::string
OsmBarrier::toString(OsmBarrier::BarrierType accessType)
{
    if(accessType >= sTypeStrings.size())
    {
        throw OsmException("OsmBarrier:toString: Unknown Barrier Type");
    }
    return OsmBarrier::sTypeStrings[accessType];
}

std::string
OsmBarrier::toString() const
{
    return sTypeStrings[this->mType];
}

bool
OsmBarrier::restrictsAccess(OsmBarrier::RestrictionsRule rule) const
{
    return rule.restrictsAccess(mType);
}

bool
OsmBarrier::costsToPass(OsmBarrier::CostsRule rule) const
{
    return rule.costsToPass(mType);
}

//============================= OPERATIONS ===================================
//============================= ACESS      ===================================
//============================= INQUIRY    ===================================
/////////////////////////////// PROTECTED  ///////////////////////////////////

/////////////////////////////// PRIVATE    ///////////////////////////////////
const std::vector<std::string> OsmBarrier::sTypeStrings
{
    "none",
    "block",
    "bollard",
    "border_control",
    "bump_gate",
    "bus_trap",
```

```
    "cattle_grid",
    "chain",
    "cycle_barrier",
    "debris",
    "entrance",
    "full-height_turnstile",
    "gate",
    "hampshire_gate",
    "height_restrictor",
    "horse_stile",
    "jersey_barrier",
    "kent_carriage_gap",
    "kissing_gate",
    "lift_gate",
    "log",
    "motorcycle_barrier",
    "rope",
    "sally_port",
    "spikes",
    "stile",
    "sump_buster",
    "swing_gate",
    "toll_booth",
    "turnstile",
    "yes"
};

const std::vector<std::string> OsmBarrier::sDisregardedTypes
{
    "cable_barrier",
    "city_wall",
    "ditch",
    "fence",
    "guard_rail",
    "handrail",
    "hedge",
    "kerb",
    "retaining_wall",
    "wall",
};
```

D.12.6 OsmException.h

```
/**  Exception thrown in the 'osm' package.
 *
 * #include "OsmException.h"
 *
 * @author  Jonas Bergman
 */

#ifndef OSM_OSMEXCEPTION_H_
#define OSM_OSMEXCEPTION_H_

// SYSTEM INCLUDES
//
#include <exception>
#include <string>
```

```
// PROJECT INCLUDES
//

// LOCAL INCLUDES
//

// FORWARD REFERENCES
//

/**
 * Exception to throw in the 'osm' package.
 * More information of the type of exception is given in the 'what()' message.
 */
class OsmException : public std::exception
{
public:
// LIFECYCLE
    /** Default constructor.
     */
    OsmException() = delete;

    /** Constructor taking a message to display.
     *
     * @param   message     The message to prepend when 'what()' is called.
     */
    OsmException(const std::string& rMessage) noexcept
        : std::exception(), mMessage(rMessage)
    {}

// OPERATORS
// OPERATIONS
// ACCESS
// INQUIRY
    const char* what() const noexcept
    { return (mMessage.c_str()); }

protected:
private:
// ATTRIBUTES
    std::string     mMessage;
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* OSM_OSMEXCEPTION_H_ */
```

D.12.7 OsmHighway.h

```
/**  Access to `Highway` data from the OSM file.
 *
 * #include "OsmHighway.h"
 *
```

```
 * @author Jonas Bergman
 */

#ifndef OSM_OSMHIGHWAY_H_
#define OSM_OSMHIGHWAY_H_

// SYSTEM INCLUDES
//
#include <string>
#include <sstream>
#include <vector>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//
#include "OsmException.h"

// FORWARD REFERENCES
//


/**
 * Class for categoreis of OSM `highway` and `surface`.
 */
class OsmHighway
{
public:
// TYPES
    enum HighwayType
    {
        // roads
        MOTORWAY,
        MOTORWAY_LINK,
        TRUNK,
        TRUNK_LINK,
        PRIMARY,
        PRIMARY_LINK,
        SECONDARY,
        SECONDARY_LINK,
        TERTIARY,
        TERTIARY_LINK,
        UNCLASSIFIED,
        RESIDENTIAL,
        SERVICE,

        // special types
        LIVING_STREET,
        BUS_GUIDEWAY,
        ROAD,

        NR_HIGHWAY_TYPES
    };

    enum SurfaceType
    {
```

```
        PAVED,
        ASPHALT,
        COBBLESTONE,
        COBBLESTONE_FLATTENED,
        SETT,
        CONCRETE,
        CONCRETE_LANES,
        CONCRETE_PLATES,
        PAVING_STONES,
        METAL,
        WOOD,

        UNPAVED,
        COMPACTED,
        DIRT,
        EARTH,
        FINE_GRAVEL,
        GRASS,
        GRASS_PAVER,
        GRAVEL,
        GROUND,
        ICE,
        MUD,
        PEBBLESTONE,
        SALT,
        SAND,
        SNOW,
        WOODCHIPS,

        METAL_GRID,

        NR_SURFACE_TYPES
    };

    enum JunctionType
    {
        ROUNDABOUT
    };
    static constexpr const char* JUNCTION_ROUNDABOUT = "roundabout";

// LIFECYCLE
    OsmHighway() = delete;
    OsmHighway(HighwayType type);
    OsmHighway(const OsmHighway& from) = default;
    ~OsmHighway() = default;

// OPERATORS
// OPERATIONS
    /** Attempt to parse a string to a HighwayType
     * @param   rTypeString     String which could contain a Highway type
     * @return  A valid HighwayType
     * @throw   OsmException if invalid string.
     */
    static HighwayType  parseString(const std::string& rTypeString);

    /** Attempt to parse a string to a SurfaceType
     * @param   rTypeString     String which could contain a Surface type
```

```
     * @return  A valid SurfaceType
     * @throw   OsmException if invalid string.
     */
    static SurfaceType parseSurfaceString(const std::string& rSurfaceString);

    /** Convert a Highway Type to a string representation.
     * @param   highwayType     The type to convert.
     * @return  string representation of the type.
     * @throw   OsmException if unknown highway type (out of bounds).
     */
    static std::string  toString(HighwayType highwayType);

    /** Convert a SurfaceType to a string representation.
     * @param   surfaceType     The type to convert.
     * @return  string representation of the type.
     * @throw   OsmException if unknown highway type (out of bounds).
     */
    static std::string  toSurfaceString(SurfaceType surfaceType);

    /** Convert this HighwayType to a string.
     * @return  string representation of this HighwayType.
     */
    std::string         toString() const;

// ACCESS
    /**
     * @return A vector of all types as strings.
     */
    static const std::vector<std::string>& typeStrings();

    /**
     * @return A vector of all surface types as strings.
     */
    static const std::vector<std::string>& surfaceTypeStrings();

    /** Return "(motorway, trunk....)".
     * @return A string of all types, comma separated, with parentheses round.
     */
    static std::string typesAsCommaSeparatedString();

// INQUIRY
protected:
private:
    HighwayType                             mType {ROAD};
    static const std::vector<std::string>   sTypeStrings;
    static const std::vector<std::string>   sSurfaceTypeStrings;
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* OSM_OSMHIGHWAY_H_ */
```

D.12.8 OsmHighway.cc

```
/*
 * OsmHighway.cc
 *
 * @author  Jonas Bergman
 */

#include "OsmHighway.h"  // class implemented

////////////////////////////// PUBLIC //////////////////////////////////////

//============================ LIFECYCLE ===================================
OsmHighway::OsmHighway(OsmHighway::HighwayType type)
    : mType(type)
{}

//============================ OPERATORS ===================================
//============================ OPERATIONS ==================================
//static
OsmHighway::HighwayType
OsmHighway::parseString(const std::string& rTypeString)
{
    for(size_t i = 0; i < NR_HIGHWAY_TYPES; ++i)
    {
        if(rTypeString == OsmHighway::sTypeStrings[i])
        {
            return static_cast<HighwayType>(i);
        }
    }
    throw OsmException("OsmHighway:parseString: Unknown Highway Type string.");
}

//static
OsmHighway::SurfaceType
OsmHighway::parseSurfaceString(const std::string& rSurfaceString)
{
    for(size_t i = 0; i < NR_SURFACE_TYPES; ++i)
    {
        if(rSurfaceString == OsmHighway::sSurfaceTypeStrings[i])
        {
            return static_cast<SurfaceType>(i);
        }
    }
    throw OsmException("OsmHighway:parseSurfaceString: Unknown Surface Type string.");
}

//static
std::string
OsmHighway::toString(OsmHighway::HighwayType highwayType)
{
    if(highwayType >= NR_HIGHWAY_TYPES)
    {
        throw OsmException("OsmHighway:toString: Unknown Highway Type");
    }
    return OsmHighway::sTypeStrings[highwayType];
}

```

```
//static
std::string
OsmHighway::toSurfaceString(OsmHighway::SurfaceType surfaceType)
{
    if(surfaceType >= NR_SURFACE_TYPES)
    {
        throw OsmException("OsmHighway:toSurfaceString: Unknown Surface Type");
    }
    return OsmHighway::sSurfaceTypeStrings[surfaceType];
}

std::string
OsmHighway::toString() const
{
    return sTypeStrings[this->mType];
}

//============================= ACESS      ===================================
//static
const std::vector<std::string>&
OsmHighway::typeStrings()
{
    return OsmHighway::sTypeStrings;
}

//static
const std::vector<std::string>&
OsmHighway::surfaceTypeStrings()
{
    return OsmHighway::sSurfaceTypeStrings;
}

// static
std::string
OsmHighway::typesAsCommaSeparatedString()
{
    std::string cols;
    std::stringstream ss;
    ss << "(";
    for(size_t i = 0; i < sTypeStrings.size(); ++i)
    {
        ss << "'" << sTypeStrings[i] << "'";
        if(i < sTypeStrings.size() - 1)
        {
            ss << ", ";
        }
    }
    ss << ")";
    return ss.str();
}
//============================= INQUIRY    ===================================
///////////////////////////////// PROTECTED  ///////////////////////////////////

///////////////////////////////// PRIVATE    ///////////////////////////////////
const std::vector<std::string> OsmHighway::sTypeStrings
{
    "motorway",
```

```
    "motorway_link",
    "trunk",
    "trunk_link",
    "primary",
    "primary_link",
    "secondary",
    "secondary_link",
    "tertiary",
    "tertiary_link",
    "unclassified",
    "residential",
    "service",

    "living_street",
    "bus_guideway",
    "road"
};

const std::vector<std::string> OsmHighway::sSurfaceTypeStrings
{
    "paved",
    "asphalt",
    "cobblestone",
    "cobblestone:flattened",
    "sett",
    "concrete",
    "concrete:lanes",
    "concrete:plates",
    "paving_stones",
    "metal",
    "wood",

    "unpaved",
    "compacted",
    "dirt",
    "earth",
    "fine_gravel",
    "grass",
    "grass_paver",
    "gravel",
    "ground",
    "ice",
    "mud",
    "pebblestone",
    "salt",
    "sand",
    "snow",
    "woodchips",

    "metal_grid"
};
```

D.12.9 OsmId.h

```
/*
 * OsmId.h
 *
```

```
 * @author  Jonas Bergman
 */

#ifndef OSM_OSMID_H_
#define OSM_OSMID_H_

#include <limits>

typedef long long OsmIdType;

struct Osm
{
    static const OsmIdType MAX_ID;
};



#endif /* OSM_OSMID_H_ */
```

D.12.10 OsmId.cc

```
/*
 * OsmId.cc
 *
 * @author  Jonas Bergman
 */

#include "OsmId.h"

const OsmIdType Osm::MAX_ID = std::numeric_limits<OsmIdType>::max();
```

D.12.11 OsmTurningRestriction.h

```
/**  Access to Turning restriction data from the OSM file.
 *
 * #include "OsmTurningRestriction.h"
 *
 * @author  Jonas Bergman
 */

#ifndef OSM_OSMTURNINGRESTRICTION_H_
#define OSM_OSMTURNINGRESTRICTION_H_

// SYSTEM INCLUDES
//
#include <algorithm>
#include <initializer_list>
#include <sstream>
#include <string>
#include <vector>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//
#include "OsmException.h"
```

```cpp
#include "../graph/Edge.h"
#include "../graph/Vertex.h"

// FORWARD REFERENCES
//

/**
 * Class for working with "turning restrictions" from OSM relations.
 */
class OsmTurningRestriction
{
public:
// TYPES
    enum TurningRestrictionType
    {
        NONE,
        NO_LEFT_TURN,
        NO_RIGHT_TURN,
        NO_STRAIGHT_ON,
        NO_U_TURN,
        ONLY_RIGHT_TURN,
        ONLY_LEFT_TURN,
        ONLY_STRAIGHT_ON,
        NO_ENTRY,
        NO_EXIT,

        NR_TURNING_RESTRICTION_TYPES
    };

    enum TurningViaType
    {
        VIA_NODE,
        VIA_WAY
    };

// LIFECYCLE
    /** Constructor. Disabled*/
    OsmTurningRestriction() = delete;

    /** Constructor.
     * Turning restriction via a vertex.
     * @param   type        The type of turning restriction.
     * @param   fromEdgeId  The Edge the turn starts at.
     * @param   viaVertexId The Vertex the turn travels via.
     * @param   toEdgeId    The Edge the turn ends at.
     */
    OsmTurningRestriction(TurningRestrictionType    type,
                          EdgeIdType                fromEdgeId,
                          VertexIdType              viaVertexId,
                          EdgeIdType                toEdgeId);

    /** Constructor.
     * Turning restriction via  other Edges.
     * @param   type        The type of turning restriction.
     * @param   fromEdgeId  The Edge the turn starts at.
     * @param   viaOsmIds   String with the OsmIds of roads the turn travels via.
     * @param   toEdgeId    The Edge the turn ends at.
```

```
     */
    OsmTuringRestriction(TurningRestrictionType    type,
                         EdgeIdType                fromEdgeId,
                         std::string               viaOsmIds,
                         EdgeIdType                toEdgeId);

    /** Copy constructor. */
    OsmTurningRestriction(const OsmTurningRestriction& from) = default;

    /** Destructor. */
    ~OsmTurningRestriction() = default;

// OPERATORS
// OPERATIONS
    /** Attempt to parse a string to a TurningRestrictionType
     * @param   rTypeString     String which could contain a Turning Restriction
     * @return  A valid TurningRestrictionType
     * @throw   OsmException if invalid string.
     */
    static TurningRestrictionType  parseString(const std::string& rTypeString);

    /** Convert a Turning Restriction Type to a string representation.
     * @param   turnRestrictionType     The type to convert.
     * @return  string representation of the type.
     * @throw   OsmException if unknown turn restriction type (out of bounds).
     */
    static std::string  toString(TurningRestrictionType turnRestrictionType);

    /** Convert this TurningRestriction to a string.
     * @return  string representation of this turning restriction.
     */
    std::string         toString() const;

    /** Convert this TurningRestrictions type to a string.
     * @return  string representation of this turning restriction type.
     */
    std::string         typeToString() const;

// ACCESS
    /**
     * @return  The Edge Id of the 'from' edge
     */
    EdgeIdType          fromEdgeId() const;

    /**
     * @return  The Via type, 'way' or 'node'.
     */
    TurningViaType      viaType() const;

    /**
     * @return  The Ids of the Edges in a 'via way' relation.
     */
    std::string         viaOsmIds() const;

    /**
     * @return  The Vertex Id of the 'via' vertex.
     */
```

```
    VertexIdType        viaVertexId() const;

    /**
     * @return  The Edge id of the 'to' Edge.
     */
    EdgeIdType          toEdgeId() const;

// INQUIRY
    /** Check if an Edge is in this restriction.
     * @param   Edge Id to check.
     * @return  true if the edge is part of this restriction, false if not.
     */
    bool                isInRestriction(EdgeIdType edgeId) const;

    /** Check if Travel from 'from' to 'to' is restricted.
     * @param   fromEdgeId    Travel from edge.
     * @param   toEdgeId      Travel to edge.
     * @return  true if travle is restricted.
     */
    bool                isRestricted(
                            EdgeIdType fromEdgeId,
                            EdgeIdType toEdgeId) const;

protected:
private:
    TurningRestrictionType              mType {NONE};
    EdgeIdType                          mFromEdgeId;
    TurningViaType                      mViaType {VIA_NODE};
    std::string                         mViaOsmIds;
    VertexIdType                        mViaVertexId;
    EdgeIdType                          mToEdgeId;
    static std::vector<std::string>     sTypeStrings;
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* OSM_OSMTURNINGRESTRICTION_H_ */
```

D.12.12 OsmTurningRestriction.cc

```
/*
 * OsmTurningRestriction.cc
 *
 * @author  Jonas Bergman
 */

#include "OsmTurningRestriction.h"  // class implemented

/////////////////////////////// PUBLIC ///////////////////////////////////////

//============================= LIFECYCLE ====================================
OsmTurningRestriction::OsmTurningRestriction(
    OsmTurningRestriction::TurningRestrictionType   type,
```

```
    EdgeIdType                                      fromEdgeId,
    VertexIdType                                    viaVertexId,
    EdgeIdType                                      toEdgeId)
    : mType(type),
      mFromEdgeId(fromEdgeId),
      mViaType(VIA_NODE),
      mViaOsmIds(),
      mViaVertexId(viaVertexId),
      mToEdgeId(toEdgeId)
{}

OsmTurningRestriction::OsmTurningRestriction(
    OsmTurningRestriction::TurningRestrictionType   type,
    EdgeIdType                                      fromEdgeId,
    std::string                                     viaOsmIds,
    EdgeIdType                                      toEdgeId)
    : mType(type),
      mFromEdgeId(fromEdgeId),
      mViaType(VIA_WAY),
      mViaOsmIds(viaOsmIds),
      mViaVertexId(),
      mToEdgeId(toEdgeId)
{}

//============================= OPERATORS ====================================
//============================= OPERATIONS ===================================
//static
OsmTurningRestriction::TurningRestrictionType
OsmTurningRestriction::parseString(const std::string& rTypeString)
{
    for(size_t i = 0; i < sTypeStrings.size(); ++i)
    {
        if(rTypeString == OsmTurningRestriction::sTypeStrings[i])
        {
            return static_cast<TurningRestrictionType>(i);
        }
    }
    throw OsmException(
        "OsmTurningRestriction:parseString: "
        "Unknown TurningRestriction Type string.");
}

//static
std::string
OsmTurningRestriction::toString(
    OsmTurningRestriction::TurningRestrictionType turnRestrictionType)
{
    if(turnRestrictionType >= sTypeStrings.size())
    {
        throw OsmException(
            "OsmTurningRestriction:toString: Unknown TurningRestriction Type");
    }
    return OsmTurningRestriction::sTypeStrings[turnRestrictionType];
}

std::string
OsmTurningRestriction::toString() const
```

```
{
    std::ostringstream oss;
    oss << sTypeStrings[this->mType] << ": "
        << "from: " << mFromEdgeId;

    // via vertex
    if(mViaType == VIA_NODE)
    {
        oss << ", via vertex: " << mViaVertexId;
    }
    else // via edges
    {
        oss << ", via edges: [" << mViaOsmIds << "]";
    }

    oss << ", to: " << mToEdgeId;

    return oss.str();
}

std::string
OsmTurningRestriction::typeToString() const
{
    return sTypeStrings[this->mType];
}
//============================ ACESS      ===================================
EdgeIdType
OsmTurningRestriction::fromEdgeId() const
{
    return mFromEdgeId;
}

OsmTurningRestriction::TurningViaType
OsmTurningRestriction::viaType() const
{
    return mViaType;
}

std::string
OsmTurningRestriction::viaOsmIds() const
{
    return mViaOsmIds;
}

VertexIdType
OsmTurningRestriction::viaVertexId() const
{
    return mViaVertexId;
}

EdgeIdType
OsmTurningRestriction::toEdgeId() const
{
    return mToEdgeId;
}
//============================ INQUIRY    ===================================
bool
```

```
OsmTurningRestriction::isInRestriction(EdgeIdType edgeId) const
{
    if(edgeId == mFromEdgeId
    || edgeId == mToEdgeId)
    {
        return true;
    }
    return false;
}

bool
OsmTurningRestriction::isRestricted(
    EdgeIdType fromEdgeId,
    EdgeIdType toEdgeId) const
{
    if(mFromEdgeId == fromEdgeId && mToEdgeId == toEdgeId)
    {
        if(mType == NO_LEFT_TURN
        || mType == NO_RIGHT_TURN
        || mType == NO_STRAIGHT_ON
        || mType == NO_U_TURN
        || mType == NO_ENTRY
        || mType == NO_EXIT)
        {
            return true;
        }
    }
    return false;
}
/////////////////////////////// PROTECTED  ///////////////////////////////////

/////////////////////////////// PRIVATE    ///////////////////////////////////
// static
std::vector<std::string> OsmTurningRestriction::sTypeStrings
{
    "none",
    "no_left_turn",
    "no_right_turn",
    "no_straight_on",
    "no_u_turn",
    "only_right_turn",
    "only_left_turn",
    "only_straight_on",
    "no_entry",
    "no_exit"
};
```

D.12.13 OsmVehicle.h

```
/**  Access to Vehicle data from the OSM file.
 *
 * #include "OsmVehicle.h"
 *
 * @author  Jonas Bergman
 */

#ifndef OSM_OSMVEHICLE_H_
```

```
#define OSM_OSMVEHICLE_H_

// SYSTEM INCLUDES
//
#include <string>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//
#include "OsmException.h"

// FORWARD REFERENCES
//

/**
 * Class for working with different categories of vehicles.
 */
class OsmVehicle
{
public:
// TYPES
    enum VehicleType
    {
        MOTORCAR,
        GOODS,
        HGV,
        PSV,
        LHV,
        MOTOR_VEHICLE,
        VEHICLE,

        NR_VEHICLE_TYPES
    };

// LIFECYCLE
    OsmVehicle() = delete;
    OsmVehicle(VehicleType type);
    OsmVehicle(const OsmVehicle& from) = default;
    ~OsmVehicle() = default;

// OPERATORS
// OPERATIONS
    /** Attempt to parse a string to a VehicleType
     * @param    rTypeString     String which could contain a Vehicle type
     * @return   A valid VehicleType
     * @throw    OsmException if invalid string.
     */
    static VehicleType  parseString(const std::string& rTypeString);

    /** Convert a Vehicle Type to a string representation.
     * @param    vehicleType     The type to convert.
     * @return   string representation of the type.
     * @throw    OsmException if unknown vehicle type (out of bounds).
     */
    static std::string  toString(VehicleType vehicleType);
```

```

    /** Convert this VehicleType to a string.
     * @return  string representation of this VehicleType.
     */
    std::string         toString() const;

// ACCESS
// INQUIRY
protected:
private:
    VehicleType              mType {VEHICLE};
    static const std::string sTypeStrings[];
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* OSM_OSMVEHICLE_H_ */
```

D.12.14 OsmVehicle.cc

```
/*
 * OsmVehicle.cc
 *
 * @author  Jonas Bergman
 */

#include "OsmVehicle.h"  // class implemented

///////////////////////////////// PUBLIC ///////////////////////////////////////

//============================= LIFECYCLE ====================================
OsmVehicle::OsmVehicle(OsmVehicle::VehicleType type)
    : mType(type)
{}

//============================= OPERATORS ====================================

//static
OsmVehicle::VehicleType
OsmVehicle::parseString(const std::string& rTypeString)
{
    for(size_t i = 0; i < NR_VEHICLE_TYPES; ++i)
    {
        if(rTypeString == OsmVehicle::sTypeStrings[i])
        {
            return static_cast<VehicleType>(i);
        }
    }
    throw OsmException("OsmVehicle:parseString: Unknown Vehicle Type string.");
}

//static
std::string
```

```
OsmVehicle::toString(OsmVehicle::VehicleType vehicleType)
{
    if(vehicleType >= NR_VEHICLE_TYPES)
    {
        throw OsmException("OsmVehicle:toString: Unknown Vehicle Type");
    }
    return OsmVehicle::sTypeStrings[vehicleType];
}

std::string
OsmVehicle::toString() const
{
    return sTypeStrings[this->mType];
}

//============================ OPERATIONS ===================================
//============================ ACESS      ===================================
//============================ INQUIRY    ===================================
/////////////////////////////// PROTECTED  ///////////////////////////////////

/////////////////////////////// PRIVATE    ///////////////////////////////////
const std::string OsmVehicle::sTypeStrings[] =
{
    "motorcar",
    "goods",
    "hgv",
    "psv",
    "lhv",
    "motor_vehicle",
    "vehicle"
};
```

D.12.15 OsmAccess_test.cc

```
/*
 * OsmAccess_test.cc
 *
 * @author  Jonas Bergman
 */

#include "../OsmAccess.h"
#include "../../catchtest/catch.hpp"

SCENARIO ("OsmAccess functionality testing", "[osm][access]")
{
    try
    {
        // -------------------------------------------------------------------
        GIVEN ("a valid string of an access type")
        {
            std::string type_string("designated");

            //..............................................................
            WHEN ("parsing string to an AccessType")
            {
                OsmAccess::AccessType type =
                    OsmAccess::parseString(type_string);
```

```
                THEN ("we should get the corresponding type")
                {
                    REQUIRE (type == OsmAccess::AccessType::DESIGNATED);
                }
            }
        }

        // ------------------------------------------------------------------
        GIVEN ("an invalid string of an access type")
        {
            std::string type_string("foo");

            //..................................................................
            WHEN ("parsing string to a AccessType")
            {
                THEN ("we should get an OsmException")
                {
                    REQUIRE_THROWS_AS (OsmAccess::parseString(type_string),
                        OsmException&);
                }
            }
        }

        // ------------------------------------------------------------------
        GIVEN ("an access type")
        {
            OsmAccess type(OsmAccess::DELIVERY);

            //..................................................................
            WHEN ("converting type to a string")
            {
                THEN ("we should the corresponding string")
                {
                    REQUIRE (type.toString() == "delivery");
                }
            }
        }

        // ------------------------------------------------------------------
        GIVEN ("an access rule")
        {
            OsmAccess::AccessRule rule({OsmAccess::YES, OsmAccess::PERMISSIVE});

            //..................................................................
            WHEN ("checking for access for type not in rule")
            {
                OsmAccess type(OsmAccess::DELIVERY);

                THEN ("we should not be allowed access")
                {
                    REQUIRE_FALSE (type.allowsAccess(rule));
                }
            }
        }
    }
    catch (OsmException& oe)
    {
```

```
            INFO(oe.what());
            REQUIRE (false);    // force output of error and failure
        }
        catch (const std::exception& e)
        {
            INFO(e.what());
            REQUIRE (false);    // force output of error and failure

        }
}
```

D.12.16 OsmBarrier_test.cc

```
/*
 * OsmBarrier_test.cc
 *
 * @author  Jonas Bergman
 */

#include "../OsmBarrier.h"
#include "../../catchtest/catch.hpp"

SCENARIO ("OsmBarrier functionality testing", "[osm][barrier]")
{
    try
    {
        // ------------------------------------------------------------------
        GIVEN ("a valid string of an access type")
        {
            std::string type_string("swing_gate");

            //...............................................................
            WHEN ("parsing string to a BarrierType")
            {
                OsmBarrier::BarrierType type =
                    OsmBarrier::parseString(type_string);
                THEN ("we should get the corresponding type")
                {
                    REQUIRE (type == OsmBarrier::BarrierType::SWING_GATE);
                }
            }
        }

        // ------------------------------------------------------------------
        GIVEN ("an invalid string of a barrier type")
        {
            std::string type_string("foo");

            //...............................................................
            WHEN ("parsing string to a BarrierType")
            {
                THEN ("we should get an OsmException")
                {
                    REQUIRE_THROWS_AS (OsmBarrier::parseString(type_string),
                        OsmException&);
                }
            }
```

```
        }

        // -------------------------------------------------------------------
        GIVEN ("a barrier type")
        {
            OsmBarrier type(OsmBarrier::CATTLE_GRID);

            //..................................................................
            WHEN ("converting type to a string")
            {
                THEN ("we should the corresponding string")
                {
                    REQUIRE (type.toString() == "cattle_grid");
                }
            }
        }

        // -------------------------------------------------------------------
        GIVEN ("a restriction rule")
        {
            OsmBarrier::RestrictionsRule rule({OsmBarrier::YES, OsmBarrier::BOLLARD});

            //..................................................................
            WHEN ("checking if access is restricted for type not in rule")
            {
                OsmBarrier type(OsmBarrier::SPIKES);

                THEN ("we should be told there is no restriction on access")
                {
                    REQUIRE_FALSE (type.restrictsAccess(rule));
                }
            }
            //..................................................................
            WHEN ("checking if access is restricted for type in rule")
            {
                OsmBarrier type(OsmBarrier::BOLLARD);

                THEN ("we should be told there restriction on access")
                {
                    REQUIRE (type.restrictsAccess(rule));
                }
            }
        }

        // -------------------------------------------------------------------
        GIVEN ("a cost rule")
        {
//            OsmBarrier::CostsRule rule({OsmBarrier::GATE, OsmBarrier::DEBRIS});
            OsmBarrier::CostsRule rule;
            rule.addCost(OsmBarrier::GATE, 10);
            rule.addCost(OsmBarrier::DEBRIS, 10);

            //..................................................................
            WHEN ("checking if access costs for type not in rule")
            {
                OsmBarrier type(OsmBarrier::YES);

```

```
                THEN ("we should be told there is no cost on access")
                {
                    REQUIRE_FALSE (type.costsToPass(rule));
                }
            }
            //.................................................................
            WHEN ("checking if access costs for type in rule")
            {
                OsmBarrier type(OsmBarrier::DEBRIS);

                THEN ("we should be told there is cost on access")
                {
                    REQUIRE (type.costsToPass(rule));
                }
            }
        }
    }
    catch (OsmException& oe)
    {
        INFO(oe.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (const std::exception& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure

    }
}
```

D.12.17 OsmHighway_test.cc

```
/*
 * OsmHighway_test.cc
 *
 * @author  Jonas Bergman
 */

#include "../OsmHighway.h"
#include "../../catchtest/catch.hpp"

SCENARIO ("OsmHighway functionality testing", "[osm][highway]")
{
    try
    {
        // ------------------------------------------------------------------
        GIVEN ("a valid string of a highway type")
        {
            std::string type_string("primary");

            //.................................................................
            WHEN ("parsing string to a HighwayType")
            {
                OsmHighway::HighwayType type =
                    OsmHighway::parseString(type_string);
                THEN ("we should get the corresponding type")
                {
```

```
                REQUIRE (type == OsmHighway::HighwayType::PRIMARY);
            }
        }
    }

    // -------------------------------------------------------------------
    GIVEN ("an invalid string of a highway type")
    {
        std::string type_string("foo");

        //..................................................................
        WHEN ("parsing string to a HighwayType")
        {
            THEN ("we should get an OsmException")
            {
                REQUIRE_THROWS_AS (OsmHighway::parseString(type_string),
                OsmException&);
            }
        }
    }

    // -------------------------------------------------------------------
    GIVEN ("a highway type")
    {
        OsmHighway type(OsmHighway::PRIMARY);

        //..................................................................
        WHEN ("converting type to a string")
        {
            THEN ("we should the corresponding strng")
            {
                REQUIRE (type.toString() == "primary");
            }
        }
    }
}
catch (OsmException& oe)
{
    INFO(oe.what());
    REQUIRE (false);    // force output of error and failure
}
catch (const std::exception& e)
{
    INFO(e.what());
    REQUIRE (false);    // force output of error and failure

}
}
```

D.12.18 OsmTurningRestriction_test.cc

D.12.19 OsmVehicle_test.cc

```
/*
 * OsmVehicle_test.cc
 *
 * @author  Jonas Bergman
```

```
 */

#include "../OsmVehicle.h"
#include "../../catchtest/catch.hpp"

SCENARIO ("OsmVehicle functionality testing", "[osm][vehicle]")
{
    try
    {
        // ------------------------------------------------------------------
        GIVEN ("a valid string of a vehicle type")
        {
            std::string type_string("motorcar");

            //...................................................................
            WHEN ("parsing string to a VehicleType")
            {
                OsmVehicle::VehicleType type =
                    OsmVehicle::parseString(type_string);
                THEN ("we should get the corresponding type")
                {
                    REQUIRE (type == OsmVehicle::VehicleType::MOTORCAR);
                }
            }
        }

        // ------------------------------------------------------------------
        GIVEN ("an invalid string of a vehicle type")
        {
            std::string type_string("foo");

            //...................................................................
            WHEN ("parsing string to a VehicleType")
            {
                THEN ("we should get an OsmException")
                {
                    REQUIRE_THROWS_AS (OsmVehicle::parseString(type_string),
                    OsmException&);
                }
            }
        }

        // ------------------------------------------------------------------
        GIVEN ("a Vehicle type")
        {
            OsmVehicle type(OsmVehicle::PSV);

            //...................................................................
            WHEN ("converting type to a string")
            {
                THEN ("we should the corresponding string")
                {
                    REQUIRE (type.toString() == "psv");
                }
            }
        }
    }
```

```
    catch (OsmException& oe)
    {
        INFO(oe.what());
        REQUIRE (false);    // force output of error and failure
    }
    catch (const std::exception& e)
    {
        INFO(e.what());
        REQUIRE (false);    // force output of error and failure

    }
}
```

D.13 preparation

D.13.1 README.md

```
Preparing database
==================

The preparation differs depending on how you import the map data and build the topology. So far in this project import has been done with `osm2pgsql` and topology built with `postgis_topology`.


Preparation for `osm2pgsql` and `postgis_topology`
-----------------------------------------------------

To prepare the database for this software module when using `osm2pgsql` as importer of OpenStreetMap data into the database, we need to install extensions for:

- `postgis`
- `postgis_topology`
- `hstore`

and a couple of custom functions for finding turning restrictions:

- function `find_topo_edges_at_turning_restriction()`
- function `find_osm_turning_restrictions()`

The steps to follow are (assuming `mikhailovsk.osm` as source for OpenStreetMap data, and `tester` as a user with administrative rights in database, and `mikh_0530` as name of database):

### 1. Create database

    $ createdb mikh_0530 -U tester

### 2. Install extensions and functions

    $ psql -U tester -d mikh_0530 -f init_osm2pgsql_postgis_topology.sql

### 3. Import OSM data

    $ osm2pgsql -U tester -d mikh_0530 -s -k -S LGU.style mikhailovsk.osm
```

```
This uses the tool `osm2pgsql` to parse the osm-file into database tables.
The flags are

- `-s` slim, keeping extra tables.
- `-k` keeping tags in `hstore` if not in their own column.
- `-S` Style file, configuring which tags to have columns or not.

### 4. Building topology

This step is optional. It should be efficient and safe to build the topology once
 ↪  and for all after importing as differing conditions and temporary closures
 ↪  could be specified with costs and restrictions instead of via topology. But
 ↪  one can also configure the tool to build topology on each call, see the
 ↪  `configuration` package.

    $ psql -U tester -d mikh_0530 -f build_postgis_topology.sql

This step creates a table `public.highways_lgu` and adds a new schema called
 ↪  `topo_lgu` which contains tables for the topology.


Test databases
--------------
During testing different databases has been tested, they were created so:

#### `mikh_style`

    $ createdb mikh_style -U jonas
    $ psql -U jonas -d mikh_style -c "CREATE extension postgis;"
    $ psql -U jonas -d mikh_style -c "CREATE extension postgis_topology;"
    $ psql -U jonas -d mikh_style -c "CREATE extension hstore;"
    $ psql -U jonas -d mikh_style -c "SET search_path=topology,public
    $ osm2pgsql -U jonas -d mikh_style -s -k -S new.style mikhailovsk.osm
    $ psql -U jonas -d mikh_style -c "CREATE TABLE highways_test AS SELECT * FROM
     ↪  planet_osm_line WHERE highway IS NOT NULL;"
    $ psql -U jonas -d mikh_style -c "SELECT topology.CreateTopology('topo_test',
     ↪  900913);"
    $ psql -U jonas -d mikh_style -c "SELECT
     ↪  topology.AddTopoGeometryColumn('topo_test', 'public', 'highways_test',
     ↪  'topo_geom', 'LINESTRING');"
    $ psql -U jonas -d mikh_style -c "UPDATE highways_test SET topo_geom =
     ↪  topology.toTopoGeom(way, 'topo_test', 1, 1.0);"

#### `mikh_0522`

    $ createdb mikh_0522 -U jonas
    $ psql -U jonas -d mikh_0522 -c "CREATE EXTENSION postgis;"
    $ psql -U jonas -d mikh_0522 -c "CREATE EXTENSION postgis_topology;"
    $ psql -U jonas -d mikh_0522 -c "CREATE EXTENSION hstore;"
    $ osm2pgsql -U jonas -d mikh_0522 -s -k -S LGU.style mikhailovsk.osm
    $ psql -U jonas -d mikh_0522 -c "CREATE TABLE highways_test AS SELECT * FROM
     ↪  planet_osm_line WHERE highway IS NOT NULL;"
    $ psql -U jonas -d mikh_0522 -c "SELECT topology.CreateTopology('topo_test',
     ↪  900913);"
    $ psql -U jonas -d mikh_0522 -c "SELECT
     ↪  topology.AddTopoGeometryColumn('topo_test', 'public', 'highways_test',
     ↪  'topo_geom', 'LINESTRING');"
```

```
    $ psql -U jonas -d mikh_0522 -c "UPDATE highways_test SET topo_geom =
    ↪  topology.toTopoGeom(way, 'topo_test', 1, 1.0);"

#### `mikh_0530`
Described above.

#### `mikh_restr_0602`
As `mikh_0530` but using the modified osm-file `mikhailovsk-turnrestriction.osm`
↪  instead. That file has been modified with a turn restriction for testing
↪  purposes.

#### `mikh_restr_0617`
As `mikh_0602` but extra columns in `planet_osm_point` to get point restrictions.
```

D.13.2 build_postgis_topology.sql

```sql
CREATE TABLE highways_lgu
AS SELECT *
FROM planet_osm_line
WHERE highway IS NOT NULL;

SELECT topology.CreateTopology('topo_lgu', 900913);

SELECT topology.AddTopoGeometryColumn('topo_lgu',
                                      'public',
                                      'highways_lgu',
                                      'topo_geom',
                                      'LINESTRING');

UPDATE highways_lgu SET topo_geom = topology.toTopoGeom(way, 'topo_lgu', 1, 1.0);
```

D.13.3 init_osm2pgsql_postigs_topology.sql

```sql
CREATE EXTENSION postgis;
CREATE EXTENSION postgis_topology;
CREATE EXTENSION hstore;

DROP TABLE IF EXISTS turning_restrictions;
CREATE TABLE turning_restrictions(
   from_osm_id       bigint,
   to_osm_id         bigint,
   via_osm           varchar,
   edge_ids          integer[],
   restriction_type  varchar);

--
-- Find the topology edge ids affected by osm turn restrictions.
--
CREATE OR REPLACE FUNCTION
   find_topo_edges_at_turning_restriction(
      osm_edges_table text,
      from_osm_id bigint,
      to_osm_id bigint,
      topo_edges_table text)
RETURNS setof RECORD
AS $$
```

```
BEGIN
   RETURN QUERY EXECUTE format('
      SELECT edge_id
      FROM %4$s
      WHERE ST_DWithin (
         geom,
         (   SELECT ST_Intersection(a.way, b.way)
             FROM %1$I a, %1$I b
             WHERE a.osm_id = %2$s AND b.osm_id = %3$s
         ),
         1.0
      );'
      , osm_edges_table, from_osm_id, to_osm_id, topo_edges_table);
END;
$$ LANGUAGE 'plpgsql';


--
-- Find all the restrictions and put them in table 'turning_restrictions'
--
CREATE OR REPLACE FUNCTION
    find_osm_turning_restrictions(osm_edges_table text, topo_edges_table text)
RETURNS integer
AS $$
DECLARE
   nrFindings integer := 0;
   nrFrom integer := 0;
   nrTo integer := 0;
   restrictionRecord  record;
   ix integer;
   id bigint;
   fromOsmId bigint;
   toOsmId bigint;
   viaText text := '';
   restrictions text[] := '{
            "no_right_turn",
            "no_left_turn",
            "no_u_turn",
            "no_straight_on",
            "only_right_turn",
            "only_left_turn",
            "only_straight_on",
            "no_entry",
            "no_exit"
         }';
   restrType text;
   edgeId integer;
   edges integer[];

BEGIN
   FOR restrictionRecord IN
      SELECT *
      FROM planet_osm_rels
      WHERE (
         SELECT 'restriction' = ANY(tags)
      )
      AND ( -- check that the restriction type is given in tags
```

```
            SELECT restrictions && tags
        )
    LOOP
        -- look through 'members' in all restrictions, must have at least 6 elements
        -- {from_id, from, via_id, via, to_id, to}
        IF (array_upper(restrictionRecord.members, 1) >= 6) THEN

            nrFrom := 0;
            nrTo := 0;

            -- look for type: from, via, to
            FOR ix IN 1..(array_length(restrictionRecord.members, 1)-1)
            LOOP
                IF restrictionRecord.members[ix+1] LIKE 'from' THEN
                    fromOsmId :=
                        trim(leading 'wn' from restrictionRecord.members[ix])::bigint;
                    nrFrom := nrFrom + 1;
                ELSIF restrictionRecord.members[ix+1] LIKE 'to' THEN
                    toOsmId :=
                        trim(leading 'wn' from restrictionRecord.members[ix])::bigint;
                    nrTo := nrTo + 1;
                ELSIF restrictionRecord.members[ix+1] LIKE 'via' THEN
                    viaText := viaText || restrictionRecord.members[ix] || ',';
                END IF;
            END LOOP;

            IF (nrFrom != 1 OR nrTo != 1) THEN
                CONTINUE;
            END IF;

            -- look for restriction type
            FOR ix IN 1..array_upper(restrictions, 1)
            LOOP
                IF (SELECT restrictions[ix] = ANY(restrictionRecord.tags)) THEN
                    restrType := restrictions[ix];
                    EXIT;
                END IF;
            END LOOP;

          -- find topology edge ids that might be affected
          -- each osm edge could have two topology edges (in and out at vertex)
          -- and there is no really easy way of finding who is who?
          FOR edgeId IN
              SELECT *
              FROM find_topo_edges_at_turning_restriction(
                  osm_edges_table,
                  fromOsmId,
                  toOsmId,
                  topo_edges_table)
              AS f(id integer)
          LOOP
              edges := array_append(edges, edgeId);
          END LOOP;

            -- store findings
            INSERT INTO turning_restrictions
                VALUES (fromOsmId, toOsmId, viaText, edges, restrType);
```

```
        nrFindings :=  nrFindings + 1;
      END IF;
    END LOOP;
    RETURN nrFindings;
END;
$$ LANGUAGE 'plpgsql';
```

D.13.4 LGU.style

```
# This is the default osm2pgsql .style file that comes with osm2pgsql.
#
# A .style file has 4 columns that define how OSM objects end up in tables in
# the database and what columns are created. It interacts with the command-line
# hstore options.
#
# Columns
# =======
#
# OsmType: This is either "node", "way" or "node,way" and indicates if this tag
# applies to nodes, ways, or both.
#
# Tag: The tag
#
# DataType: The type of the column to be created. Normally "text"
#
# Flags: Flags that indicate what table the OSM object is moved into.
#
# There are 5 possible flags. These flags are used both to indicate if a column
# should be created, and if ways with the tag are assumed to be areas. The area
# assumptions can be overridden with an area=yes/no tag
#
# polygon - Create a column for this tag, and objects the tag with are areas
#
# linear - Create a column for this tag
#
# phstore - Don't create a column for this tag, but objects with the tag are areas
#
# delete - Drop this tag completely and don't create a column for it. This also
# prevents the tag from being added to hstore columns
#
# nocache - Deprecated and does nothing
#
# If an object has a tag that indicates it is an area or has area=yes/1,
# osm2pgsql will try to turn it into an area. If it succeeds, it places it in
# the polygon table. If it fails (e.g. not a closed way) it places it in the
# line table.
#
# Nodes are never placed into the polygon or line table and are always placed in
# the point table.
#
# Hstore
# ======
#
# The options --hstore, --hstore-match-only, and --hstore-all interact with
# the .style file.
#
# With --hstore any tags without a column will be added to the hstore column.
```

```
# This will also cause all objects to be kept.
#
# With --hstore-match-only the behavior for tags is the same, but objects are
# only kept if they have a non-NULL value in one of the columns.
#
# With --hstore-all all tags are added to the hstore column unless they appear
# in the style file with a delete flag, causing duplication between the normal
# columns and the hstore column.
#
# Special database columns
# ========================
#
# There are some special database columns that if present in the .style file
# will be populated by osm2pgsql.
#
# These are
#
# z_order - datatype int4
#
# way_area - datatype real. The area of the way, in the units of the projection
# (e.g. square mercator meters). Only applies to areas
#
# osm_user - datatype text
# osm_uid - datatype integer
# osm_version - datatype integer
# osm_changeset - datatype integer
# osm_timestamp - datatype timestamptz(0).
# Used with the --extra-attributes option to include metadata in the database.
# If importing with both --hstore and --extra-attributes the meta-data will
# end up in the tags hstore column regardless of the style file.

# OsmType   Tag                 DataType     Flags
###################################################
node,way    access              text         linear
node,way    barrier             text         linear
node        crossing            text         linear
node,way    disused             text         linear
node,way    emergency           text         linear
node,way    highway             text         linear
node,way    incline             text         linear
way         junction            text         linear
way         lanes               text         linear
way         maxheight           text         linear
way         maxlength           text         linear
way         maxspeed            text         linear
way         minspeed            text         linear
way         maxweight           text         linear
way         maxwidth            text         linear
node,way    noexit              text         linear
way         oneway              text         linear
node,way    public_transport    text         linear
node,way    restriction         text         linear
node,way    railway             text         linear # :level_crossing, tram, tram_stop
way         surface             text         linear
node        toll                text         linear
way         tracktype           text         linear
node,way    traffic_calming     text         linear
```

```
node,way    traffic_sign        text         linear

# Access restrictions for vehicle types
node,way    goods               text         linear
node,way    hgv                 text         linear
node,way    lhv                 text         linear
node,way    motorcar            text         linear
node,way    motor_vehicle       text         linear
node,way    psv                 text         linear
node,way    vehicle             text         linear



# Deleted tags
# These are tags that are generally regarded as useless for most rendering.
# Most of them are from imports or intended as internal information for mappers
# Some of them are automatically deleted by editors.
# If you want some of them, perhaps for a debugging layer, just delete the lines.

# These tags are used by mappers to keep track of data.
# They aren't very useful for rendering.
node,way    note                text         delete
node,way    note:*              text         delete
node,way    source              text         delete
node,way    source_ref          text         delete
node,way    source:*            text         delete
node,way    attribution         text         delete
node,way    comment             text         delete
node,way    fixme               text         delete

# Tags generally dropped by editors, not otherwise covered
node,way    created_by          text         delete
node,way    odbl                text         delete
node,way    odbl:note           text         delete
node,way    SK53_bulk:load      text         delete

# Lots of import tags
# TIGER (US)
node,way    tiger:*             text         delete

# NHD (US)
# NHD has been converted every way imaginable
node,way    NHD:*               text         delete
node,way    nhd:*               text         delete

# GNIS (US)
node,way    gnis:*              text         delete

# Geobase (CA)
node,way    geobase:*           text         delete
# NHN (CA)
node,way    accuracy:meters     text         delete
node,way    sub_sea:type        text         delete
node,way    waterway:type       text         delete

# KSJ2 (JA)
# See also note:ja and source_ref above
```

```
node,way    KSJ2:*              text        delete
# Yahoo/ALPS (JA)
node,way    yh:*                text        delete

# osak (DK)
node,way    osak:*              text        delete

# kms (DK)
node,way    kms:*               text        delete

# ngbe (ES)
# See also note:es and source:file above
node,way    ngbe:*              text        delete

# naptan (UK)
node,way    naptan:*            text        delete

# Corine (CLC) (Europe)
node,way    CLC:*               text        delete

# misc
node,way    3dshapes:ggmodelk   text        delete
node,way    AND_nosr_r          text        delete
node,way    import              text        delete
node,way    it:fvg:*            text        delete
```

D.13.5 mikhailovsk.osm

The file used during testing was supplied by the company, and is an edited version. But it is not necessary to put 2.1 MiB of *xml* data here. A new file, containing more information, can be downloaded, see listing D.1:

```
$ wget -O mikhailovsk.osm "http://overpass-api.de/api/map?bbox=41.9491,45.0918,42.1151,45.173"
```

Listing D.1: Download Mikhailovsk map data.

D.13.6 partille.osm

The file used during testing was supplied by the company, and is an edited version. But it is not necessary to put 4.4 MiB of *xml* data here. A new file, containing more information, can be downloaded, see listing D.2:

```
$ wget -O partille.osm "http://overpass-api.de/api/map?bbox=12.0873,57.7168,12.1703,57.7475"
```

Listing D.2: Download Partille map data.

D.14 util

D.14.1 Logging.h

```
/* Use Boost loggging, and handle setup in this file.
 *
 * #include "Logging.h"
 *
 * Needs a lot of linking to work:
 * -lboost_log -lboost_log_setup -lboost_thread -lboost_system -lpthread
 *
```

```
 * @author  Jonas Bergman
 *
 */

#ifndef LGU_LOGGING_H_
#define LGU_LOGGING_H_

#include <boost/log/common.hpp>
#include <boost/log/core.hpp>
#include <boost/log/expressions.hpp>
#include <boost/log/sinks/text_file_backend.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/support/date_time.hpp>
#include <boost/log/utility/setup/file.hpp>
#include <boost/log/utility/setup/common_attributes.hpp>
#include <boost/log/trivial.hpp>

/** To simplify the set up of logging in the application: include this file
 * and call the 'initLogging()' function.
 */
struct Logging
{
    static void initLogging()
    {
        if(isInited)
        {
            return;
        }

        boost::log::add_file_log(
            boost::log::keywords::file_name = "lgu.log",
            boost::log::keywords::format = "[%TimeStamp%]: %Message%"
            );
        boost::log::core::get()->set_filter(
            boost::log::trivial::severity >= boost::log::trivial::info
        );

        isInited = true;
    }

    Logging() = delete;
    Logging(const Logging& from) = delete;

private:
    static bool isInited;
};

#endif /* LGU_LOGGING_H_ */
```

D.14.2 Logging.cc

```
/*
 * Logging.cc
 */

#include "Logging.h"

```

```
//static
bool Logging::isInited {false};
```

D.14.3 Point.h

```
/**  Data structure for Point.
 *
 * #include "Point.h"
 *
 * @author  Jonas Bergman
 */

#ifndef UTIL_POINT_H_
#define UTIL_POINT_H_

// SYSTEM INCLUDES
//
#include <ostream>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//

// FORWARD REFERENCES
//

struct Point
{
// ATTRIBUTES
    double     x {0.0};
    double     y {0.0};

    Point(double x, double y) : x(x), y(y) {}
    Point() = default;
    Point(const Point&) = default;

// OPERATORS
    friend std::ostream& operator<<(std::ostream& os, const Point& rPoint)
    {
        os << std::fixed << "Point [x: " << rPoint.x << ", y: " << rPoint.y << "]";
        return os;
    }

    bool    operator==(const Point& rhs) const
    {
        return (rhs.x == x) && (rhs.y == y);
    }
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

```

```
#endif /* UTIL_POINT_H_ */
```

D.14.4 TimeToStringMaker.h

```
/**  Static class to provide strings based on time.
 *
 * #include "TimeToStringMaker.h"
 *
 * @author  Jonas Bergman
 */

#ifndef UTIL_TIMETOSTRINGMAKER_H_
#define UTIL_TIMETOSTRINGMAKER_H_

// SYSTEM INCLUDES
//
#include <string>

// PROJECT INCLUDES
//

// LOCAL INCLUDES
//

// FORWARD REFERENCES
//

/**
 * Class who provide strings from times.
 */
class TimeToStringMaker
{
public:
// LIFECYCLE

    /** Default constructor.
     */
    TimeToStringMaker() = delete;

    /** Copy constructor */
    TimeToStringMaker(const TimeToStringMaker& from) = delete;

// OPERATORS
// OPERATIONS

    /** Get the current time as a string.
     *
     * @return    A string representation of the time.
     */
    static std::string        getEpochMsTimeString();

// ACCESS
// INQUIRY

protected:

private:
```

```
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif /* UTIL_TIMETOSTRINGMAKER_H_ */
```

D.14.5 TimeToStringMaker.cc

```
/*
 * TimeToStringMaker.cc
 *
 * @author  Jonas Bergman
 */


#include "TimeToStringMaker.h"  // class implemented

#include <chrono>

/////////////////////////////// PUBLIC ///////////////////////////////////////

//============================= LIFECYCLE ====================================

//============================= OPERATORS ====================================

//============================= OPERATIONS ===================================
//static
std::string
TimeToStringMaker::getEpochMsTimeString()
{
    using namespace std::chrono;
    milliseconds ms = duration_cast< milliseconds >(
        system_clock::now().time_since_epoch());
    return std::to_string(ms.count());
}

//============================= ACESS      ===================================
//============================= INQUIRY    ===================================
/////////////////////////////// PROTECTED  ///////////////////////////////////

/////////////////////////////// PRIVATE    ///////////////////////////////////
```