LiRA Map:

A Cloud-based Geo-Information System for Road Maintenance

Jonathan Drud Bendsen

25. May 2020

Supervised by: Ekkart Kindler

Technical University of Denmark

Table of Contents

Abstract

The purpose of this project is to develop a prototypical web application for the LiRA system capable of displaying road condition data collected by cars to the user. By analyzing example situations, and potential ways of modelling these in data, several key concepts were developed. They were used in combination with the LiRA system to develop the prototype of the LiRA Map cloud-based web application. This prototype was then evaluated and reflected upon, in order to investigate its potential as a proposed geographic information system. With the flexibility upon which it was designed, it may become a possible stepping stone towards a fully developed final product.

1. Introduction

The Live Road Assessment project, called *LiRA* for short, aims to develop a system used to assess the condition of Danish roads in order to help the maintenance process of said roads [1]. Currently, road conditions are assessed through manual inspection at longer time intervals. This comes with an operation cost, and also means lower data accuracy and resolution. LiRA aims at solving these problems by making use of sensors built into modern passenger cars. These sensors collect a lot of different data while traversing the roads, which can be collected and analyzed in order to assess the road's condition. By taking advantage of the many modern passenger cars already out in the country, manual assessment costs can effectively be removed and the data becomes much more accurate through the significantly higher update frequency.

The LiRA project consists of multiple separate components which all work together to solve the problem. This report focuses on a single one of these components: The LiRA Map Application. This application is a geographic information system (GIS) application [2] which serves the purpose of displaying the condition data collected by the cars and translated by a machine learning system. It will present the data overlaid on an easily navigable map, providing a quick overview of the state of roads in any area navigated to by an end-user, namely employees of road maintenance management organizations. The app aims to facilitate the maintenance planning process by providing end-users with an easily understood interface and allowing them to quickly locate what roads need repair next.

GIS applications for this purpose already exist, such as Rosy [3] and Vejman [4] by Sweco and the Danish Road Directorate respectively, as well as a previous version of the LiRA Map application made by DTU M.Sc. student Markus Berthold [5]. The LiRA Map app is built to be a non-commercial, open source alternative to these existing products, and to remove the need for third-party software for LiRA. The version built in this project is rebuilt from the ground up to be a *cloud-based web application* as opposed to the old version being a standalone application.

LiRA Map, along with the rest of the LiRA project, is made with Denmark and the Danish Road Directorate in mind. There is however no technical reason for this restriction, and as such the LiRA Map application design will not be limited to Denmark.

2. Problem Analysis

In order to develop LiRA Map in the best way possible, it is important to first analyze and understand the concept of *Road Maintenance and Assessment* and the problems therein. We can then develop a *domain model* which can be used to further the understanding of the information the application will work with.

Firstly we will look at an example situation of a small subset of roads with defects on them, and from there analyze what entities are at play and how they can be modelled in data. After understanding the types of situations LiRA Map works with, we will look at and draw useful aspects to reuse from existing software, including the previous iteration of LiRA Map, and use it to develop a domain model. Then we will look at application requirements, first non-functional requirements given by the project administrators, and then develop *user stories* to help define functional requirements and goals. By identifying the end users and how GIS tools within road maintenance planning are used, we can develop a graphical user interface layout that will streamline the usage of the application for the desired end users.

2.1 An Example Situation

In order to better introduce and understand the concepts discussed in this chapter, we will first examine an example of an imaginary situation with a small subset of roads which contain cracks, holes, or other road conditions that would need maintenance. We will analyze it, in order to learn what aspects and concepts are at play, and how to best model them for the use of this application.

Imagine a simple T-shaped intersection of two roads. Take for example the intersection of Anker Engelunds Vej and Lundtoftegårdsvej near DTU Lyngby campus. A map of this can be seen in Figure 1.

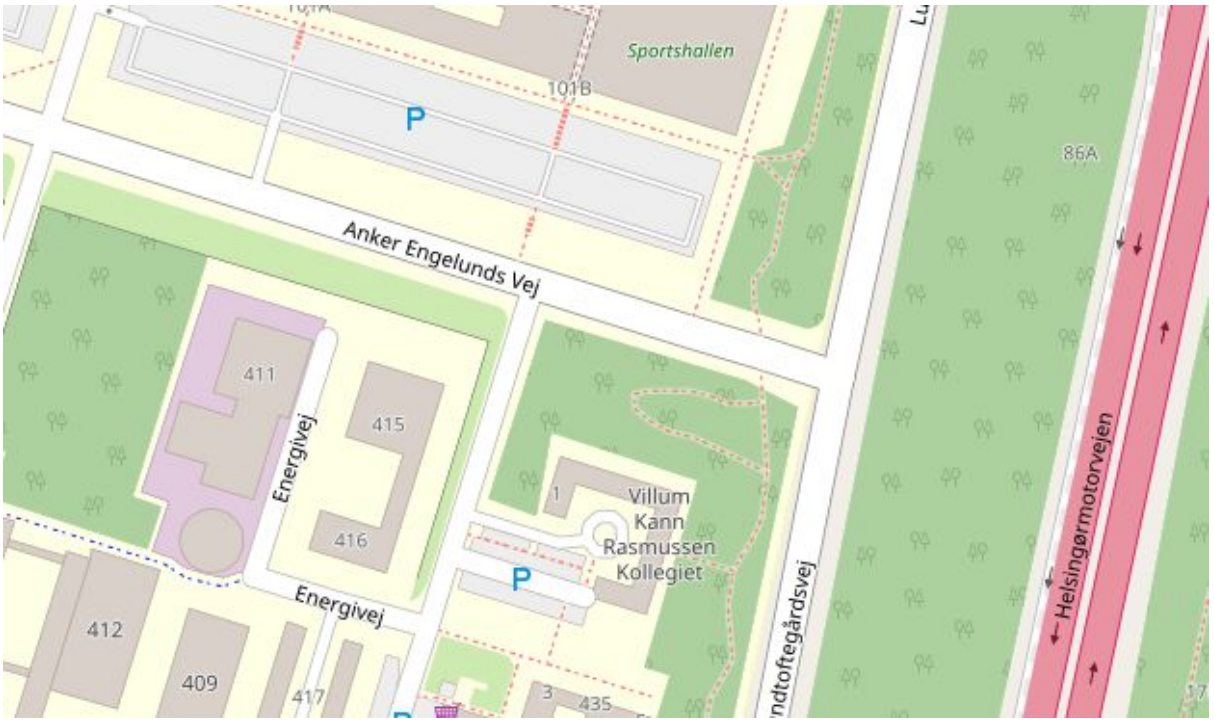


Figure 1: A map image of the intersection between Anker Engelunds Vej and Lundtoftegårdsvej near DTU Lyngby, from OpenStreetMap.

Roads deteriorate over time and will eventually need repair or maintenance to fix damages it may have accumulated since last it was repaired. The LiRA project aims to detect these defects through the use of sensors attached to everyday cars that cover most of the road network regularly. The data collected from these cars is sent to a central server which collect and compile the data through the use of *machine learning* to understand the condition of the road.

Assume sensors in the cars driving by the intersection in our example have collected data which the machine learning system has used, and from it identified that Lundtoftegårdsvej has cracks stretching past the intersection, starting at a given point and ending at another. Also assume that data from other cars led to the discovery of a pothole on Anker Engelunds Vej. With these assumptions, we can overlay the measured defects on the map above where the cars detected them, in order to visualize the state of the roads as shown on Figure 2.

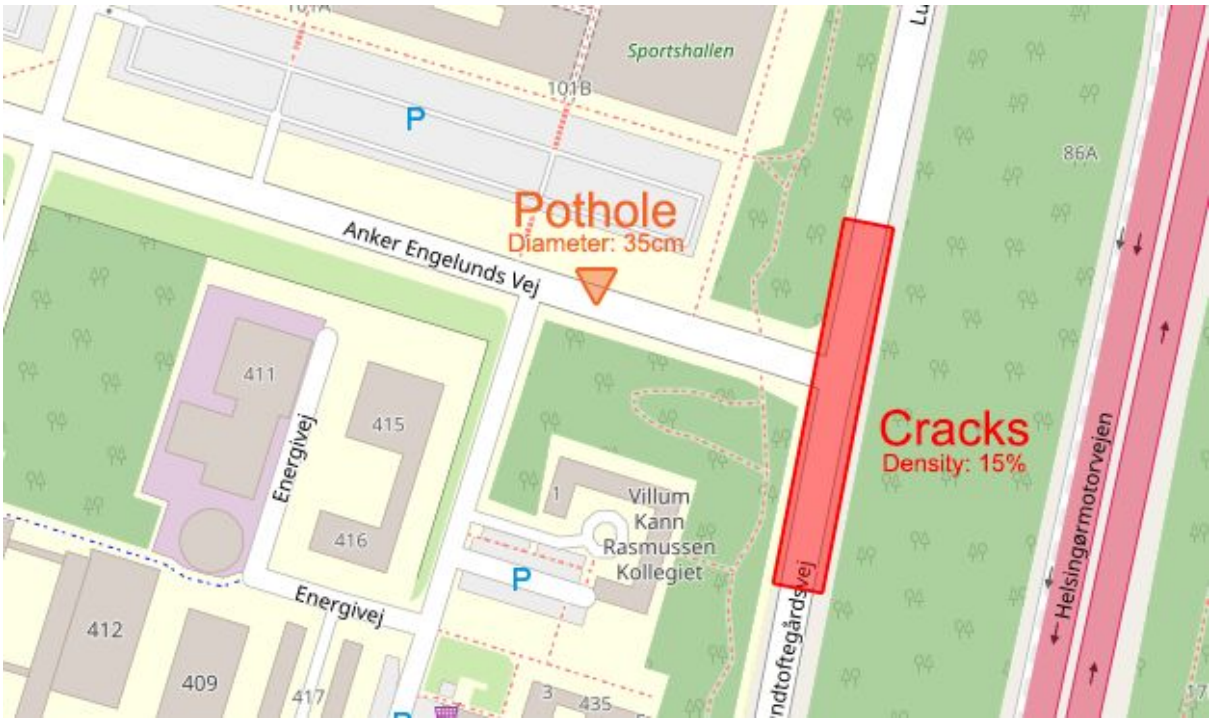


Figure 2: Example of measured conditions overlayed on top of Figure 1.

In this example, we can see a visual representation of a situation that could exist out in the world. We see the two measured defects, which we will call *road conditions*, and their position. We also see that road conditions can be described as one of two *kinds:* either *point* or *stretch*. The pothole is an example of a *point* condition as it identifies a defect at a single position on the road, while the cracks are an example of a *stretch* condition, stretching over a section of the road. We also see that each road condition can be associated a numerical value describing the severity or properties of the condition.

This illustrates what the LiRA Map application's objective is: To visualize conditions and their values of roads on a map.

2.2 Domain Analysis

Before we can move on to the implementation of the application, we must first analyze and identify the different parts that act in the situation above. The core concept of road assessment can be described as a process involving three main entities. These three entities are *roads*, *conditions*, and *measurements*, however *measurements* do not play an important role in the LiRA Map project.

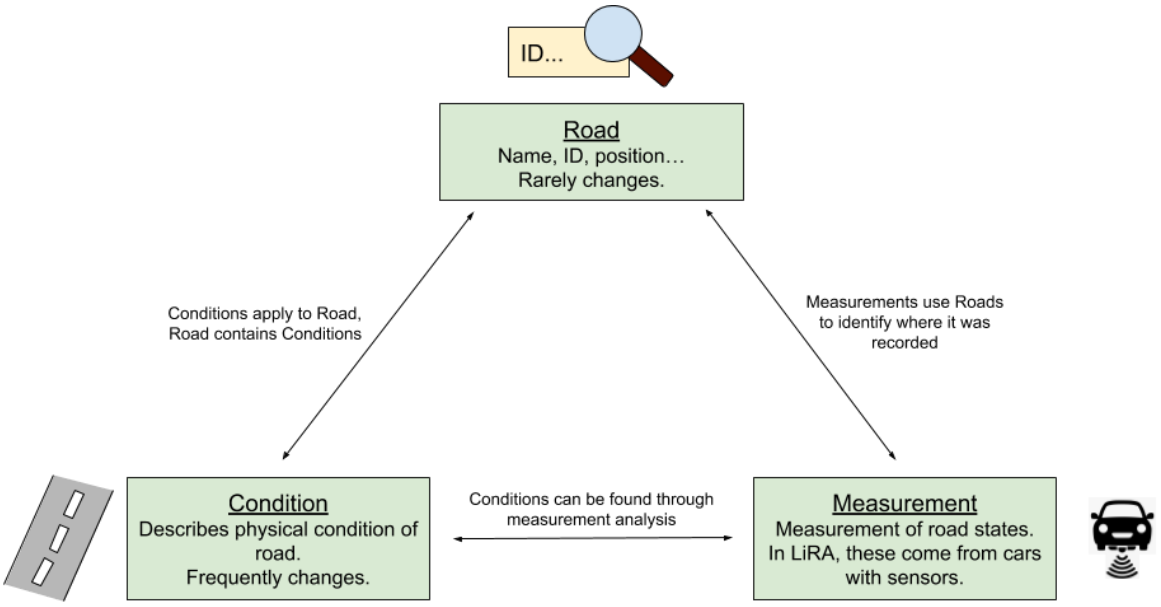


Figure 3: The three entities in road assessment through modern cars.

Figure 3 shows the three entities and how they interact in a road assessment context. In order to create a model that is better suited for application purposes, we see that the term *Road* does not refer to the physical construction of a road out in the world itself. Instead, it describes the abstract *identification* and properties of the road. This may include the name of the road, the geographical position, a possible code or identifier, information about speed limits, and other similar data. These properties rarely change. The actual *physical state* of the road however can change frequently, and is thus described by separate entity we call *Conditions*. Conditions are time-based data that describe information about a road or a section of one. It could be a defect on that part of the road, or it could be informational such as recently repaired. If a road has no conditions, it can be considered in perfect state. *Measurements* are the third entity in the process, and are the source from where road conditions can be derived. These could be created through manual inspection. In LiRA, the goal is to equip everyday passenger cars with sensors that can measure data about the road's state as they travel across it. This data is reported with a GPS-location and timestamp attached. LiRA uses its machine learning system to derive Conditions from the measurements, where the GPS-location and timestamp are used to match it to a Road and inform when it was recorded.

In the example seen in *Figure 2*, the two *Roads* in play are Anker Engelunds Vej and Lundtoftegårdsvej. Notice that the *identification* and positional data we defined "Road" as is enough to create a visual map of the roads, from which we can see sections of in Figure 2. The two *Conditions* seen are the cracks and the pothole which are positioned along the visible sections of the roads. We can see that the two conditions indicate two different types of defects. We will introduce a new term to describe these: *Condition Types*. In our example on Figure 2, the *Condition Types* of the two conditions are *cracks* and *pothole*.The *Measurements* are not seen in Figure 2, however they were used in the creation of the *Conditions* and their types and properties that do show.

While Measurements and cars are important to the LiRA project, they are in another part of the system than the LiRA Map project belongs to and are not relevant to the development of the application. Thus we will not mention them for the remainder of this report. Instead, LiRA Map will focus entirely on Roads and Conditions already derived.

2.2.1 LiRA Map 1.0

A previous version of the LiRA Map application was already made, and by drawing on how it modelled the domain, as well as through analyzing the design and structure of the resulting application, we may find and extract aspects that may be useful in the development of the domain model for this iteration.

This previous application, referred to as “LiRA Map 1.0” for the remainder of this report, was made by DTU M.Sc. student Markus Berthold in collaboration with the LiRA team as part of his Master’s thesis. While there were some small differences which will be made clear later in this section, the overall goal was the same: To create a geographic information system (GIS) application for the purposes of supporting maintenance planning focused on the condition data of roads from LiRA [5].

LiRA Map 1.0 extracts several important notions to reuse from its analysis of Rosy, a pavement management system developed by Sweco, and Vejman, a company tool which includes a pavement management system created by the Danish Road Directorate. Through the analysis of the domain model of those two systems, LiRA Map 1.0 borrows the concept of different *condition types*. These are known as defect types in Rosy, and represent different variations of conditions so that each condition’s type can be described through an enumeration. By extension, this also means each condition must be associated with exactly one condition type. LiRA Map 1.0 also reuses the *section* system Rosy uses. In LiRA Map 1.0, each road consists of a set of sub-sections. These individually contain information about the state of the section, and are also what conditions are relative to. The sections also reuse the notion of a *condition index*, also known as the *KPI* value, which describe the severity of the damages on a section as a whole, as an aggregation of all conditions and their severity found on the section. *Pavement types* is also a data point in the sections of LiRA Map 1.0 extracted from Rosy and Vejman. The roads and sections used by LiRA Map 1.0 are based on the ones in the databases of Rosy and Vejman, such that data and conditions end-users might already have can be more easily repurposed to LiRA Map.

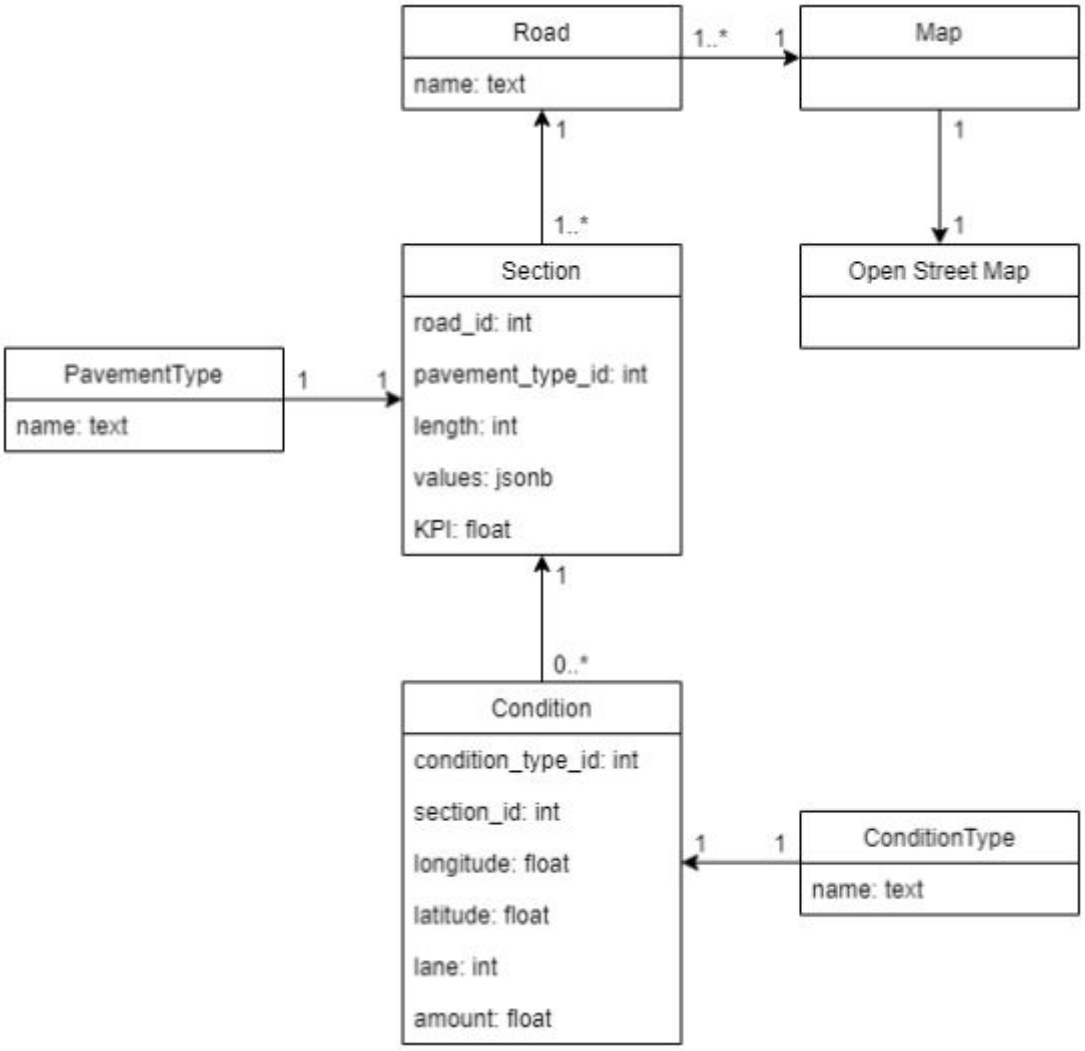


Figure 4: Domain model of LiRA Map 1.0

The LiRA Map 1.0 prototype was built as a standalone, offline application through the Windows Presentation Form (WPF) framework. The domain model used is seen in Figure 4. It loads road and condition information from a database, geographic road and section data from a shapefile, and displays these on a map whose tiles are based on Open Street Map (OSM). It displays the KPI value, the *condition index* previously discussed, by splitting the rendering of the selected road into a line for each section, each line being color scaled from green to red depending on its section's KPI value.

The evaluation of LiRA Map 1.0 provides insight to several shortcomings in the design and implementation of the application in regards to its usability for the targeted end user. For example, a lacking feature was the ability to select individual sections for closer inspection, instead of only being able to select the whole road at any given time. Likewise, it was also a concern that the app was to strict on sections. For example, a user might want to closely inspect a specific part of a section that may be affected by a specific condition, but LiRA Map 1.0 does not allow seeing conditions themselves, and instead only visualizes the average of a section. An example of this would be a section where half of it is in a pristine state while another half is covered by a severe condition. LiRA Map 1.0 would display this as the whole

section being yellow - the average of half pristine and half severe. Another desired evolution was to have the application move away from the dependency of a shapefile for the geometry of the roads, and instead work more directly with OSM which is already aware of roads on its own.

This iteration, LiRA Map 2.0 (referred to for the remainder of this report as just “LiRA Map” without the “1.0”), differs from the previous version in that it is an attempt to make a more flexible, more easily scalable and extendable application in the form of a *web/cloud-based application*. LiRA Map 1.0 can help greatly in the development of this web version. We will take inspiration from the structure and draw out useful notions to reuse from the domain model of LiRA Map 1.0, and we will keep the evaluation and shortcomings in mind as we design the new version of LiRA Map. This will be used later throughout the report.

2.2.2 Road Representation

A road as it will be referred to in this context is the *identity* of a physical road in the real world. This is the mostly static data that can be used to, for example, represent a road on a map, without taking into account its current physical state. This can be seen in Figure 1 and Figure 2. In order to fully represent such a road, specific properties must exist. These are a name, some unique ID, and geographical position and path data. Since road names are not necessarily unique, the names themselves cannot be used for identification, and therefore a unique ID is needed. Even though the geographical path data can be considered a description of the *physical* properties of the road, it can be elevated into the road representation, as the location is mostly static and unaffected by the *state* of the road. For example, if cracks in a road are repaired with an filling of new asphalt, the geographical position of the road remains the same.

It is a *non-functional requirement* of the LiRA Map application to use an external database for geographical road data and representation, specifically Open Street Map (See Section 2.4.1).

2.2.2.1 Open Street Map (OSM)

In order to make best use of Open Street Map for our project, it is important to first research and analyze the structure of the OSM system and how it represents roads, in order to find out how it can be used to fill the role of road representation as our project needs it.

Open Street Map, named OSM for short, is a crowdsourced and open source database for geographical data, including amongst other things, roads. The main site can be accessed on https://openstreetmap.org and displays a map generated by the data in their database.

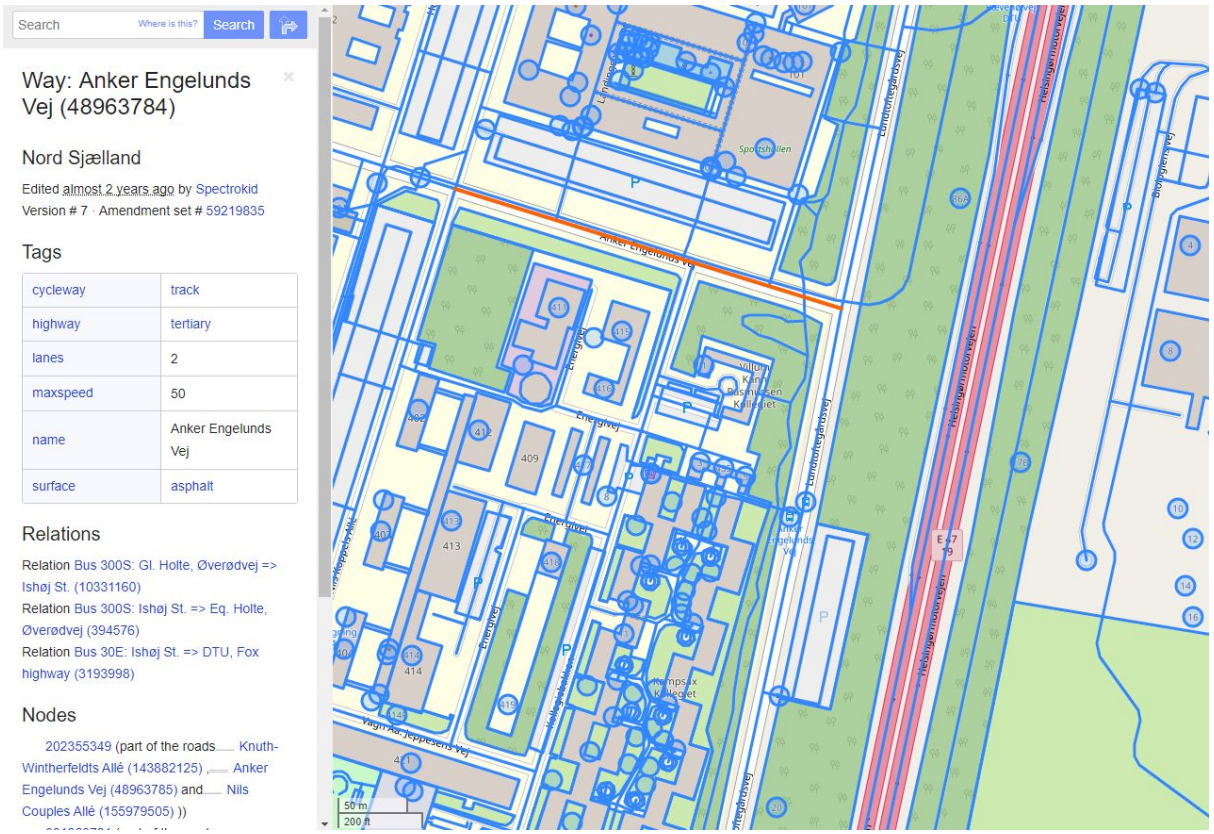


Figure 5: A screenshot of the main page of OpenStreetMap.org navigated to the same intersection as seen in Figure 1 and 2, and with Map Data turned on.

Figure 5 shows a screenshot of the main page and visualizes the extent of their data collection. Each blue line and circle represents an object in the OSM database at its associated geographical location on the map. The orange line is a section of Anker Engelunds Vej and is currently selected by the user. The menu on the left displays the data associated with this section ("Way" in OSM). This is the same area as the example in Section 2.1 shows (Figure 1 and Figure 2).

The database of OSM is built up from four different basic types of data:

- **Nodes**
  Nodes represent a geographical point on the map. Each node has a latitude and longitude coordinate. These are commonly used in Ways, but may also be single point data, such as trees or lamp posts.
- **Ways**
  Ways represent a polyline. They are defined by a series of nodes referenced by their ID in order, and may both be an open line or a closed polygon. These are used to describe shapes, the exact representation given by the Way's tags. Examples of shapes are roads, rivers, buildings, or even non-physical shapes such as boundaries and borders.

- **Relations**
  Relations are multi-purpose data structures that can contain two or more other elements (other relations included) in an ordered list. These often describe how multiple elements work together. An example of this could be bus routes, where the relation contains the nodes and ways ordered such that the nodes represent bus stops in the order the bus travels. Another example could be turn restrictions, where the relation tells that from the first way onto the second way, you may only turn a certain way. Tags describe what exactly the relation means, and what properties it has.
- **Tags**
  Tags can apply to any of the three other types and define a key/value-pair of data. This can for example be the name of a road, the meaning of the element (such as a way being a river or road), or address number. There cannot be two tags of the same key on any given data object.

Nodes, ways, and relations are considered *elements* in the OSM data structure, and each have their own unique ID. Tags are part of any given *element* and describe properties of that element, giving meaning. In Figure 5, every blue circle is a node. Every line, including the selected orange one, is a way. Relations and tags are not shown on the map, but can be seen in the menu on the side for the selected (orange) way. On top of that, every point in a way's line is also a node. These aren't shown on the map as the way is shown instead, but they can be seen listed in the menu as well.

Note that Open Street Map does *not* contain any data structure that represents a *complete road* as we would imagine them out in the real world. We may expect any OSM way to correlate to a single whole real-world road (i.e. with the same name), but this is not the case in the OSM database. Instead, a whole road may be split into multiple ways in OSM, such that each individual way can have their own tags. While a whole road can then be defined by a *set of ways*, there are no widely-adopted relation data types to indicate which ways are bound to the same street, as the relation type for this purpose has still not been widely adopted past its *proposed* state [6].

2.2.2.2 Overpass API

There exists many external APIs that make use of the OSM database, which allow users to access the data in various different ways and formats [7]. Some provide front-end rendering of the data onto maps similar to OSM's own page, others provide access to grouped data dumps, and some provide a way to query the OSM data through more advanced filters for use in other applications. Overpass is one of such APIs, and is what we will be using to query OSM data in the LiRA Map app.

Overpass is a read-only API that allows access to the OSM databases [8]. As opposed to the main OSM API, Overpass provides much more flexibility towards the criteria and filters to search for data. It allows searches such as by area, by name or tag, by proximity or radius, and much more. On top of having such flexibility in its querying capabilities, it also supports getting the combination, the difference, or intersection of separate queries as sets.

More information and detailed examples of the capabilities of the Overpass API can be found on the Overpass examples page from the OSM wiki [9]. The Overpass API is useful for the LiRA Map application as it allows us to filter out ways that represent other things than roads, such as rivers and boundaries, to allow us to only get roads within a requested bounding box.

2.2.3 Condition Representation

Conditions are the core of the project. They are modelled from the data collected by the cars, and are what describes the state of the road. The main purpose of the LiRA Map application developed in this project is to display these, and make it easy for the user to visualize the extent of the current conditions on a map or in a list view.

We can reuse the notion of different *condition types* from LiRA Map 1.0 as well as having a numerical value describing its severity. In the example seen in Figure 2, the *types* would be pothole and cracks, the numerical value would be the diameter and amount respectively. We can also reuse the idea of describing stretches by a start- and end position. If the start and end positions are the same, we can consider it a *single point* condition. With this, we can differentiate between the two kinds *Point Condition*, and *Stretch Condition*. A point condition can now be defined by just a single position. In the example, the “Pothole” condition is a *point*, and the “Cracks” is a single *stretch*.

Figure 2 (Repeat): Example of measured conditions overlayed on top of Figure 1.

We see that condition positions are relative to a specific road - for example in Figure 2, the pothole exists on Anker Engelunds Vej, a certain amount of meters in. The cracks apply to Lundtoftegårdsvej, and has two specific relative positions indicating start and end. We can

take inspiration from the idea of describing a condition's position by pointing at a road and providing relative start- and end positions along that road from LiRA Map 1.0.

Every condition has a type and a numerical value. The available types of conditions can be freely defined based on need. Each type has their own interpretation of the numerical severity value. For example, for a Pothole, its numerical value could be interpreted as the diameter of the hole. Cracks consider it a relative amount of crack density. These interpretations, and in extension the scale of which the numerical severity operates, can be adjusted at will.

2.2.4 Representation Refinements

The current definitions of roads and conditions are very similar to that of LiRA Map 1.0, and reuses a lot from how that system works. However, some issues arise in certain types of road structures. Take for example the section of a road seen in Figure 6.

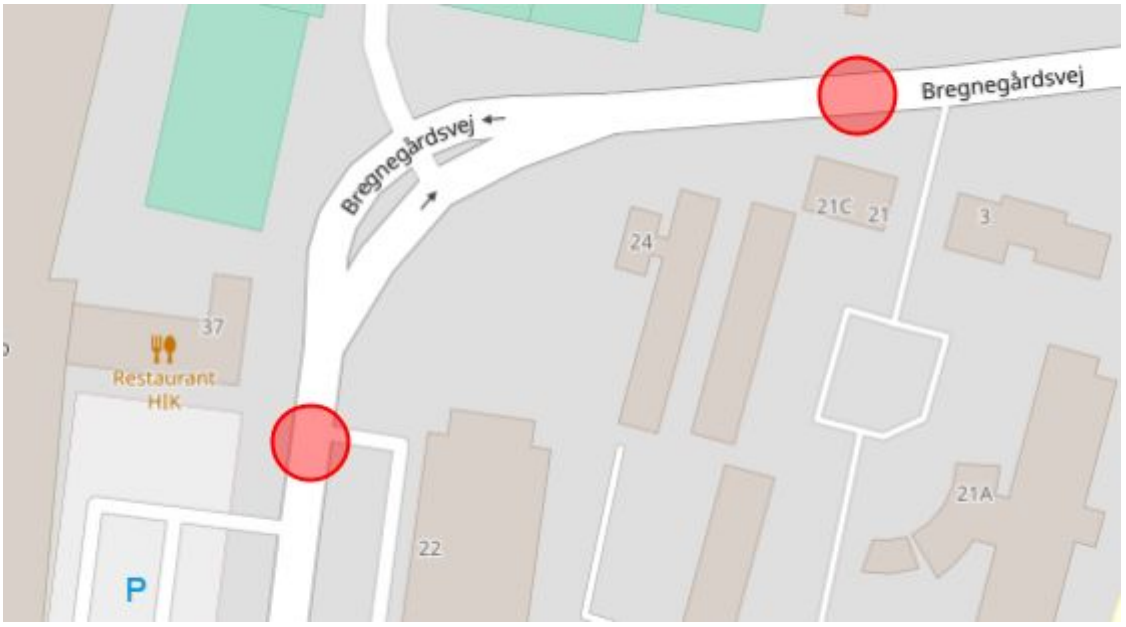


Figure 6: An example of a splitting road, from OSM. Overlaid red circles to represent condition start- and end position.

In this example we see a road that splits into two separated lanes through a turn. Assume a condition was detected on this road, with its relative start- and end positions where the red circles mark. Where does this condition stretch across? Does it move along the upper lane, the lower lane, or both? We see that a single start- and end position relative to a whole road is not enough information to accurately describe where the condition exists on its own. LiRA Map 1.0 currently solves this by using a lane number to identify what lane the condition stretches across. This however does not allow the condition to stretch across *both* lanes.

It is also possible that a condition may extend across *multiple* roads. There is no physical restrictions that prevent the same defect from stretching across the boundary of two named roads. It is also entirely possible that a defect may extend in multiple *directions*. Consider another example. This time, imagine the same scenario as seen in Figure 2, but where the

cracks extend across the intersection onto Anker Engelunds Vej. This can be seen in Figure 7.

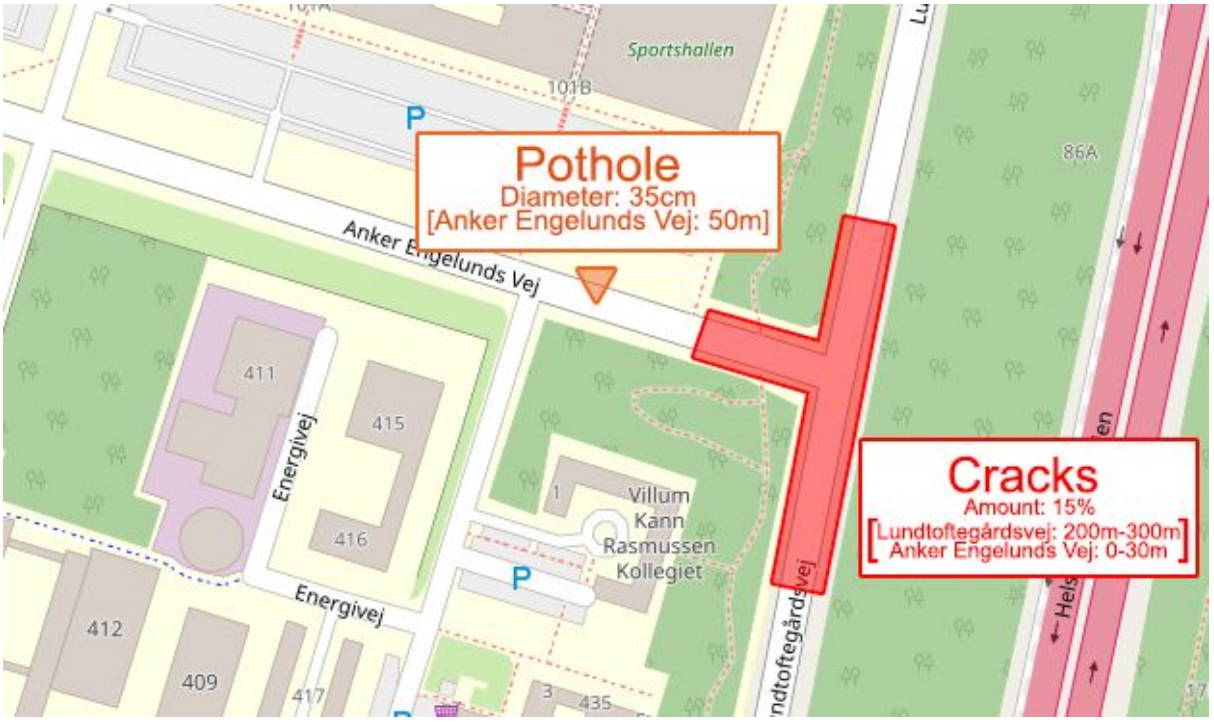


Figure 7: An example of a stretch condition stretching in multiple directions in an intersection. Note that all numbers are example numbers and may not be accurate.

We see that the cracks in the road extend in multiple directions, independently of road identities. It is thus also learned that a stretch condition cannot be fully defined by two single points.

A possible solution to this problem, is to make conditions relative to *sections* of a road, and allowing multiple to be covered, each with their own relative start- and end positions. By ensuring sections are split such that they are always *linear* parts of roads, we will never run into these issues on a per-section level. As such, we refine our definition of a condition as containing a *set* of *road sections* that are covered, rather than a single whole road, each with their own start- and end positions relative to the respective section. Conditions are considered *point conditions* if the set of covered sections only contains 1 section, and it only contains a start position.

This differs from the modelling in LiRA Map 1.0, where conditions always only pointed at one road and one lane. If the example in Figure 6 and Figure 7 had to be modelled by this, the stretch conditions would have to be split up into two separate condition objects.

This refinement of the definition of a condition means conditions are no longer basing on roads, but are instead basing on sections of roads. As we source our road data from OSM, these sections are the equivalent of the ways in OSM that together make up the road. As OSM ways are defined as an ordered set of nodes, they are guaranteed to be linear. By rebasing conditions like this, we can choose to make ways the *baseline* for the system. This

means throwing away the concept of a *road* entirely, and working directly on ways instead, treating each way as their own individual road. We can accept this, as for the purposes of the LiRA Map project, grouping ways into sets defining a whole road does not matter. Road and condition rendering would remain the same. Through discussing with LiRA project members, basing directly on ways was found to even be better, as the map-matching algorithm used to match measurements to roads also work with OSM ways. This meant that conditions could just be trivially assigned directly to these ways without any road lookup process. On top of that, one of the problems in LiRA Map 1.0 was the inability to select and inspect the individual sections rather than whole roads. By breaking down the representation of roads into the smaller sections that is OSM Ways, the system will inherently work with more detailed selections.

As such, we choose to redefine *road representation* to be based on individual *OSM ways*. We will refer to them as “way” for the remainder of this report. Following this, conditions now apply to a set of *ways*.

2.3 Domain Model

Knowing the representation models of the ways and conditions, as well as where their data is sourced from and how they connect, a *domain model* can be developed. Assuming sourcing map data from Open Street Map, the domain model is seen in Figure 8.

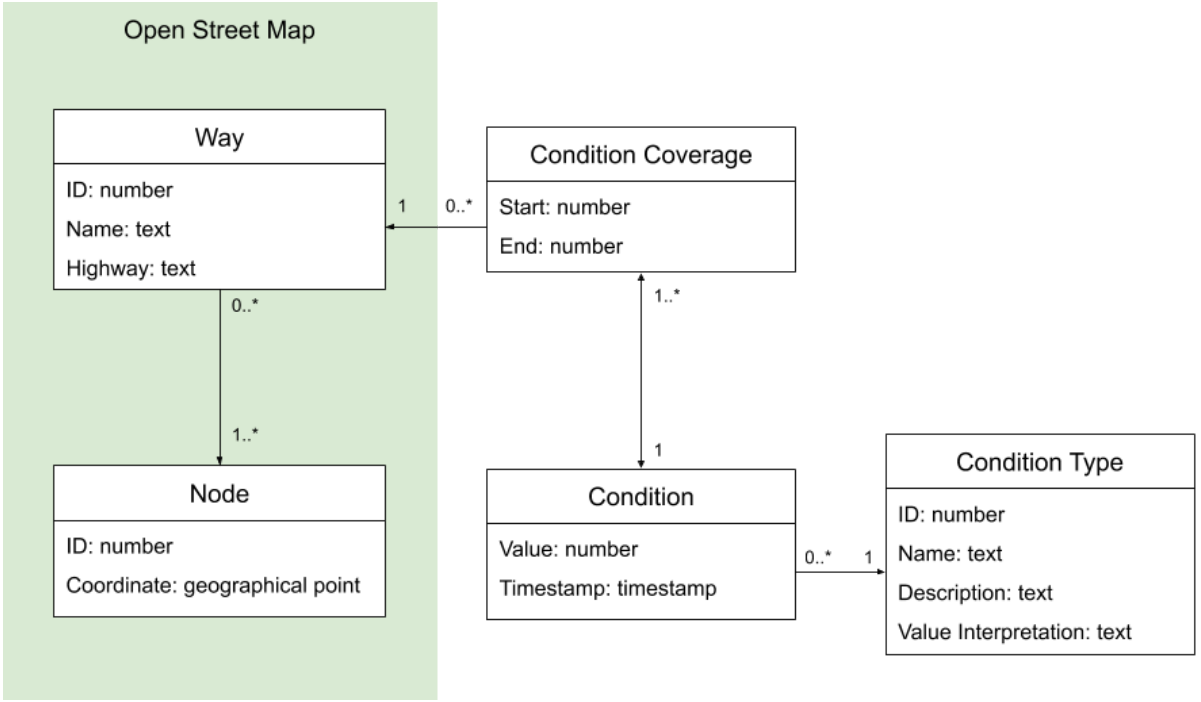


Figure 8: The domain model of the delimited domain of the LiRA Map project.

Figure 8 shows one possible interpretation of the domain, and the model representing it. This model is based on using *ways* as the baseline for road representation. It would be possible to model the domain retaining the concept of whole roads as a set of ways, however as discussed in Section 2.2.4, this is not necessary for the LiRA Map project.

Condition Coverage is the *navigation object* between conditions and ways. This is a type of object often used in relational databases to link sets of objects together. In this case, it represents the many-to-many relationship between conditions and ways. A condition may apply to many ways, and a way may have many conditions. It is enriched with the start- and end positions of the exact condition it belongs to, on the exact way it points to. This enables every condition to have individual start- and end positions for each way it covers.

The *Name* and *Highway* fields are based on tags in OSM. The *Highway* tag is especially important, as it is the tag that indicates this way represents a section of a path. The value of this tag indicates what type of path it is. Examples of this are *highway, motorway*, or *footpath*. The Highway tag is used when filtering what ways are relevant to LiRA Map, i.e. which ways represent actual sections of road, and thus which ways will actually exist in the application.

If we overlay the domain model in Figure 8 to the example seen in Figure 7, it will look as seen in Figure 9.

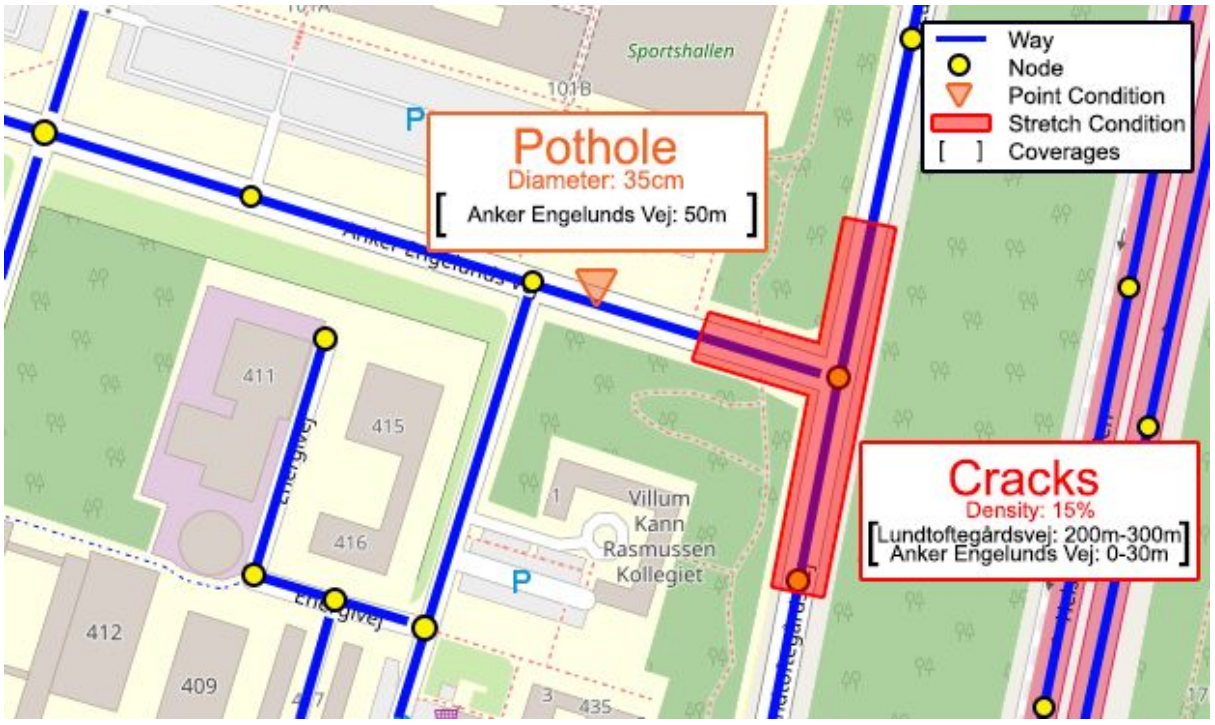


Figure 9: Example overlaid with labels correlating each element to their respective model in the domain model in Figure 8.

In Figure 9 we see the elements of the domain model applied to the example of Figure 7. Each way is shown as a blue line connecting an ordered set of nodes together. Each node is represented as a yellow circle. These may be shared by multiple ways. Intersections where ways end are represented by the blue line not fully passing through the node. For example, Anker Engelunds Vej can be seen not fully connecting the node at the intersection with Lundtoftegårdsvej. This indicates Anker Engelunds Vej ends at this node. We see the same two conditions as in Figure 7, and we see that the stretch condition in particular covers two individual ways in its coverage set. We also see that the pothole only covers a single way, and that this way only contains a single distance, meaning the condition is a point condition.

This illustrates how the LiRA Map application will model its data.

2.4 Application Requirements

Now that we have defined the domain we are working in, the next step to get a complete idea of what the purpose of this project is, is to determine the requirements the final application should aim to complete. This will give a better idea of how the system should be designed, as well as help understanding the directions the implementation should go in.

2.4.1 Non-Functional Requirements

First and foremost, the project has a small set of *non-functional requirements* defined by the project description and the project leaders themselves. These requirements are:

- **Using OpenStreetMap for Map Data**
  Previous iterations of the LiRA Map project have used closed-source databases for map data. This had the problems that the data would have to be acquired by the end user of the LiRA Map application in order to get any functionality from it. The LiRA team would like to move to an open-source, crowd-sourced database, and specifically chose OSM for this.

- **Application to be Open Source**
  The application and system of LiRA is aimed to be open source, which includes the LiRA Map application. This is a big part of the reason LiRA Map exists; to develop an open source version of a geographic information system instead of the many close source ones already in existence. This is also part of the reason why OSM is desired.

- **Application to be Cloud-Based**
  LiRA Map is aimed to be a web application in order to allow remote accessing and distribution, as well as be more easily scaled for performance. It should work with a front end only tasked with showing the UI and the data, while the back end is responsible for getting that data. Due to this architecture, it easily allows the back end to be modified in how it gets and translates the data, without having to change the front end.

- **Application to be easily Extendable**
  Large parts of the program should be coded in such a way, that modifying individual parts would be easy, without having to dig deep into the core code. This means the application should, as much as possible, be modular and have different services responsible for different actions. For example OSM may be desired now, but it may be changed in the future.

These requirements form the core of the application's purpose. Upon these, the application will be built so that it always adheres to the requirements put forth by the project leaders. Because of these, the LiRA Map app developed in this project will be a **web application** that uses **OpenStreetMap** as its foundation.

2.4.2 User Stories

The purpose of user stories is to imagine different situations the end user would like to be able to perform, so that we can optimize the flow of the application to make these as easy and intuitive as possible. These user stories are the result of our own analysis and build *on top* of the core requirements above.

Before the user stories can be properly defined, we must first look into who the actual *end user* is.

The targeted end-user for the LiRA Map application is the employees of the Danish state and government, specifically the sector for public infrastructure. More specifically, it is the employees of the Danish Road Directorate, who monitor and oversee the construction and maintenance of roads across most of Denmark. Generalized, the *end user* can be defined as the *employees of the entity responsible for maintenance of roads within the area of usage,* the area being Denmark in this case, but could later be any other country or local community. Within these organizations, the specific end-users are the staff responsible for maintenance planning, and the managing staff responsible for maintaining condition data.

With this in mind, we can develop user stories based on these positions in the road maintenance entity.

User Story 1: As an end-user, I want to be able to get an overview and easily navigate road conditions in a certain area through a 2D map representation that accurately reflects the physical world, so that I can get an overview of road states by area.

User Story 2: As an end-user, I want to be able to see conditions on the 2D map with a visual indication of their numerical value, such that I can more easily get an idea of which defects exist where and how bad they are.

User Story 3: As an end-user, I want to be able to easily view detailed information about a specific condition, including its severity, type, date of recording, and which roads it applies to, in order to further understand the condition in question.

User Story 4: As an end-user, I want to be able to easily view detailed information about a specific way, including all conditions that apply to it, in order to further understand the state about that specific way.

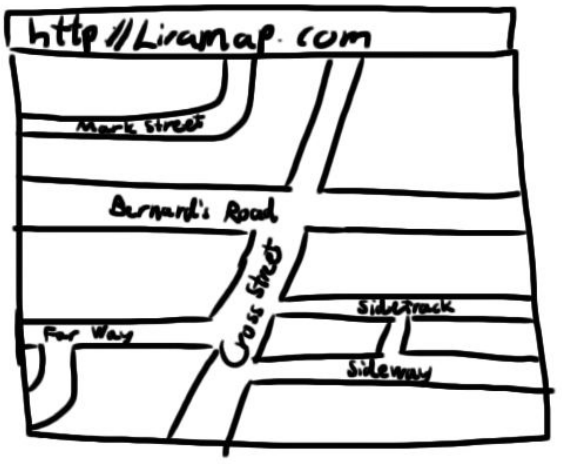
User Story 5: As an end-user, I want to be able to quickly find a specific way by name or other identification, in order to quickly check the status of a specific way in mind.

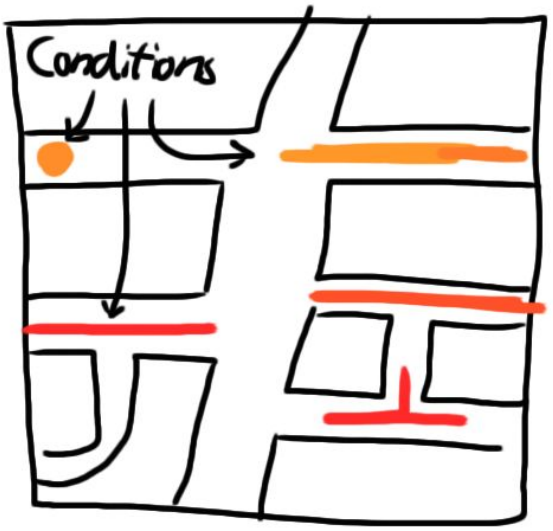
User Story 6: As an end-user, I want to be able to be able to view all roads in a list-like view, which can be sorted both by names or IDs of roads, as well as any given condition type, in order to facilitate quickly finding the state of a specific way or set of ways if I already know them by name rather than position.

2.4.3 Desired Features

Through the User Stories, a list of desired features can be derived. These can be interpreted as the acceptance criteria of the various stories. By fulfilling these features while adhering to the non-functional requirements, the program can be considered a viable prototype.

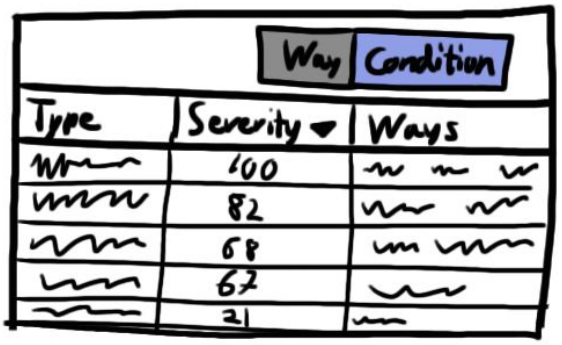
Some features listed here were not yet implemented in the resulting prototype by the end of this project. These have been marked by an asterisk (*) in the header. See Section 5.2 for details about these missing features.

A map-based application	
The application should have its main page consist of a large map, upon which all the roads within view are displayed. The map should be easily navigable, allowing the user to pan and zoom with intuitive controls. The map should base itself on readily available map resources such that it is accurate to the physical world, even when no conditions, ways, or other LiRA-based data are overlaid on top of it. Acceptance criteria of **User Story 1**.	 *Figure 10: Sketch of map-based web page.*

Condition Displaying on Map	
Conditions should be drawn on top of the map wherever they exist. Their color should be determined by how severe the condition is through some scale based on its type and numerical value, giving an easy way to get an overview of the severity of each at a glance. Zooming out and seeing clusters of colors give a quick and easy overview of whole areas and their states. Point conditions and stretch conditions could be displayed differently, such as lines for stretches, and dots or markers for points. Acceptance criteria of **User Story 2**.	 *Figure 11: Sketch of a map with overlaid conditions. Color coded from orange to red.*

Details Panel & Road/Condition Selection	
Ways should also be drawn on the map, underneath conditions. When any road or condition is clicked, it is selected and a panel displaying all the details about the selected entity should be easily available. For conditions, this should display the data and information collected and compiled into the condition. It should also display a list of all roads covered by this condition. For roads, this should display metadata about the road, and display a list of all conditions found on this road. Acceptance criteria of **User Story 3 & 4**.	 *Figure 12: Sketch of roads drawn on the map, with a selected road and the details panel on the side.*

Road Search Bar*	
A search bar should be readily available on the map view, allowing the user to quickly search for any way by name and quickly jump to it on the map. Acceptance criteria of **User Story 5**.	 *Figure 13: Sketch of a search bar with "Cross Stre" written into it. A dropdown of matching roads can be seen below.*

Roads List View*	
Secondary to the map view, the application should have a page or a full screen panel that displays all (known) roads and conditions in a structured list. This list could potentially be set to be able to base both on conditions and roads by the user's selection, either having each row associated to a road and be displayed with a list of associated conditions, or make each row correlate to a single condition. The list could be sortable by any given column. Acceptance criteria of **User Story 6**.	 *Figure 14: Sketch a list view listing all conditions sorted by Severity.*

2.4.4 Layout

In order to effectively develop an application that can quickly perform the tasks desired from the Desired Features, designing a complete layout for the application focused on these tasks will help give an idea of what the final product could look like, and help develop it towards being streamlined towards the tasks desired. Using the sketches from Desired Features, a possible layout can be designed as follows:

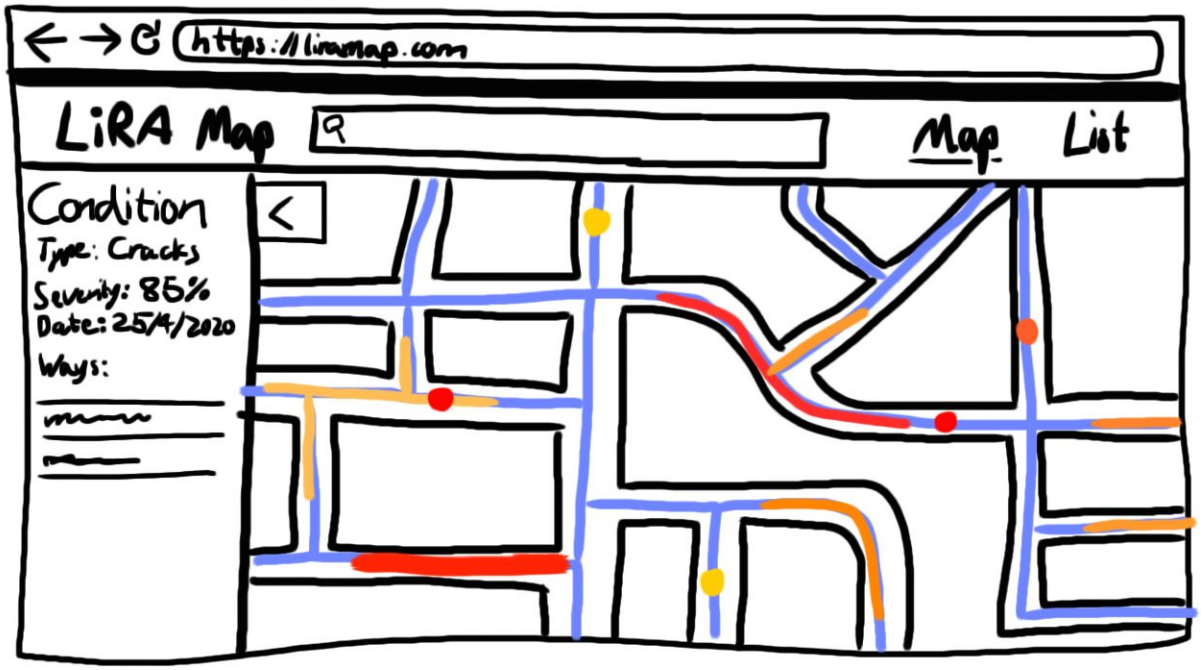


Figure 15: A sketch of a possible layout for the complete website on the Map page. The red condition bottom-left is selected by the user.

In this layout, a navigation bar exists across the top of the screen which remains on all pages of the application. We see that the Map page is currently viewed, but you can quickly change to the List page in the top-right corner on the navigation bar. To the left is the logo for the application, and in the center the search bar can be seen. Below the navigation bar is the application's main view. For the List page, this will be the same as seen in Figure 14. In Figure 15 we see the Map view being displayed, along with a network of roads and some conditions. To the left of the map is the details panel which currently lists the details of the red condition marked by a thicker stroke at the bottom-left of the map. We can also see that this panel can be collapsed by the "<" button sticking out from it.

This layout quickly gives access to the details at any given time, as the details panel will attach to the screen's edge rather than the moving map. The map takes the largest portion of the screen space, allowing greater opportunity to get an overview of larger areas. The details panel being able to be collapsed also allows an even greater view of the map. The map is navigated with intuitive drag-and-drop mouse controls, and mouse wheel to zoom.

The application's main view is the Map, and the map will be the default page opened first. The List view can be selected in the top corner of the navigation bar, and will replace the map and details panel with a full-screen list as seen in Figure 14. This list will display all known conditions and roads in a way that can be sorted by name, severity, or any other column of the user's choosing. This can be helpful to quickly find the worst of a specified type of condition, or to provide an overview of a large amount of ways in a table-like fashion.

3. Design

Before implementing the application, designing how it should function and what the objects and modules should be and do will greatly help providing a better understanding and making the program more clean and easy to extend, as it was desired by the application requirements. Taking into account the analysis performed in Section 2, we will now design how the architecture and flow of the program should work.

3.1 Flow & Communication Model

Since LiRA Map will be a *cloud-based web application*, it is important to take into account how the server will communicate with the client, and what type of data it will retrieve and return to the client in the response. The importance is only furthered by the desire to use OSM for map data, an external source, as well as the desire for the various parts of the program to be extendable or replaceable with other sources. For example, future developers may want to move to a local database for road data rather than OSM. As such, the design of the system should be modular so that each module can be replaced with one that uses another source.

The LiRA Map application is designed around the two objects *Way* and *Condition*, representing the objects in the domain model by the same name respectively. The *Way* object represents an OSM way, however only the values deemed necessary for LiRA Map. Each of these two objects are created by their own module, allowing easy replacement with another implementation of the same module.

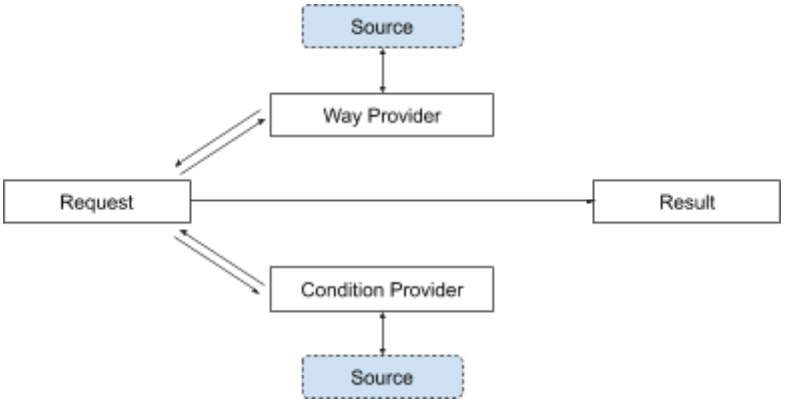


Figure 16: A data flow diagram describing the flow of a client request through the two provider modules and to the returned result.

Figure 16 shows a diagram of how a client HTTP request is treated. For example, if the client requests data for a specific bounding box (e.g. the client's map view), an HTTP request is sent from the client to the server at a specific API endpoint designed for this purpose. This then goes through both the Way Provider and Condition Provider modules, each of which communicate with their own source. The result is compiled and returned to the client.

The order of which the two modules are called upon depends on the design of the request endpoint. For example, in order to get all data within a bounding box, we must first find all ways that exist within this geographical box, and then afterwards ask the Condition Provider for all conditions that apply to these ways. Another example might be to find all potholes over a certain size. In this case, the design of the other API endpoint responsible for this type of request will be such that it first asks the Condition Provider to find all pothole conditions with a value over the specified size, and then ask the Way Provider to get all ways that these conditions point at by a set of IDs.

These two modules are what provide the flexibility and extensibility desired, as discussed in Section 2.4.1. How they internally work is not relevant to the rest of the system, meaning they can easily be replaced at any point in time with another module that for example uses a different source for its data. For example, if the LiRA project wishes to move away from OSM and to another map data source in the future, the Way Provider module could simply be replaced by another module that communicates with this new source instead. The same applies to the Condition Provider.

In the problem analysis, we determined that for now OSM was desired to be used as the source for ways, and we discussed the usage of the Overpass API as a well-suited API, as it allows great flexibility to the queries rather than OSM's "get everything" API. Since Conditions are the core of LiRA and are sourced from LiRA's own system, these will be retrieved from a local database. Implementing these as modules following the chart in Figure 16 provides the extensibility desired for potential future development of the LiRA Map application.

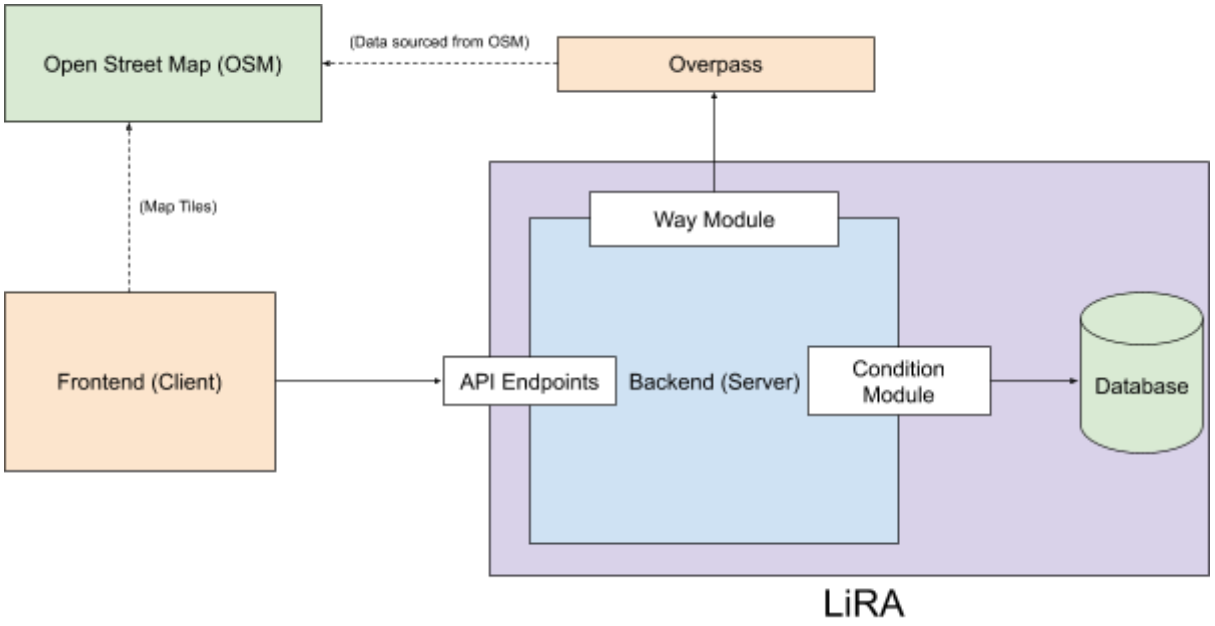


Figure 17: Communication model of the designed system, using Overpass and a local database for the ways and conditions respectively.

On Figure 17 we see the design of the communication following the design discussed up until now. We see that the frontend is unaware of the sources from which the backend retrieves its data from. It would have alternatively been possible for the frontend itself to query OSM for the map data, however by putting that responsibility on the backend we can easily replace the source completely, without affecting the frontend or how it operates.

The purple box denotes the LiRA servers. This is the internal back-end of the application and the servers on which it is deployed. As the diagram shows, the API endpoints are exposed to the world wide web. All other parts remain internal. The Way Provider module communicates with Overpass which exists outside LiRA, but is still internal to the system. The client directly communicates with OSM for the tiles used in the map rendered to the client.

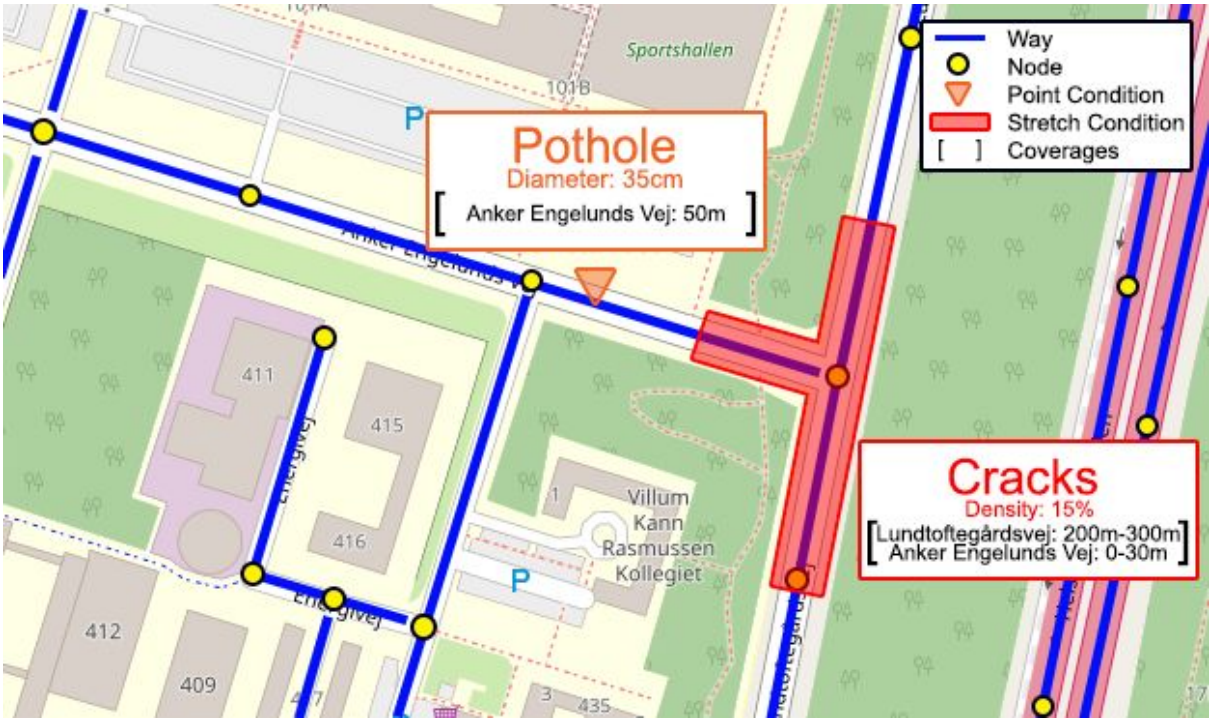


Figure 9 (Repeat): Example overlaid with labels correlating each element to their respective model in the domain model in Figure 8.

We can use Figure 9 as an example of a possible outcome a client might want to see in the LiRA Map application. The client will request an API endpoint on the backend for bounding box search, providing the bounds of this area. The backend then calls upon the way module which independently creates an API request to the Overpass API as seen in Figure 17. The backend itself does not care how the way module retrieves its ways, it merely awaits the module and expects a resulting list of ways and nodes. These are what appears as the blue lines and yellow circles in Figure 9. To then get the conditions, the backend will use the resulting ways' IDs to query the condition module for all conditions that point at these way IDs. Much like the way module, the backend does not care how the condition module handles this request, it just awaits a set of condition objects. These are the data values seen in the boxes next to the crack and pothole conditions.

After all of these have all been retrieved, the complete set of ways, nodes and conditions is sent back to the client. The frontend may use this response in any way it sees fit, for example rendering these on a map similarly to Figure 9.

This describes the lifecycle of an API request in LiRA Map, and the communication between the client, server, Overpass and OSM.

The API of LiRA Map will be designed as a RESTful (also known as REST) API. This means the API is stateless, and does not retain any state across different requests. This design pattern is chosen for its simplicity to understand and to implement. Because of this, the endpoints must be implemented in such a way that the request contains all necessary information to be understood by the server, making the state of the application stored entirely client-side.

3.2 The Map View

As determined in Section 2.4, a map page will function as the main page of LiRA Map. The tiles of the map (i.e. the images that make up the map) is rendered completely independently of the backend, with the frontend sourcing them directly from OSM. The ways and conditions that will be rendered as an overlay on top of the map must however be networked from the LiRA servers to the client. It would not be feasible to just network all ways and conditions to the client on page load, as that could potentially be a very large download size, and much of it would may be seen by the user. Instead, we must look into and determine how the application should handle networking only parts at a time, such that it never networks more than what is necessary to the client, and only does so on-demand.

We want LiRA Map to only network data needed by the specific state of the individual client. For the map page, it is only necessary to network data that is within view on the map. As such, the client application should only request data that exists within the bounding box of the current user’s view on the map. Similarly, when the user moves the map, we want the client application to request data that exists within the newly moved bounding box. The application should never request any geographical areas that the user never views.

Figure 18: Moving the view of the map. The transparent boxes represent two different views at two different times.

On Figure 18, the blue and orange transparent boxes represent two different views before and after a move respectively. The arrow represents the move. In the case of the example in Figure 18, the application should request all ways and conditions intersecting the blue box at first, and then request all ways and conditions in the orange box after the move.

We will design an endpoint in the LiRA Map API that gets all ways within a geographical bounding box, together with all nodes that these need, as well as all conditions that apply to any of the ways. The endpoint will take the latitude and longitude that make up the bounds of the requested box. These bounds will then be used by the backend to query the way module, which in this case passes them on to the Overpass API in order to find all ways intersecting these bounds. Overpass will also be asked to get nodes that are part of the resulting ways. Using the OSM IDs of the ways, the backend will then query the condition module to find all conditions that cover any of these. The resulting nodes, ways, and conditions are compiled into the response returned back to the client. This conforms to the design requirements of REST, as the endpoint is entirely independent of any state of the client.

3.2.1 Optimization

While the above stated design might work fine on its own, it is greatly unoptimized and will often make unnecessary requests or receive already-known data. In order to increase scalability and make the application more responsive, we may want to look into ways to optimize the design of the map view's networking.

A quick optimization would be to only perform the request once the map has *finished* moving after a user has interacted with it, rather than continuously perform requests as the map view changes. In the example of Figure 18, this means only the blue and orange boxes will be requested. A side effect of this is that the map will not render any new overlaid information as the user actively moves the map around. This is decided to be an acceptable tradeoff.

We can also see in Figure 18 that a new view of the map may overlap with an old one (as indicated by the two boxes overlapping). The ways inside the overlap were already retrieved during the first request and is unnecessary to retrieve in the second. The overlap could potentially be large depending on the zoom level of the view and how much it overlaps previous requests. We can optimize the networking by remembering what areas have already been searched in previous requests, in order to exclude data the client has already received. This can be done in one of two ways. Either we can implement an algorithm that takes a box and a set of holes and divides the box into as few boxes as possible that fill it out without intersecting any of the holes. This will result in splitting the orange box in Figure 18 into two smaller boxes, representing their own individual requests avoiding the overlap. Figure 19 demonstrates this approach on the example in Figure 18.

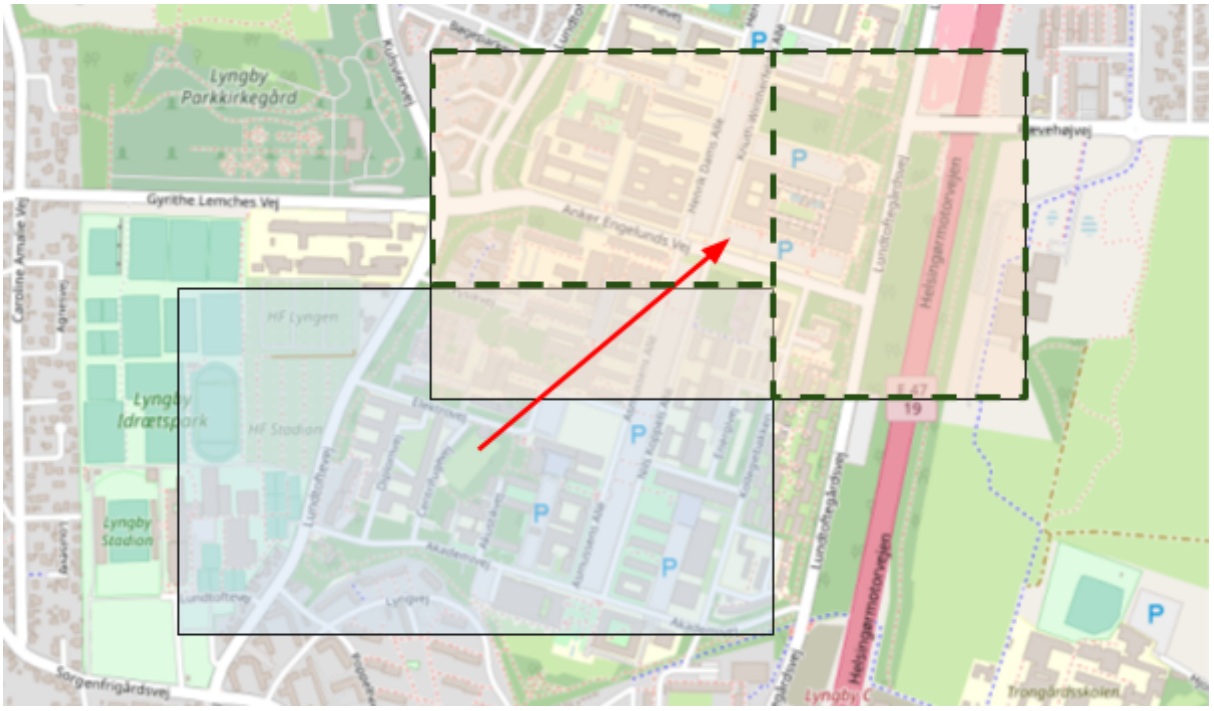


Figure 19: Demonstrating possible split of the orange box from Figure 18 into two smaller boxes that avoid the overlap, indicated by green dashed lines.

The other option is to make use of the Difference operator in the Overpass API. By passing all previously searched boxes that intersect the requested one, Overpass can subtract all ways that intersect any of these from the resulting set of the main requested one. In Figure 18 and AA, this would be taking the resulting set of ways from the full orange box, and subtracting the resulting set of ways from the full blue box. This method is chosen as it avoids having to code an advanced algorithm and is easier to implement.

Because the API is RESTful, the server must not store the history of searches for a client, in order to know what previous searches to subtract. Instead, the client caches these upon requesting. They will be stored in an R-Tree, which is a data structure that can be efficiently used for spatial indexing of intersecting rectangles. When a new request is started, the client application first searches this tree for all intersecting past searches, and then attaches them to the request together with the requested bounds. The server receives all of these and uses them in its query to Overpass.

The requests can be further optimized by removing all past searches that are entirely contained within a new search from the R-tree, as these are now unnecessary. For example, if the user zooms out causing a request of a larger area, the previously zoomed in search does not have to be remembered anymore as it is entirely covered by the zoomed out one. Furthermore, the application can avoid making requests altogether if the requested view is entirely contained within any previous search, for example when zooming *in* or navigating the map to a location that has already been entirely requested.

This describes how the main page of the LiRA Map application will function.

3.3 System Architecture

The web application is chosen to be implemented using the ASP.NET framework with the React template which uses React as the frontend framework. This is chosen due to its native support in the Visual Studio and .NET environments, and because other parts of the LiRA project are also implemented in C#, which ASP.NET uses, making it easier to connect it all together at the final stages of the project. The database is chosen to be implemented using PostgreSQL, a popular open source tool with a wide range of resources and documentation, and Entity Framework, an open source Object Relational Mapper (ORM) developed by the .NET Foundation, is used to connect the application to the database due to its easy integration and wide support for ASP.NET, as well as allowing code-first database modelling. Both of these are also chosen due to being the chosen systems in other parts of the LiRA project and recommended by LiRA project members for easier integration.

The frontend is implemented as a Single-Page Application (SPA) through the React framework. This framework is chosen due to its widespread popularity, providing greater resources and being more likely to be understood by a potential future developer. The map itself is implemented using **Leaflet** [10], an easy-to-use and also widely popular map library that is greatly optimized and also works on mobile. LiRA Map relies on being able to draw overlays atop a map in order to visualize ways and conditions, and Leaflet has many built-in features for these purposes, including drawing lines or markers, grouping overlays together in layers, as well as flexibly allowing sourcing tiles from any given source.

As the frontend of LiRA Map is developed as an SPA, the architecture is designed around a central object instance, which we will refer to as the LiRA instance, that exists across the entire web application. This object acts as the connection between the UI and the backend, and is what handles requesting and storing of data therefrom.

3.3.1 Back-End

The backend is a C# .NET application in the ASP.NET framework. It is designed to be relatively simple in itself, relying on the two modules discussed in Section 3.1. The API endpoints will be implemented through ASP.NET controllers, and it is through the functions which the controllers bind the incoming requests to that the modules are called upon. The modules are injected as *services*, following the standard ASP.NET design convention.

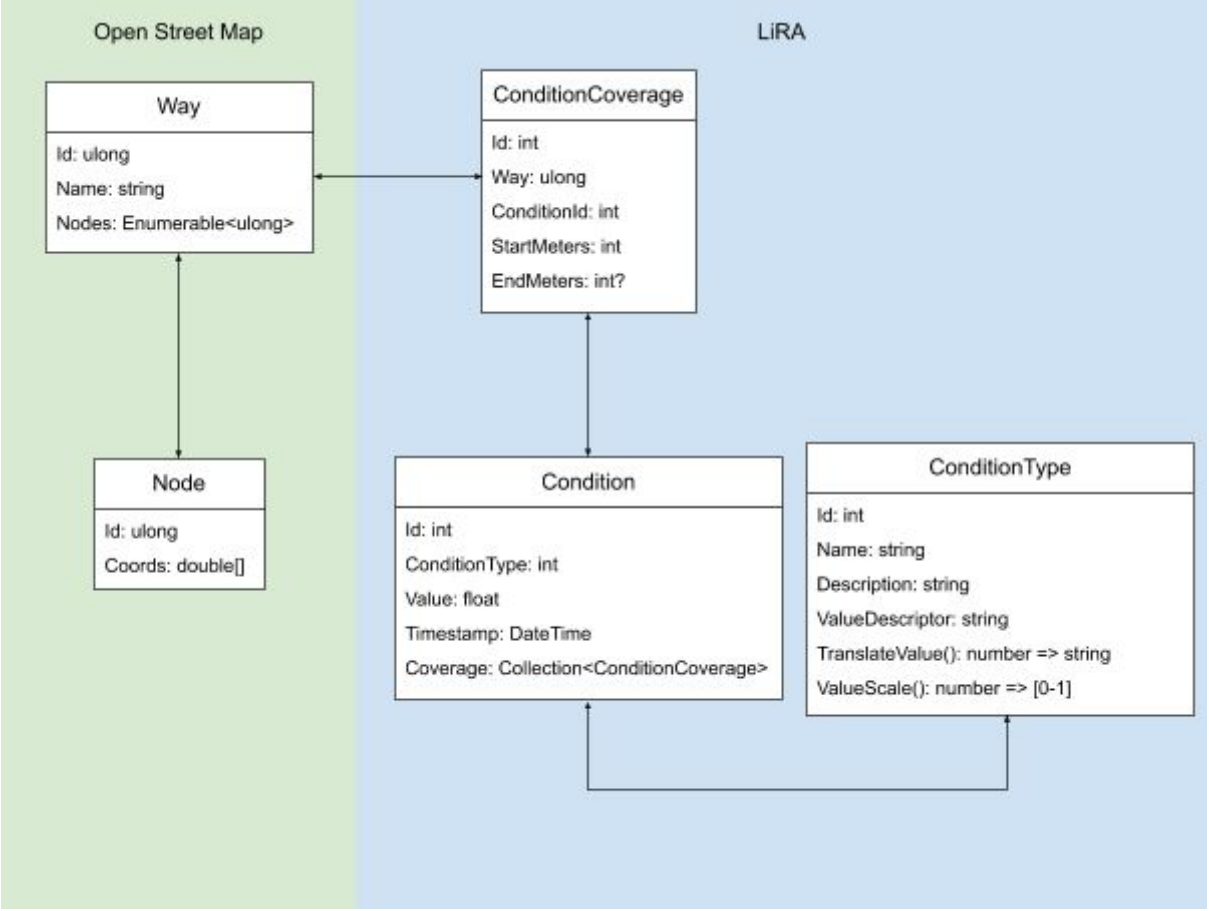


Figure 20: Object diagram displaying the object models used by LiRA Map and where they are stored. Their positioning is based on the domain model in Figure 7.

The backend of the LiRA Map application focuses around the four main models: *Way, Node, Condition* and *ConditionCoverage*. These are based on the definitions discussed in Section 2.3. They can be seen in Figure 20. Note that while Way and Node are stored in OSM, the models shown by their respective boxes represent the LiRA-side models generated by the OSM data. ConditionType also appears in Figure 20 and is stored in LiRA, but is not handled by the backend as it is instead implemented directly into the files sent to the frontend (See Section 3.3.2.2).

The Way and Node models are generated from the response from Overpass whenever a query is received. These model the ways as defined in Chapter 2. The Way model consist in base of a name, an ID given by OSM, and an ordered list of node IDs. Any extension can

however be implemented by the specific implementation of the way module in use. Nodes are the only object that actually contain any geographical data. As opposed to Ways, these are not designed to be extendable with data, however it could be possible with a larger change to the system (See Section 5.3). Nodes contain only an ID defined by OSM, and a geographical coordinate defined in latitude and longitude as a pair of doubles.

Condition is the model of a condition as defined in Chapter 2. These objects are generated from the database through Entity Framework whenever the condition provider module is requested to retrieve all conditions on a certain way or set of ways. They contain the number representing the value of the condition, along with the set of ConditionCoverage objects linking it to the ways it covers as described in Section 2.2.4. They also contain a timestamp. The condition type is stored as an integer pointing to the ID of the associated type.

3.3.1.1 Database

Because of how relational databases work, *navigation objects* are necessary in order to represent many-to-many relationships. These are objects stored only to act as links between every individual connection, using foreign keys to point at the two objects. They can often be auto-generated by database tools. In our case, *ConditionCoverage* is the explicitly defined navigation object between conditions and ways. It links a single condition to a single way, with the ConditionId being the foreign key to the condition. The “Way” number is however not a foreign key, as the ways do not exist in the database. As many ConditionCoverages may point at the same condition, as well as many may contain the same way ID, it enables the many-to-many relationship.

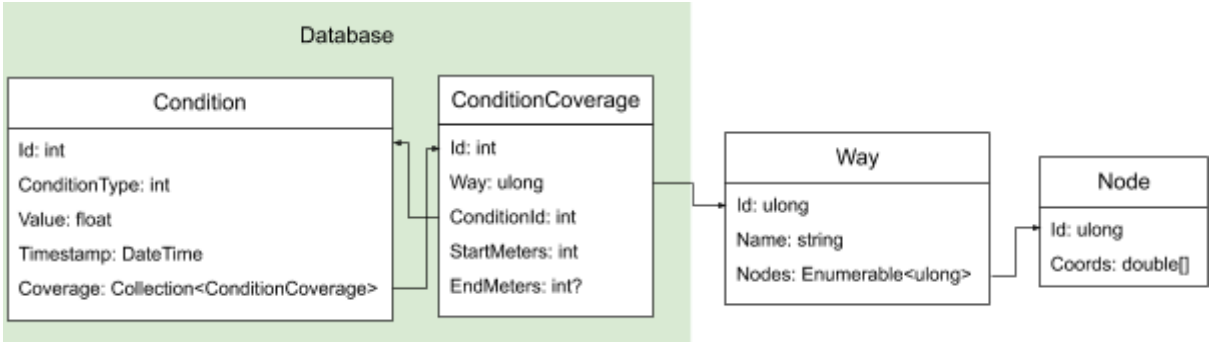


Figure 21: Relational model of the four main classes.

Figure 21 shows the design of how these classes are built, and what properties bind them together through relations. Only Condition and Condition Coverage exists in the database, while Way and Node are retrieved through OSM and only exist in runtime.

In order to query for all conditions on a specific way or set of ways, we can query on the table of ConditionCoverages instead of Conditions. By making the *Way* value an index in the ConditionCoverage object, the table can be effectively searched for all ConditionCoverages retaining to a way in question. These can then be used to traverse to the associated Condition(s) using the foreign keys. Using this querying process, the application process described in Section 3.2 can be carried out efficiently.

3.3.1.2 Modules

As decided in earlier sections of the report, the extensibility of the LiRA Map application relies on the design and implementation of the system in a modular fashion. In Section 3.1, we determined that the two basic modules are the Way Provider, and the Condition Provider. In order to make these truly modular, and making the rest of the system independent on the actual implementation of these, we will design the system architecture such that it uses *interfaces* for these two module types.

Way Provider

The Way Provider module is responsible for retrieving Ways and Nodes. We include Nodes under the definition of a Way Provider, as a potential use case for an implementation could be based on data that in itself does not rely on Nodes. This particular implementation would just generate its own Nodes from whatever data format the Ways are sourced.

The Way Provider should be capable of two main functions: Finding specific ways by a single or set of IDs, and finding ways by bounding box. Because of the design decisions made in Section 3.2.1, we rely on the backend (and Overpass) to do the already-searched subtraction. This means the Way Provider should be able to take an additional (potentially empty) set of already-searched bounds received from the client to subtract in its bounds function.

The interface design comes out to be as follows:

"IWayProvider" Interface Design		
Description	*Input*	*Output*
Retrieve ways and nodes by bounds.	*4 doubles* Bounds of requested box described in latitude and longitude.	*Ways, Nodes* Ways within requested bounds, and all nodes used by these.
Retrieve ways and nodes by bounds, and set of already-searched bounds.	*4 doubles* *List of <4 doubles>* Bounds of requested box described in latitude and longitude. List of boxes to exclude from result, also described in latitude and longitude.	*Ways, Nodes* Ways within requested bounds that are not also within any excluded box. All nodes used by the resulting set of ways.
Retrieve ways and nodes by way ID.	*ulong* OSM ID of the requested way.	*Way, Nodes* The way by the specified OSM ID, and all nodes used by it.
Retrieve ways and nodes by set of way IDs.	*List of <ulong>* List of OSM IDs for all ways requested.	*Ways, Nodes* Ways by the specified IDs, and all nodes used by these.

Condition Provider

The Condition Provider module is responsible for getting Conditions. The exact implementation is not important, instead it is only important what types of inputs it can receive and what results to expect. For the map view, the only necessary function is getting the list of conditions from a single way or a set of ways.

For the list view as discussed in Section 2.4.3, it becomes important to also be able to retrieve conditions by their severity and condition type. As the list view would be page-based, it also becomes important to support some implementation of pagination, allowing page-based queries to be performed. For example, a user may want to retrieve the conditions sorted by their numerical value, but then navigate to Page 2, upon which the Condition Provider must provide the next set of conditions following the previous.

In order to fulfill these purposes, the Condition Provider interface is designed as such:

"IConditionProvider" Interface Design		
Description	*Input*	*Output*
Retrieve conditions by way	*Way* Way to retrieve conditions on.	*Conditions* All conditions on the specified way.
Retrieve conditions by list of ways.	*List of <Way>* List of ways requested to find conditions on.	*Conditions* All conditions on any of the specified ways.
Retrieve conditions sorted by field, and some pagination indication.	*Field specification* *Pagination* Specification of what field to sort by. Pagination indicating what page to retrieve.	*Conditions* Conditions sorted by the specified field, the subset specified by the pagination indication.
Retrieve conditions sorted by field, and some pagination indication, with filtering.	*Field specification* *Pagination* *Filter* Specification of what field to sort by. Pagination indicating what page to retrieve. Some filtering indication.	*Conditions* Conditions sorted by the specified field that pass the specified filter. Subset specified by the pagination indication.

The pagination implementation could for example be based on a max amount of items on a page. Having the client specify how many items it can fit on a page (for example dependent on user's application window size) together with which page to request, the backend could multiply these together and skip to this point, before retrieving this amount of conditions following that point. The exact implementation of the pagination system can be changed to whatever fits the list view implementation best.

Both the Way Provider and Condition Provider interfaces can be extended if any new ways of navigating the LiRA Map application is desired. In this case, all module implementations must conform to the new version of the interface. This guarantees the application will always be able to perform its function, regardless of which source the ways and conditions are generated from, be it *map view*, *list view*, or even a potential new third view.

3.3.2 Front-End

The frontend architecture is designed around the central LiRA instance introduced at the start of Section 3.3. The LiRA instance is the container for all ways and conditions currently known to the client, and provides reading accessibility to these. However it is designed to only be a reflection of what data exists in the backend and nothing more; neither it, nor the ways and conditions it contains, store any map- or UI-related objects, such as lines or markers. The responsibility for these are instead shifted over to the individual instance of the map or UI component they belong to. This design means that the ways and conditions are in essence independent from the UI, allowing the UI to independently manage its own state without interfering with other UI components that may use the same data. The structure of the LiRA instance also allows the sharing of data across different UIs, such as for example the list view, being able to use ways and conditions discovered by the map without any further requests, and vice-versa.

The LiRA instance also handles communication with the LiRA API, and any UI component can call the instance to request data. The instance will itself handle whether to actually send an HTTP request or not based on past requests and known data. This is used in the implementation of the optimizations discussed in Section 3.2.1.

For the version of LiRA Map developed in this project, these tiles will be sourced from OSM as discussed in Section 3.1. As the both the map tiles are sourced from OSM just like the ways, the geographical overlays will also natively match the background tiles the application intends to display them on.

3.3.2.1 Classes

To provide a flexible and concise handling of the data representing ways and conditions, these two are designed their own classes. The main purpose of these classes beside being containers of the properties and fields, is to add drawing-related functionality such that the classes can be called upon to generate the visual layers that will represent themselves to be shown on the map screen. This includes generating the path or marker for the Leaflet map, providing color, and providing information for the details panel. All this are, as discussed above, not stored as data in the class, but are instead functions meant to facilitate the generation of such, returned to whichever component may call the functions, after which it is expected to manage its own state.

By using class hierarchy, it becomes possible to create subclasses that can override draw implementation, such as the *PointCondition* class (See below). This makes it easy to add different types of conditions, or even entirely new geographical objects in the future.

We will briefly go through the design of the main classes used by the frontend application.

Way

The Way class provides representation of a way. It contains the necessary information about the way, including the ordered list of Node IDs that make up the way's path. It also contains a list of references to all Conditions that apply to it. Its drawing function is designed to generate a PolyLine whose points are the coordinates of the ordered list of Nodes.

Conditions

Conditions represent a condition. It contains its type, numerical value, a timestamp, and the list of ways it covers along with their respective start- and end-meters. Its draw function generates a MultiLine made up of a line for each way it covers. These sub-lines are calculated as the positions of the start- and end-meters relative to the way, and every ordered node between. Their coloring function handles scaling the color with its numerical severity, relative to the condition type.

Point Conditions

Point Conditions are an extension of Conditions. It only overrides the draw function. This override will instead now generate a single marker at the position of the condition.

While Point Conditions are still identified as Conditions in the backend, in the frontend it has its own class. This class is constructed instead of Condition if the incoming data of a condition has its coverage only containing one covered way, and the end-meters of this is not defined.

Nodes

Nodes do not have an associated class, and are instead stored as an array of two doubles.

3.3.2.2 Condition Types

Condition types are objects that identify what type a condition is. They are important in order to display details about the condition to the user, as well as to provide understanding for what numerical scale each type considers when determining how good or bad the condition is.

The different types are stored statically in the Condition class, and are objects with the following structure:

Condition Type		
Name	`String`	The display name of the condition
Description	`String`	The display description of the condition
ValueDescriptor	`String`	A string representing how to interpret the value of the condition. This could for example be “Amount” or “Size”, and would as descriptors for the value.
TranslateValue	`number => string`	A function that takes the numerical value as an input, and outputs a string to display the value. This is primarily intended to add units, for example taking 45.5 and outputting “45.5 cm” or “45.5%”. It can however also be used to mathematically evaluate the value. The result of this is what is displayed on the UI whenever the value should be shown.

ValueScale	`number => number`
	or
	`[number, number]`
	or
	`number`
		Either a function, an array of two numbers, or a single number, that identify how the value scales in the condition. If it is a function, it will receive the numerical value and output the scale. When it is two numbers they will be treated as lower and upper limits, and the scale will be calculated as the numerical value linearly scaled between the two. When it is a single number, this is considered the upper limit and the lower limit is implicitly 0. The resulting scale is used to determine how bad the condition is. 1 is considered 100% severe, while 0 is considered 0%. This is mainly used in color scaling.

As the primary purpose of these are UI-related, the implementation of condition types is moved directly to the frontend. The condition types are however designed so that they are loaded from their own file. This means that the file can be generated by the backend, allowing dynamic creation of condition types based on backend code. Since frontend files are served by the backend whenever a client connects, they will be the most recent up-to-date generated version, achieving the same result as having an API endpoint to retrieve them.

The definitions of what types exist and what their properties are is entirely up to the system in which LiRA Map will be deployed to define. For the purposes of the prototype, this file is

manually scripted rather than generated, and contains the condition types from LiRA Map 1.0.

3.3.2.3 Map Component

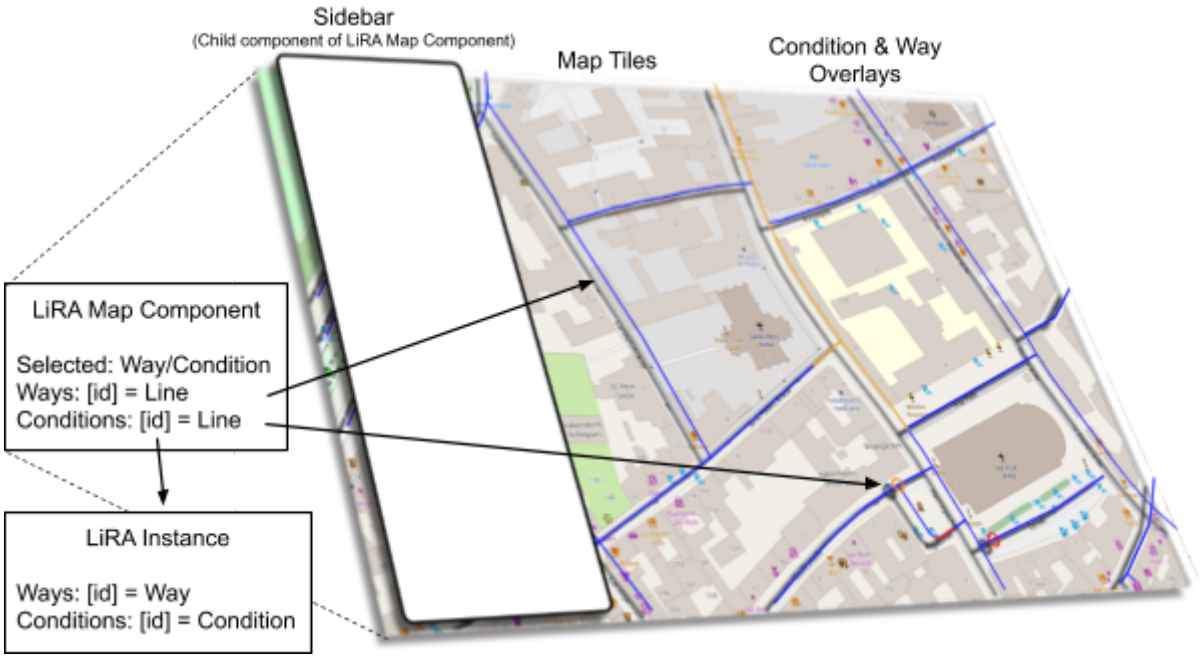


Figure 22: Diagram of map view architecture. Boxes represent parts of the underlying code objects.

Figure 22 illustrates how the map view is designed to work. A map component, here named LiRA Map Component, handles rendering the map through Leaflet. It contains a dictionary for ways and conditions that map each by their ID to their respective line on the map. These lines are drawn as an overlay layer on the Leaflet map. The sidebar is a child component that is drawn on top of the map. Its rendering is dependent on the *Selected* property in the LiRA Map component underneath.

This map component is the main React component in the UI of the map view. Upon initialization, it adds event listeners to the internal Leaflet map for whenever the map has been moved, which are used to request data from the LiRA instance. The map component adds a listener to the LiRA instance for whenever new ways or conditions are created or updated, and uses their draw functions to generate the visual overlay it can add to the map. These overlay objects are stored in the map component itself, making it manage its own state independently of the LiRA instance.

Whenever a map overlay layer is created, it adds listeners for when they are hovered and clicked, and these are used to control the user selection functionality of the map. Whenever a layer is hovered, it is highlighted, and whenever it is clicked, the way or condition from which it was generated is selected in the map component. This causes an update to the details panel in the sidebar, showing the details of the selected object.

3.3.2.4 LiRA Instance

The LiRA instance is the core of the frontend. It is a single instance of the LiRA class, which is designed to be the center of all communication and data to and from the server. It contains all discovered Way and Condition objects, as well as the global list of nodes, which are stored simply as a dictionary mapping the node ID to the latitude and longitude coordinates of said node. It provides getters for retrieving these objects, and also implements the functionality to create API requests depending on need, such as for example whenever the map is moved.

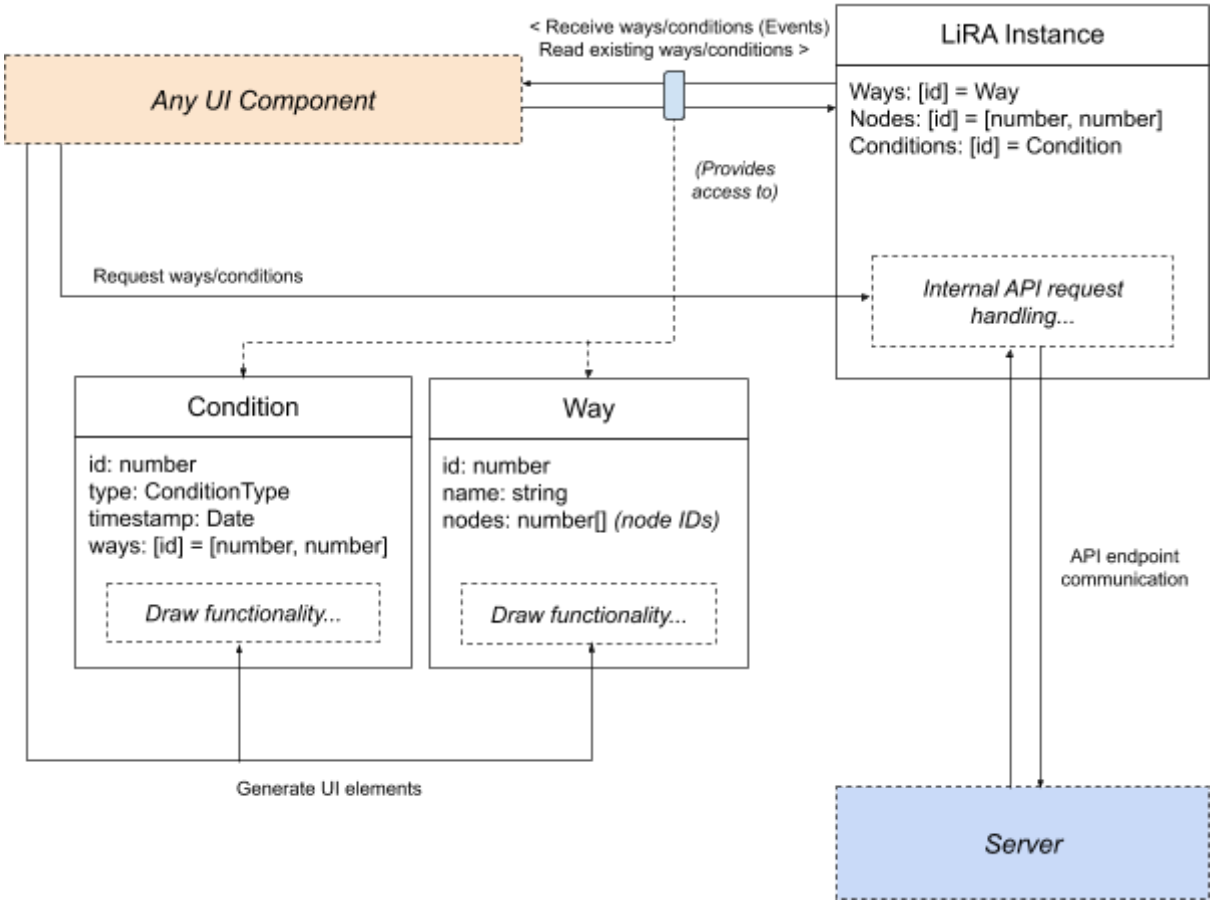


Figure 23: Diagram illustrating communication between the UI components of LiRA Map and the LiRA instance.

The LiRA instance is event-based, and fires events whenever an API request succeeds, and new ways or conditions are loaded. Components such as the map can add listeners to these, loading in the objects as they are received by the server. Figure 23 shows how a component communicates with the LiRA instance and how they use Ways and Conditions. The component can access ways and conditions from the instance, upon which it can call the appropriate drawing functions to generate the UI elements needed for the component. Components may also request new data through the LiRA instance's API request functions, which mirror those of the endpoints available to LiRA Map. The requests are asynchronous, and thus events are used for whenever the server responds with new data.

As this is a single shared event, any component listening to it will update its UI, even if it wasn't the component responsible for creating the request. Additionally, an event is fired whenever the LiRA instance begins requesting on the server, and whenever all active requests are finished. This is used in the UI to display to the user that the LiRA Map application is currently awaiting data from the server.

The instance is the only one of its class, and is shared globally across the entire application frontend. If in the future more pages or functions are desired, this design allows them all to share the ways and conditions, optimizing the application by not retrieving duplicate information across each page. The LiRA instance can easily be given more request types for components to use, and the data retrieved will be equally usable by all other components.

This describes the design of the frontend of the LiRA Map application.

4. Implementation

This chapter will talk briefly about the technical implementation of the design developed in Chapter 3, including technical details not mentioned therein, of select parts of the prototype code. This is in order to give a better understanding in some of the algorithms developed, and provide a better overview over how the application actually retrieves data, communicates, and how the application handles rendering. This will be done by providing small code snippets and explaining what they do and how it correlates to the design. At the end of the chapter in Section 4.3, there will be some screenshots from the final prototype of the application.

4.1 Back-End

4.1.1 The Controller

ASP.NET uses *controllers* to handle API endpoints. The backend is designed such that there is a controller for each frontend view, with endpoints relating to the usage of that view. The *Map Controller* corresponds to the map view, and is the only controller implemented in this prototype.

```
private static object CreateResponse(IEnumerable<Way> Ways, IEnumerable<Node> Nodes,
ICollection<Condition> Conditions)
{
    return new
    {
        Ways,
        Nodes = Nodes.ToDictionary(n => "" + n.Id, n => n.Coords),
        Conditions
    };
}
```

Snippet A (Controllers/MapController.cs, Line 25-33)

While the controller only has one endpoint in this prototype, it is designed such that the response structure is uniform between all endpoints it may end up having. The returned object is automatically parsed to JSON by ASP.NET. Ways and Conditions remain the same, however Nodes is converted to a dictionary mapping the ID to the two coordinates, as this greatly shortens the resulting JSON's length from not having to map the entire Node object.

This JSON is what is communicated to the client.

4.1.2 Way Provider

The IWayProvider interface is the interface for the Way Provider modules. In the current it only contains a single function, as opposed to the design developed in Section 3.3.1.2. This is because the list view is not implemented in this prototype version, and thus this remains the only necessary function at this point.

```
public Task<Tuple<IEnumerable<Way>, IEnumerable<Node>>> GetWaysInBounds(double left, double bottom,
double right, double top, IEnumerable<double[]> known);
```

Snippet B (Services/IWayProvider.cs, Line 11): Signature of only implemented function in the IWayProvider interface.

4.1.2.1 Overpass Way Provider

The Overpass Way Provider is the implementation of IWayProvider used in the current prototype version of LiRA Map. It implements the function above using the method decided upon in Section 3.2.1, which was to use the Difference operator in Overpass to subtract all previously searched boxes from the results of the requested one.

In order to achieve this result, we must construct a query in the shape of the following example:

```
(
  (
    way[highway][name]({{bbox}});
        -
    (
      way[highway][name]({{bbox1}});
      way[highway][name]({{bbox2}});
      way[highway][name]({{bbox3}});
    );
  );
  node(w);
);
out qt;
```

The lines `way[highway][name]({{bbox}});` asks for all ways within the coordinates specified bounding box that has a name tag and a highway tag set. This isn't exactly perfect (see Section 5.3), but is designed so that it can easily be tweaked by making this a string that can be edited at the very top of the class. The {{bbox}} entries are replaced by the four numbers specifying the bounds of the geographical box desired.

What the query in the above example does, is first find all ways within the main requested bounding box. Then, only if there is 1 or more already-searched intersecting box, it will add the difference operator ("-") and subtract the combined set of these already-searched boxes (bbox1, bbox2, bbox3, …). Finally, all nodes existing within the resulting set of ways is added using the special `node(w);` operation.

This query is sent to the Overpass API through the interpreter endpoint in the shape of a string without indentation and newlines, which may for example look like this:

```
https://lz4.overpass-api.de/api/interpreter?data=((way[highway][name](55.67519480709519,12.560580968856813,55.68003435983046,12.581180334091188);-(way[highway][name](55.67842225033158,12.565875649452211,55.680841901899626,12.576175332069397);way[highway][name](55.677218417918894,12.568295001983644,55.679638143954286,12.578594684600832);way[highway][name](55.67662858700813,12.565205097198488,55.67904834952922,12.575504779815676);););node(w);); out qt;
```

Which when written out using indentation and spaces for readability looks like this *(Numbers concatenated to four digits for readability)*:

```
https://lz4.overpass-api.de/api/interpreter?data=(
  (
    way[highway][name](55.6751,12.5605,55.6800,12.5811);
    -
    (
      way[highway][name](55.6784,12.5658,55.6808,12.5761);
      way[highway][name](55.6772,12.5682,55.6796,12.5785);
      way[highway][name](55.6766,12.5652,55.6790,12.57550);
    );
  );
  node(w);
);
out qt;
```

The result of a query of this structure is an XML response that looks similar to the example response below:

```
<node id="7625754" lat="55.6973225" lon="12.5843584"/>
<node id="7625998" lat="55.6974112" lon="12.5893746"/>
<node id="7626002" lat="55.6970361" lon="12.5878655"/>
[more nodes ...]

<way id="357113628">
    <nd ref="2534881862"/>
    <nd ref="1866790181"/>
    <nd ref="7029092292"/>
    [more nodes ...]
    <tag k="name" v="Classensgade"/>
    [other tags ...]
</way>
<way id="83719373">
   [more ways ...]
```

The XML contains the full list of nodes, each with their ID and coordinates, following by the full set of ways, each with the ordered references to nodes, along with their tags. The backend of the LiRA Map application receives this result, and uses an XML parser to read it, generating the Way and Node objects from its results.

4.1.3 Condition Provider

The IConditionProvider interface is the interface implemented for all Condition Provider modules. Similarly to IWayProvider and for the same reasons, the IConditionProvider interface is also lacking functions that were previously introduced in the design chapter in Section 3.3.1.2.

```
public Task<ICollection<Condition>> GetConditions(IEnumerable<Way> ways);
```

Snippet C (Services/IConditionProvider.cs, Line 12): Signature of the current IConditionProvider interface.

4.1.3.1 Database Condition Provider

The Database Condition Provider is the database implementation of IConditionProvider. It uses a local database to store and retrieve the conditions. It implements the two functions using Entity Framework C# to construct an SQL query that retrieves conditions corresponding to the purpose of the function, as defined in Section 3.3.1.2.

In the table below, we can see the Entity Framework queries, and the SQL equivalent *(Note: SQL generated from Entity Framework)*.

Retrieve conditions by list of ways.
Entity Framework C#

`db.Conditions`
`.Where(c => c.Coverage.Any(cov => wayids.Contains(cov.Way)))`
`.Include(c => c.Coverage)`
`.ToListAsync();`

SQL Equivalent

`SELECT c."Id", c."ConditionType", c."Timestamp", c."Value", c0."Id", c0."ConditionId", c0."EndMeters", c0."StartMeters", c0."Way"`
 	
`FROM "Conditions" AS c`
`LEFT JOIN "ConditionCoverages" AS c0 ON c."Id" = c0."ConditionId"`
`WHERE EXISTS (`
`SELECT 1`
`FROM "ConditionCoverages" AS c1`
`WHERE (c."Id" = c1."ConditionId") AND c1."Way" IN ([IDs…])`
`)`
`ORDER BY c."Id", c0."Id"`

The above code demonstrates a method that works for the purposes of the prototype of LiRA Map. This query is likely able to be optimized in future iterations, selecting from

ConditionCoverages instead of Conditions which are indexed by the way IDs. See Section 5.3.

4.2 Front-End

The implementation of the frontend is closely related to the design developed in Section 3.3.2. A series of React components is created for the purposes of the LiRA Map application, including the details panel and the Leaflet map container component, which is also responsible for communicating with the LiRA instance on the map view.

The rendering of ways and conditions are implemented into the classes of the ways and conditions themselves, rather than the map component. As they all share the same named `.draw()` function, the map can handle any implementation of a geographical overlay relating to an object, regardless of what the object is and how it draws itself.

4.2.1 Rendering of Ways and Conditions on the Map

The `.draw()` function on the ways and conditions generate the Leaflet overlay objects representing themselves. By default, both ways and conditions generate *Polylines* - lines defined by ordered geographical points.

Both the Way class and the Condition class use the function `.calculatePath()` which returns the ordered set of geographical points that should be used in the creation of the polyline. For ways, it is simply the coordinates belonging to the nodes in the way's ordered list of node IDs:

```
calculatePath(): [number, number][] | undefined {
        var path = [];
        for (var nid of this.nodes) {
                path.push(LiRA.getNode(nid));
        }

        if (path.length > 0) return path;
}
```

Snippet D (ClientApp/src/LiRA/Classes.ts, Line 52-59): The path calculation for a way.

The `.getNode(id)` function in the LiRA instance returns the node coordinates of the node by the specified ID. This results in ways being defined by a polyline made up of the node coordinates that make up the way.

The rendering of conditions is not nearly as trivial. We know a condition applies to a *set* of ways, each covered by different relative start- and end positions. As the condition does not contain any node data itself, it must use the nodes referred to by each of the ways it covers. For each of them, it may also potentially have to find a point *between* two nodes, as the start- and end positions may not always align with the nodes within the way.

```
calculatePath(): [number, number][][] | [number, number][] | undefined {
  var list: [number, number][][] = [];
  for (var wid in this.ways) {
    var way: Way = LiRA.getWay(wid);
    if (way != null) {
      var cover = this.ways[wid];
      var wnodes: [number,number][] = [];
      var dist = 0;
      var n = LiRA.getNode(way.nodes[0]);
      var l_prev = L.latLng(n);
      var started = false;
      var ended = false;

      if (cover[0] <= 0) {
        wnodes.push(n);
        started = true;
      }

      for (var i = 1; i < way.nodes.length; i++) {
        n = LiRA.getNode(way.nodes[i]);
        var l_next = L.latLng(n);
        var d = l_prev.distanceTo(l_next);
        var total_d = dist + d

        if (!started && total_d >= cover[0]) {
          var scale = (total_d - cover[0]) / d
          wnodes.push([
            l_next.lat + (l_prev.lat - l_next.lat) * scale,
            l_next.lng + (l_prev.lng - l_next.lng) * scale
          ]);
          started = true;
        }

        if (total_d >= cover[1]) {
          var scale = (total_d - cover[1]) / d
          wnodes.push([
            l_next.lat + (l_prev.lat - l_next.lat) * scale,
            l_next.lng + (l_prev.lng - l_next.lng) * scale
          ]);
          ended = true;
          break;
        }

        dist = total_d;
        l_prev = l_next;
        if (started) wnodes.push(n);
      }

      if (started && !ended) {
        wnodes.push(n);
      }

      if (wnodes.length > 0) list.push(wnodes);
    }
  }

  if (list.length > 0) return list;
}
```

Snippet E (ClientApp/src/LiRA/Classes.ts, Line 138-90): The path calculation for a condition.

The above algorithm in Snippet E is the central key to rendering conditions across ways on the map. It creates a *multiline* - that is, an *array* of polylines which will be treated as a single overlay object on the map. Each polyline in this multiline represents each individual *coverage* this condition owns, i.e. each individual way it covers. LiRA Map 1.0 used a similar method to split roads in the middle of sections in order to render its colors.

We will now go through the algorithm and explain how it works.

The algorithm iterates over the list of ways the condition covers. For each of them, it creates the empty array that will contain the coordinates for the polyline of the condition on this particular way (named `wnodes`). It will allocate a variable *n* to the node currently looked at, starting with the first node in the way. It will also allocate a variable *l_prev* representing the latitude/longitude coordinates of the node *previously* looked at. If the starting meters for how much the condition covers this way is 0 or lower, it will immediately add *n* as the first node.

The algorithm will then iterate over the rest of the nodes in the way. As long as it hasn't passed the starting point yet (indicated by the `started` variable), it will skip the iterated nodes. During each step, it will count the total distance traversed across the way. When this distance suddenly exceeds the starting meters (as long as we haven't started already), the scale of how much we went past it is used to calculate the coordinates between the current node and the previous one that match the starting position. These coordinates are pushed into `wnodes`, and the algorithm marks having started tracing the path of the condition along this way.

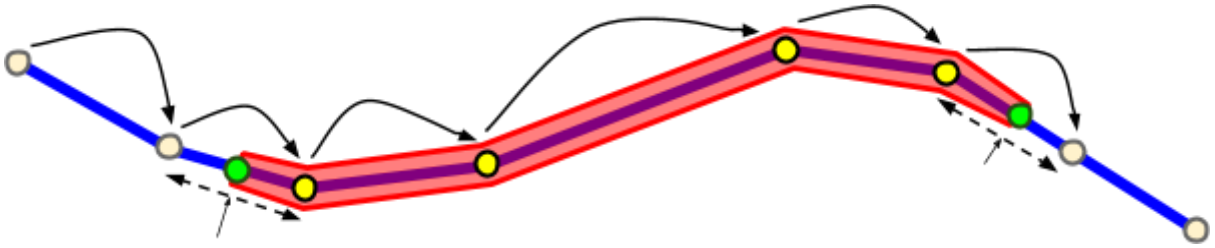


Figure 24: Traversing nodes in a way to find the path of the condition. The green nodes are the linearly scaled points between the two nodes when passing the start and end distance. The yellow nodes are all nodes added between. The grey ones are not included in the condition. Notice the last node is never visited.

While traversing nodes after the starting point was passed, it will check whether it has passed the *ending* point in the same way. While it hasn't, every node traversed is pushed into the `wnodes` array. When and if the total distance suddenly exceeds the ending meters, the same scale calculation is used to determine the ending coordinates between the current node and the previous, and these coordinates are pushed to the array and the algorithm stops the iteration of the current way's nodes. If the ending meters is not exceeded, the final node in the way is also the final in the condition's coverage of the way.

The algorithm does this individually for each way covered by the condition. The final result is the array of polylines for each of the ways. Leaflet can create Multilines using this type of array, allowing conditions to be drawn in multiple directions and multiple segments, as discussed in Section 2.2.4.

The paths generated from the `.calculatePath()` functions in both ways and conditions is used in the `.draw()` function to create the line or multiline. The *Point* Condition class extends the Condition class and overwrites the `.draw()` function to instead create a *circle* at the point determined by `.calculatePath()`. `.calculatePath()` is also overwritten to perform a similar algorithm as the Condition class did previously, but only intending to find a single point along the associated way.

4.3 Application Screenshots

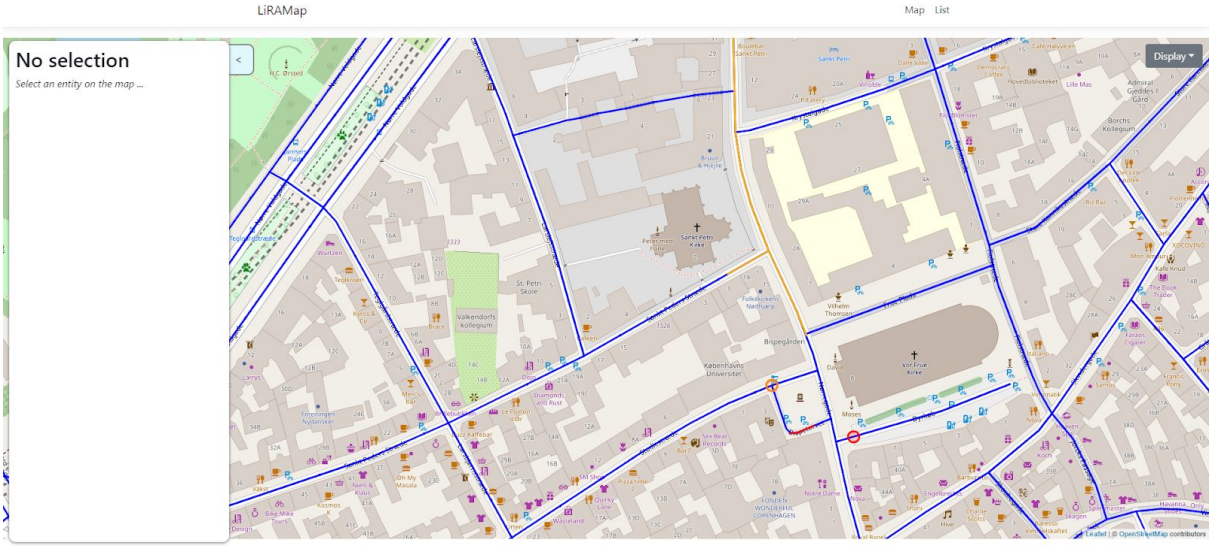


Figure 25: LiRA Map prototype application of default area. Ways are drawn in blue lines, conditions are drawn in scaled colors from orange to red depending on their severity. Four conditions can be seen.

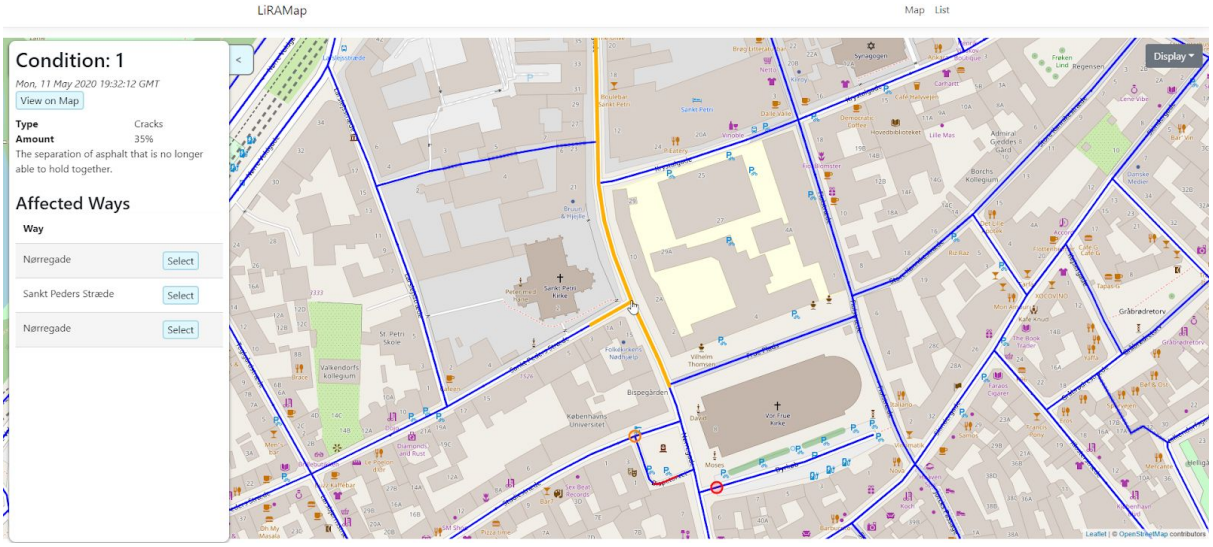


Figure 26: The user is hovering over the orange condition in the middle. It is also selected, revealing its details on the details panel to the left.

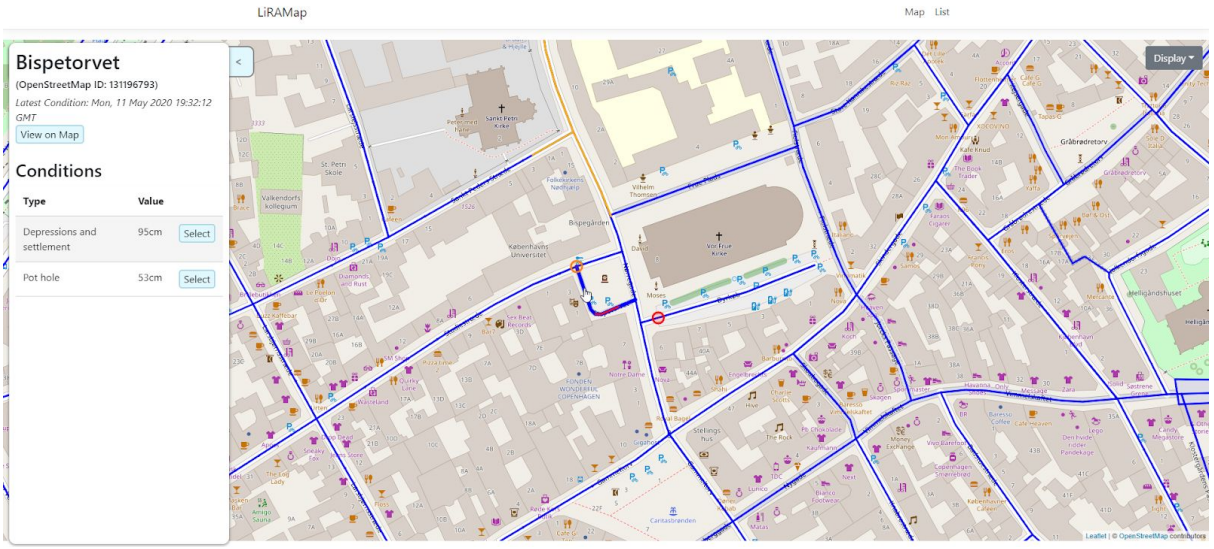


Figure 27: The hovered way is now selected. Its name and details can be seen on the details panel. The two conditions that exist on this way can also be seen listed. The select buttons will change the selection to these two.

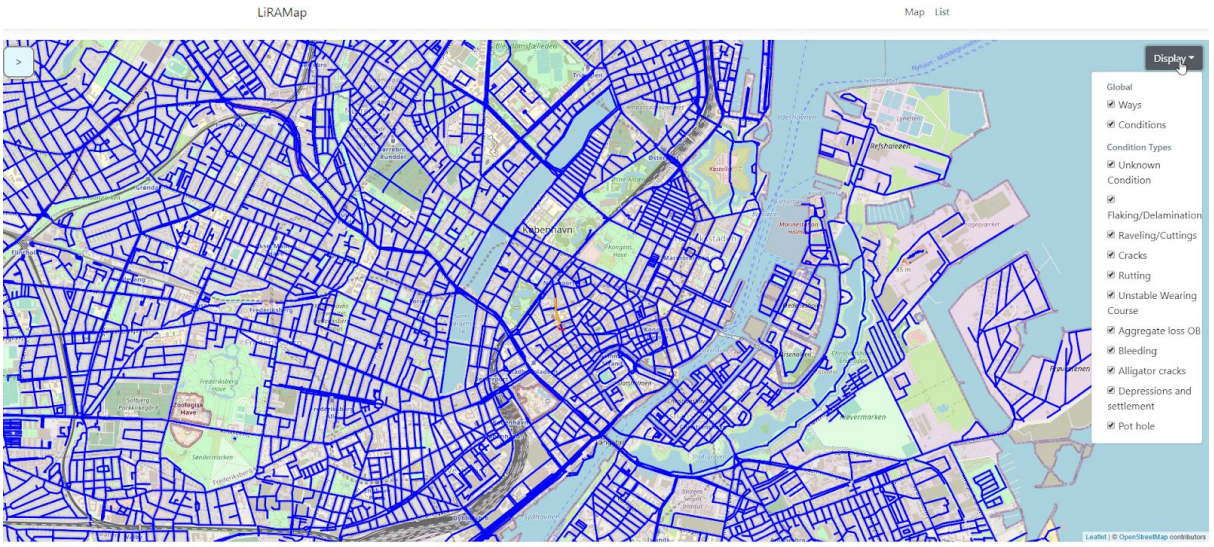


Figure 28: A larger zoomed out view, displaying a large portion of roads. The details panel has been collapsed. The display dropdown is visible, showing checkboxes that allow the hiding of ways and conditions, as well as selectively conditions by their type.

5. Evaluation

In the evaluation of the LiRA Map prototype application, we will go through first all capabilities the application has and how well they were implemented, with perspective towards usage by an end user. Then we will go through missing capabilities, being those that were planned to be implemented but weren't, or those that were only half-finished. Finally, we will discuss potential future developments and optimizations besides the missing capabilities. These are only developer observations, and LiRA Map has not yet been shown to end-users or received in-depth feedback from LiRA project members and experts.

5.1 Capabilities

The capabilities of the current LiRA Map prototype provide basic navigation and condition visualization purposes. The map is able to be intuitively navigated by the users mouse, with both dragging and zooming capabilities, and the user is able to click to select both ways and conditions, and view details on both. Details of ways will display the name of the way, the OSM ID, along with all conditions that refer to it. It will also display the timestamp of the most recent condition. Reviewing condition details will display the condition type and description, along with the value translated to readable format. An additional feature is also the *Select* buttons, that will let the user cross-navigate linked conditions and ways. All of these selections are highlighted on hover, including the Select buttons highlighting the associated overlay on the map.

The application is not restricted to any area, and can load ways anywhere in the world as long as they exist in the Overpass/OSM database. The design of the system enables LiRA to store nothing more than conditions, which makes managing the data a lot easier and without limitations. The modular design of the backend also allows the easy replacement of sources, such as for example changing to storing ways locally, or externalizing conditions.

The application also has a high flexibility for conditions and condition types. Condition types can easily be added, either manually, or through generating the code file. If it should be desired, the class-based hierarchical design of ways and conditions allow the easy creation of subclasses that may display differently. This could for example be used to make *informational* object types that don't use condition types, or possibly even whole new objects that exist entirely separate from ways.

In the evaluation of LiRA Map 1.0, end-user feedback expressed the lack of ability to inspect roads at a more detailed level. In Lira Map 1.0, the user was only able to select sections. A case could potentially exist where a section might be in a severe state only up until the halfway point, where the other half would be in perfect state. In LiRA Map 1.0, it was impossible to inspect only the subsection that was severe. In this new iteration of LiRA Map, conditions themselves can be selected, meaning a user could inspect the areas in severe state directly. This also fulfills the desire to be able to view conditions independently that LiRA Map 1.0 lacked.

5.2 Missing Capabilities

Several capabilities desired early on in the development were not implemented in time for the current prototype. Some of these features were half-implemented, others not implemented at all. All of them were intended to be implemented, and would be the next steps in the future development of LiRA Map, if this prototype is chosen to be continued.

The most notable missing feature is the *list view*. A lot of the system was designed so that it would be ready for managing this alongside the map. The list view however never fully made it to the prototype due to time constraints and prioritization of the map view. The list view

was supposed to be a different component in the SPA, and would display a table of all ways or all conditions, depending on the user's choice. It would also support filtering and sorting features, which could allow the user to for example sort the list by numerical value, or filter it to only show potholes. The design of the modules were developed for this, however as the list view never made it to the actual application, they have yet to be implemented too.

Another notable missing feature is the search bar. It was a feature desired early on, but similarly to the list view, wasn't implemented due to the limited scope of the prototype project. It would have been positioned in the top middle of the screen, and would display a dropdown of ways as the user typed into the input field. It would only be able to list ways already known to the client, however the dropdown would have been developed such that it always had a fixed *"Search for …"* option at the very top of the dropdown, which would query the backend for this way by name. When retrieved this way, the user could click it to navigate the map to its position. It would also be necessary to add a *FindByName* function to the Way Provider module interface.

A half-implemented feature is the condition type filtering on the map view. In Figure 28, this can be seen on the right side of the screen. This dropdown functions and when the checkboxes are clicked, it will correctly show or hide the conditions or ways associated to the checkbox. It also has the feature of preventing any API requests of all rendering is disabled. This feature was however a quick prototype feature, and the checkbox states do not properly retain when the user moves the map or the dropdown is closed. For example, if the user removes the tick on *Ways*, all ways will be hidden, however if the user closes the dropdown, moves the map, then opens it again, the *Ways* checkbox will be ticked despite the ways still being hidden. Additionally, hiding ways and showing them again will put them above conditions, meaning conditions will be hard to see below the blue lines.

For the purposes of the prototype, the database configuration, including the connection string, is hard-coded and requires a database set up on *localhost* with the name "LiRAMap" and a user named "liramap" with appropriate permissions and the password "updateme". This is not a good way to handle configuration, and should be updated to allow user-configuration through some means such as a configuration file or installation process.

If LiRA Map is chosen to continue development, these would be the first steps in continuing to work towards a full, viable LiRA Map version.

5.3 Future Work

On top of the things discussed in Section 5.2, there are many more ways the LiRA Map application can be improved and built upon. Many of the things listed here are things that were thought out very early or even before the development process, but were not expected to be added to the prototype version.

- Maintenance Planning System

  While it was never intended to be added to this prototype, the purpose of LiRA is to facilitate the planning and scheduling of repairs and road maintenance. It will become an important feature for LiRA Map to be able to perform this task. This is a larger feature, as it would require a full user-login system to keep track of who schedules what repairs, as well as assigning dates and workers for the repairs. There would also have to be a way for these workers to mark the repair as complete, automatically removing the condition. This system introduces an entire *write* side to the API, whereas it is currently read-only.

- Notifications

  A notification system involving notifying the user of very severe conditions that require prioritized attention. Implementing this system would require the frontend to regularly poll the backend for any updates to condition severities, and retrieving these conditions even without seeing them on the map. Alternatively, the frontend could perform this poll only once on web page load.

- Historical Data

  An early desire in the LiRA Map project was the ability to view historical data. It would be a potential future development to implement such a functionality into the LiRA Map application. This would mean having conditions not be deleted upon repair, but instead marked as repair, retaining the data for historical inspection. It would also mean conditions should support iterations, i.e. different numerical values and coverages by different timestamps. The frontend views would also have to have an intuitive way to render these. Possible solutions could be to separate repaired conditions with a checkbox in the map filtering, and making the detail panel of conditions display the historical values, possibly as a graph instead of a single number.

- Extended Node Capabilities

  It may not ever become a desire feature, but it would be possible to overhaul the node class such that it can contain data similarly to ways, as well as assigning nodes their own class in the frontend too rather than just coordinates. This could allow the *selection* of nodes, which could potentially display all connected roads, as well as all conditions that go across.

- Extended Node and Way Data

  It would also be possible to extend the way class to contain additional data such as speed limits, pavement type, lane count, and more. Many of these could be sourced from OSM just like the name already is. It would alternatively be possible to store these data points, or any data points, in the local database instead. In that case, the concept of grouping ways into whole roads could be brought back, optimizing the database by not having to store duplicate information for each way under the same name.

- Security

  The current prototype of LiRA Map is almost built with no security in mind. As it is currently read-only and has very limited endpoints, there are not many security issues to encounter. However the few endpoints that do exist do have their own lacking exception handling. For example, if the Map Controller's bounds endpoint receives parameters that cannot be parsed to doubles, it will throw an exception. For the purposes of the prototype, these concerns were ignored, however for future development, a full security check-up would be desired.

On top of all the above mentioned larger features, there are also numerous smaller usability improvements and optimizations. Conditions could for example be drawn not always as orange to red color scaling, but different hues for each condition type, possibly even letting the user change these, where the severity scaling is shown in the saturation instead. This would allow an easier overview of exactly what types of conditions exist where when viewing a large area. The database model for Conditions and ConditionCoverages could also be improved along with the queries, such as for example reworking the Database Condition Provider query to one that searches on more efficiently on ConditionCoverages, rather than iterates all Conditions. Another improvement would be to tweak the Overpass filters so that it more accurately gets roads. The current query has potential to miss roads that do not have an explicit name assigned, and also incorrectly may grab footpaths if they have a name.

The potential future development of LiRA Map could look into all of these things, all while further optimizing the application in all areas.

6. Conclusion

Talk about the overall usability of the LiRA Map application. Talk about how it is a viable good starting point for a more robust and complete application. Refer back to introduction, and explain how it fulfills those wants.

The Live Road Assessment project, called *LiRA* for short, aims to develop a system used to assess the condition of Danish roads in order to help the maintenance process of said roads. Currently, road conditions are assessed through manual inspection at longer time intervals. This comes with an operation cost, and also means lower data accuracy and resolution.

LiRA aims at solving these problems by making use of sensors built into modern passenger cars. These sensors collect a lot of different data while traversing the roads, which can be collected and analyzed in order to assess the road's condition. By taking advantage of the many modern passenger cars already out in the country, manual assessment costs can effectively be removed and the data becomes much more accurate through the significantly higher update frequency.

The LiRA project consists of multiple separate components which all work together to solve the problem. This report focuses on a single one of these components: The LiRA Map Application. This application is a geographic information system (GIS) application which serves the purpose of displaying the condition data collected by the cars and translated by a machine learning system. It will present the data overlaid on an easily navigable map, providing a quick overview of the state of roads in any area navigated to by an end-user, namely employees of road maintenance management organizations. The app aims to facilitate the maintenance planning process by providing end-users with an easily understood interface and allowing them to quickly locate what roads need repair next.

GIS applications for this purpose already exist, such as Rosy and Vejman by Sweco and the Danish Road Directorate respectively, as well as a previous version of the LiRA Map application made by DTU M.Sc. student Markus Berthold. The LiRA Map app is built to be a non-commercial, open source alternative to these existing products, and to remove the need for third-party software for LiRA. The version built in this project is rebuilt from the ground up to be a *cloud-based web application* as opposed to the old version being a standalone application.

LiRA Map, along with the rest of the LiRA project, is made with Denmark and the Danish Road Directorate in mind. There is however no technical reason for this restriction, and as such the LiRA Map application will not be restricted.

The LiRA Map application is designed to work together with the rest of the LiRA system to fulfill its purpose: Assessing the condition of Danish roads and helping the maintenance process by letting users view the state of roads from sensors attached to regular passenger cars. These sensors collect the data, which goes through the machine learning and data analysis component of the LiRA system, resulting in the creation of conditions. The LiRA Map application is a geographical information system that can display these conditions on an easily navigable map. This GIS application can help employees in the Danish Road Directorate get an overview of the state of roads in a user-specified area, allowing them to quickly locate and plan what roads to repair next.

Through developing this project, we learned how these kinds of road situations could be modelled, and how these models could be used in software to help facilitate the maintenance and repair process. We developed a prototype application for LiRA Map that demonstrates one such approach to the problem. The resulting application uses only open source and freely available libraries, and has the flexibility to be easily adapted to any source of path data, and any source of overlay data. It also provides a flexible framework for rendering geographical data on a map in general.

The LiRA Map prototype was developed in close connection with LiRA project members. The design of the prototype provides great groundwork and a potential for being a stepping stone towards a finished product that could be fully used in the LiRA project, together with the rest of the LiRA system.

7 References

Structure:

School project
[1] <Author>: <Title>. <Type?> <School/Institute>, <Dato>.
Link: <Link>

Journal
[1] <Author>: <Title>. <Journal Name> <Volume/number>, <Publisher>, <Dato>.
Link: <Link>

Workshop
[1] <Author>: <Title>. <Conference/workshop name> <Location, Country>, <Dato>.
Link: <Link>

Web
[1] <Author>: <Title>. Link: <Link>, <Dato last checked>.

[1] LiRA Project: About. Link: http://lira-project.dk/about/ (02/05/2020)

[2] Wikipedia: Geographic Information System.
Link: https://en.wikipedia.org/wiki/Geographic_information_system (23/05/2020)

[3] Sweco: ROSY® RAMS - ROAD ASSET MANAGEMENT SYSTEM
Link:
https://www.sweco.dk/vi-tilbyder/infrastruktur/drift-og-vedligeholdelse-af-belagninger-og-assets2/?service=RoSy%C2%AE%20RAMS%20-%20Road%20Asset%20Management%20System (02/05/2020)

[4] Vejman: About Link: http://www.vejman.dk/DA/om/Sider/Default.aspx (02/05/2020)

[5] Markus Berthold: Live Road Assessment based on modern cars: A prototypical geographic information system for road maintenance planning. Master Thesis, DTU, 26/01/2020. Link: https://findit.dtu.dk/en/catalog/2496058228

[6] OSM Wiki: Relation - Street. Link: https://wiki.openstreetmap.org/wiki/Relation:street (23/02/2020)

[7] OSM Wiki: Databases and data access APIs - Sources of OSM Data

Link:
https://wiki.openstreetmap.org/wiki/Databases_and_data_access_APIs#Sources_of_OSM_Data (23/05/2020)

[8] OSM Wiki: Overpass API. Link: https://wiki.openstreetmap.org/wiki/Overpass_API (23/05/2020)

[9] OSM Wiki: Overpass API - Overpass API by Example
Link: https://wiki.openstreetmap.org/wiki/Overpass_API/Overpass_API_by_Example (23/05/2020)

[10] Leaflet Library. Link: https://leafletjs.com/ (23/05/2020)