Visualisierung und Analyse von Mobilfunkmessreihen

Bachelorarbeit

von

Franz Kary

aus

Dschilikul / Tadschikistan

vorgelegt am

Lehrstuhl für Rechnernetze und Kommunikationssysteme

Prof. Dr. Martin Mauve

Heinrich-Heine-Universität Düsseldorf

Oktober 2012

Betreuer:

Norbert Goebel M. Sc.

Abstract

Im Rahmen dieser Bachelorarbeit wurde eine Software zur Darstellung von Daten aus Mobilfunkmessreihen implementiert. Mit dieser können beliebige numerische Werte (wie z.B. Datenrate, Latenz oder Droprate) an den entsprechenden Geopositionen auf Karten abgebildet werden. Dabei stehen unterschiedliche graphische Darstellungen, wie Heatmaps oder Punktabbildungen mit verschiedenen Einstellungsmöglichkeiten zur Auswahl. Dies ermöglicht den visuellen Vergleich von Messreihen, bei dem Abweichungen oder Gemeinsamkeiten schnell ersichtlich werden. Die einzelnen Parameter der Datenabfragen und eine Vielzahl von Anzeigeneinstellungen sind dabei flexibel konfigurierbar und können jederzeit geändert werden. Um weitere, alternative Betrachtungsmöglichkeiten zu bieten, werden eine Tabellenansicht und die Darstellung der Daten auf einem zweidimensionalen Koordinatensystem angeboten.

Da zum Zeitpunkt dieser Arbeit noch nicht abzusehen ist, ob die bereits implementierten Softwarekomponenten in Zukunft allen neuen Anforderungen genügen, wurde ein besonderer Wert auf deren Erweiterbarkeit gelegt. Somit wird es mit geringem Aufwand möglich sein, Daten aus alternativen Datenquellen, als der zur Zeit genutzen MySQL Datenbank auszulesen, Algorithmen bereits existierender Darstellungsarten hinzuzufügen oder neue Darstellungsarten zu entwickeln.

Danksagung

Ich möchte diese Stelle nutzen, um mich bei meinem Betreuer Norbert Goebel für die vielen kompetenten Diskussionen, Anregungen, Kritiken und Ratschläge zu bedanken. Diese haben mir bei dieser Arbeit und darüber hinaus sehr geholfen . In zahlreichen Gesprächen stand er mir dafür im Verlauf dieser Arbeit zur Verfügung.

Weiterhin möchte ich mich bei David Grönebaum und Marc Kluge für ihr Interesse, ihre Unterstützung und Hilfsbereitschaft bedanken.

Einen besonderen Dank gilt meiner Familie, die mich durch die Höhen und Tiefen der letzten drei Monaten begleitet haben und mich in dieser Zeit dennoch immer engangiert unterstützt haben.

Danke!

Inhaltsverzeichnis

Abstract iii

Abbildungsverzeichnis ix

Tabellenverzeichnis xi

1 Einleitung 1
- 1.1 Motivation 1
- 1.2 Aufbau 2

2 Verwandte Arbeiten 3
- 2.1 Wahrnehmungzyklus nach Neisser 3
- 2.2 Darstellung von Heatmaps 4

3 Design 5
- 3.1 Daten 6
  - 3.1.1 Datenbank 6
  - 3.1.2 Datenbankschnittstelle 7
  - 3.1.3 Abfrageobjekte 8
  - 3.1.4 Abfrageergebnisobjekte 8
- 3.2 Views 9
  - 3.2.1 Layer 9

4 Implementierung 11
- 4.1 GUI 13
- 4.2 Daten 15
  - 4.2.1 Abfrageklassen 15

4.2.2 Datenbank 16
4.3 Backend 17
4.4 Views 17
4.4.1 Einstellungen und Daten 18
4.4.2 Layer 18
4.4.3 Mapview 18
4.4.4 Tableview 24
4.4.5 2DPlotview 24

5 Auswertung **27**

6 Zusammenfassung und Ausblick **31**
6.1 Zusammenfassung 31
6.2 Ausblick 33

Literaturverzeichnis **35**

Abbildungsverzeichnis

2.1 Wahrnehmungszyklus nach Neisser. Quelle: [Buz00] 3
2.2 Generierung der Heatmap. Quelle: [cor]. 4

3.1 Das verallgemeinerte ER-Diagramm der Datenbankstruktur 7

4.1 Überblick GUI 14
4.2 Vereinfachtes Klassenmodell der Datenabfrage 15
4.3 Mapview mit markiertem Gebiet und geöffnetem Kontextmenü 20
4.4 Berechnung des Wertefelds eines Datenpunkts für den Heatmaplayer . . 22
4.5 Berechnung bei Überschneidung von Wertefelder für den Heatmaplayer 22
4.6 Beispiel einer Heatmap 23
4.7 Beispiel einer 2D Plotview 25

5.1 Bandbreite und Droprate im Up- und Downlink 29

Tabellenverzeichnis

5.1 Messreiheninformationen 27

Kapitel 1

Einleitung

1.1 Motivation

Damit Autofahrer sicherer, schneller und komfortabler ihr Ziel erreichen, forscht die Automobilindustrie an Kommunikationsmöglichkeiten für Fahrzeuge (*Car2x*). Dies ermöglicht es den Verkehrsteilnehmern Entscheidungen zu treffen, die nicht nur auf den eigenen Informationen beruhen, sondern auch die, der umliegenden Umgebung. Dadurch können beispielsweise Unfälle, die durch das Auffahren an Stauenden vor unübersichtlichen Kurven entstehen, vermieden werden. Bei den zu untersuchenden Verfahren, konkurriert vor allem das speziell für Fahrzeug-zu-Fahrzeug Kommunikation (*Car2Car*) entwickelte WLAN 802.11p mit dem traditionellen Mobilfunknetz.

Um möglichst genau abschätzen zu können, wie sich Anwendungen im Fahrzeugbereich in den beiden unterschiedlichen Netzwerken verhalten, sind Feldtests oder Simulationen notwendig. Aufgrund der Tatsache, dass Feldtests erfahrungsgemäß aufwendig und kostspielig sind, werden in der Regel zunächst Simulationen verwendet um erste Aussagen über das zu untersuchende Verhalten zu treffen.

Die bereits vorhanden Softwaresimulatoren für die Car2Car-Kommunikation über das Mobilfunktnetz benötigen exakte Angaben des Umgebungsmodells und der Mobilfunkparameter (wie z.B.: Sendeleistung, Sektorisierung, zugewiesene Datenrate der Sendemasten).

Aufgrund des Betriebsgeheimnisses stellen die Mobilfunkbetreiber ihre Daten hierfür allerdings nicht zur Verfügung. Zusätzlich erweist sich die genaue Erfassung des gesamten Testumfelds für das Erstellen eines Umgebungsmodells als schwierig. Ohne diese Daten ist eine der Realität entsprechende Simulation nur schwer durchzuführen.

Damit dies dennoch möglich wird, entwickelt der Lehrstuhl für Rechnernetze einen Simulator, der auf realen Messreihen basiert. Hierfür werden auf definierten Strecken mehrere Messfahrten durchgeführt um anschließend aus den resultierenden Daten eine Simulationsumgebung zu erstellen. Das Problem der Erfassung der benötigten Messdaten wurde in einer vorangegangenen Arbeit bereits untersucht [Goe10]. Für die Entwicklung der Simulation müssen zunächst die bei den Messungen entstandenden Daten analysiert werden. Eine reine Betrachtung der gesammelten Informationen in der üblichen Tabellenform erweist sich als sehr schwierig, unübersichtlich und zeitaufwendig. Demgegenüber bietet die Darstellung der Daten auf einer Karte einen umfangreichen Überblick, welcher auch die lokalen Besonderheiten des Simulationsgebiets umfasst. Ziel dieser Arbeit ist daher die Entwicklung einer Software, welche die Visualisierungen der Messreihen ermöglichen soll.

1.2 Aufbau

Zunächst befasst sich das folgenden Kapitel 2 mit verwandten Arbeiten. Anschließend werden in Kapitel 3 die nötigen Anforderungen dieser Arbeit vorgestellt um daraus ein entsprechendes Softwaredesign zu entwickeln. Die eigentliche Implementierung der Software wird in Kapitel 4 behandelt. Kapitel 5 beschäftigt sich mit den aus der Software bereits gewonnenen Erkenntnissen, bevor abschließend in Kapitel 6 diese Arbeit kurz zusammengefasst und ein kurzer Ausblick auf mögliche Anpassungen, Erweiterungen und Verbesserungen gegeben wird.

Kapitel 2

Verwandte Arbeiten

2.1 Wahrnehmungzyklus nach Neisser

Neisser hat 1976 in seinem Modell des Wahrnehmungszyklus dargestellt [Nei76], wie die Suche nach Informationen von Schemata geleitet wird. Die bei der Erkundung gewonnenen Informationen fließen anschließend wieder in die Schemata ein, was ggf. die Suche auf andere Informationen lenkt. Die Abbildung 2.1 veranschaulicht dies. Dieser Wahrnehmungsprozess spielt für die Verwendung von Visualisierungen eine bedeutende Rolle und ist daher auch für diese Arbeit relevant.

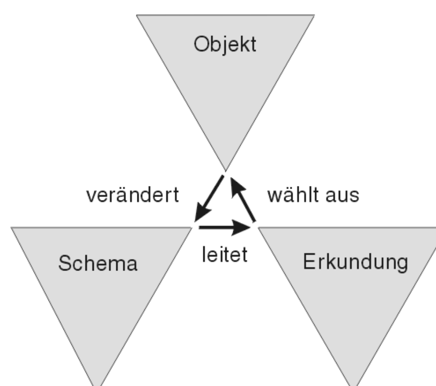
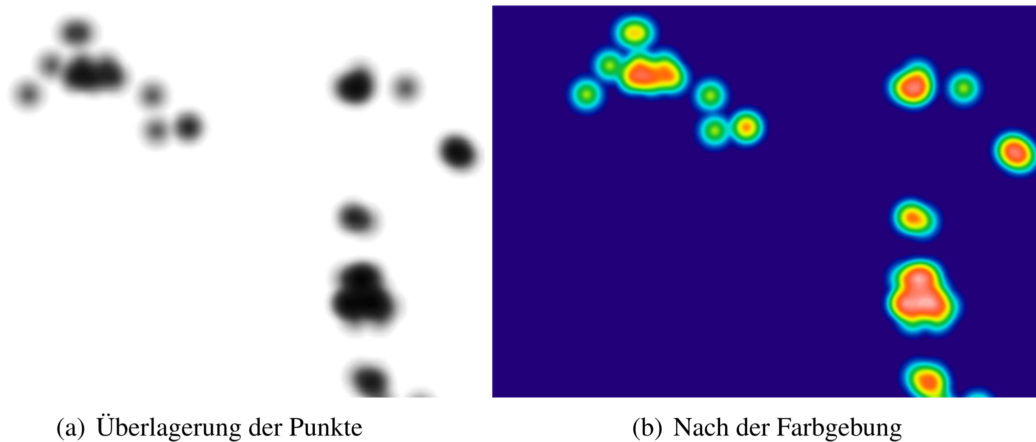


Abbildung 2.1: Wahrnehmungszyklus nach Neisser. Quelle: [Buz00]

.



(a) Überlagerung der Punkte (b) Nach der Farbgebung

Abbildung 2.2: Generierung der Heatmap. Quelle: [cor].

2.2 Darstellung von Heatmaps

Auf dem Blog der Firma *corunet* [cor] wird eine Methode zur Generierung von Heatmaps für Webseiten vorgestellt. Für diese wird eine Bilddatei verwendet, die einen schwarzen Punkt darstellt, dessen Transparenz nach aussen hin immer größer wird. Die Grafik wird anschliessend für jeden Datenpunkt an die entsprechende Stelle gezeichnet. Dadurch entsteht eine Ausgabe, wie sie in Abbildung 2.2(a) zu sehen ist. In diesem Gesamtbild wird nun jedem Punkt abhängig vom Transparentwert eine Farbe aus einem Farbverlauf zugeordnet (Abbildung 2.2(b)). Der verwendete Farbverlauf wird dabei aus einer weiteren Bilddatei ausgelesen.

Diese Art der Erstellung von Heatmaps hat dabei den Vorteil, dass sie weniger Rechenzeit benötigt als der in dieser Arbeit genutzte Algorithmus. Da allerdings die Algorithmen für die Heatmapberechnung konfigurierbar und einzelnen Werte der Heatmaps jederzeit erkennbar sein sollen, wird die hier vorgestellte Methode zur Generierung von Heatmaps nicht verwendet. Durch das Auslesen von Farbverläufen aus Grafiken bietet die vorgestellte Methode allerdings einen flexiblen Weg die Farbgebung der generierten Heatmaps anzupassen. Dies wird sich auch in dieser Arbeit wiederfinden.

Kapitel 3

Design

Im Rahmen dieser Bachelorarbeit war die Entwicklung eines Programms, welches die Möglichkeit bietet, Mobilfunkmessreihen visuell auszuwerten und zu analysieren. Dabei wurden grundsätzlich folgende Ziele verfolgt:

- Es soll unterschiedliche Möglichkeiten geben, Daten visuell darzustellen.
- Ein visueller Vergleich unterschiedlicher Daten soll möglich sein.
- Für die verarbeitenden Daten sollen Filtermöglichkeiten einstellbar sein.
- Um Erweiterungen und Anpassungen zu ermöglichen, sollen Datenquellen, wichtige Algorithmen und Ausgaben modular gestaltet werden.
- Die bereits gefilterten und aufbereiteten Daten und die daraus resultierenden Visualisierungen sollen exportierbar sein.
- Die Benutzeroberfläche soll leicht zugänglich und einfach zu bedienen sein.
- Der Speicherverbrauch und die benötigten Laufzeiten sollen möglichst gering gehalten werden.

Dieses Kapitel betrachtet vor allem zwei Hauptaspekte der zu programmierenden Software. Zunächst wird in Abschnitt 3.1 die bereits existierenden Messdaten, wie auf diese

zugegriffen wird und wie sie intern abstrahiert werden, beschrieben. In Abschnitt 3.2 werden anschließend die Grundlagen der visuellen Darstellung dieser Daten behandelt.

3.1 Daten

3.1.1 Datenbank

Die Daten der zu untersuchenden Messreihen liegen aktuell in einer MySQL Datenbank [mys] vor, welche die MyISAM Engine verwendet. Da wichtige Funktionen, die für eine andere Engine oder Datenbankwahl sprechen würden, nicht benötigt werden (referentielle Integrität, Geschwindigkeit bei verschachtelten Subquerys, ...), wurden hier keine Änderungen vorgenommen.

Bei dem Erfassen der Mobilfunkmessreihen wurden Daten von einer Vielzahl von Geräten und Softwarekomponenten gesammelt. Jede Komponente speichert hierfür ihre Daten in einer eigenen Tabelle. Dabei besitzen alle Datentupel jeweils einen Zeitstempel, aber keine sonstigen Referenzen untereinander. Desweiteren existiert eine Tabelle, in der sowohl der Start- und Endzeitpunkt, wie auch die Bezeichnung aller Messreihen enthalten sind.

Um die Daten auf Karten anzuzeigen, wird allerdings die zu einem Messdatentupel passende Geokoordinate benötigt. Diese Zuordnung ist, aufgrund der fehlenden Referenzen, nur über den in jeder Datentabelle vorhanden Zeitstempel realisierbar. Da es in Zukunft jedoch erforderlich sein kann, mehrere Messreihen zeitgleich zu erfassen, ist diese Art der Zuweisung nicht mehr eindeutig. Aus diesem Grund wurden die Tabellen um eine Spalte erweitert und darin zu jedem Datentupel die Id der jeweiligen Messreihe gespeichert. Daraus ergibt sich eine Datenbankstruktur, welche durch das Entity-Relationship-Diagramm in Abbildung 3.1 verallgemeinert dargestellt ist.

Da der Datentyp für die spätere Implementation von Bedeutung ist, mussten die Datentypen einiger Spalten geändert werden. Als Beispiel dient hier die Tabelle *gpsdo*, bei der die meisten Spalten vom Typ *varchar* auf *double* angepasst wurden.

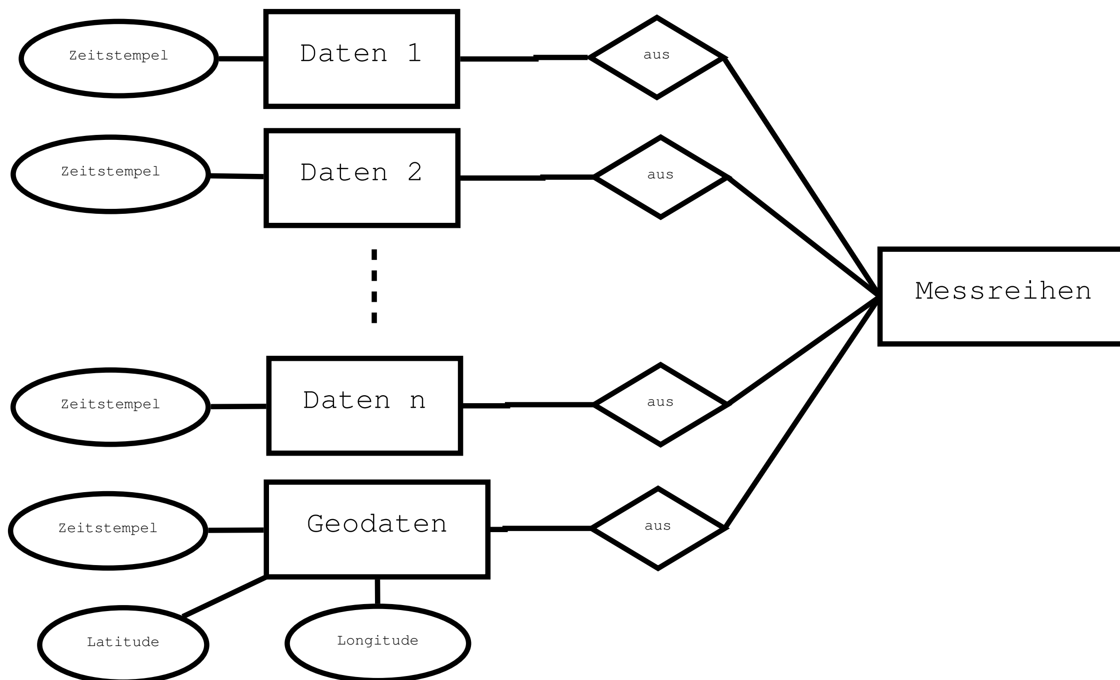


Abbildung 3.1: Das verallgemeinerte ER-Diagramm der Datenbankstruktur

3.1.2 Datenbankschnittstelle

Bei der Entwicklung der Datenbankschnittstelle geht es vor allem darum, ein klares Interface zu definieren. Dadurch soll es in Zukunft möglich sein ohne Anpassungen an den Kernmodulen neue Klassen zu erstellen, die den Zugriff auf weitere Datenquellen ermöglichen. Um diese Kapselung zu erreichen, wurde ein Interface [1] erstellt. Dieses definiert verschiedene Methodenarten.
Um zu überprüfen ob aus einem Abfrageobjekt überhaupt eine syntaktisch korrekte Abfrage generiert werden kann, muss die Methode *checkDataQuery* aufgerufen werden. Der semantische Sinn einer Abfrage kann durch die Funktion *countData* überprüft werden, welche die Anzahl der zu erwartenden Ergebnisse liefert. Die benötigten Daten werden von der Methode *getData* zurückgegeben. Bis hierhin wird für jeden Methodenaufruf ein intern verwendetes Abfrageobjekt als Parameter benötigt, welches in Abschnitt 3.1.3 näher erläutert wird. Zusätzlich existieren noch die Methoden *getLoggingRuns*, um die Informationen aller Messreihen und die Funktion *getTables* um die jeweiligen Struktur-

[1] va.Database.Database.java

informationen der Datenbank zu erhalten. Es wäre zwar möglich, dies über die zuerst erwähnten Routinen umzusetzen, allerdings erzeugt dies einen unnötigen Mehraufwand zum Erstellen und Verarbeiten der Datenobjekte.

3.1.3 Abfrageobjekte

Der Benutzer soll beim Verwenden der Software Abfragen anpassen können, um so die Ergebnisdatenmenge nach seinen Wünschen zu ändern. Dafür wurde eine spezielle Abfrageklasse entwickelt. Diese beinhaltet, aus welcher Quelle (Tabelle) welche Daten (Datenbankspalten) ausgelesen werden und wonach diese gefiltert werden sollen. Da die Implementierung eines vollständigen Abfrageneditors den Rahmen dieser Arbeit übersteigen würde, beschränken wir uns auf die für uns wichtigsten Funktionen zur Filterung von Ergebnisdatenmengen. Weitere Informationen hierzu sind in Abschnitt 4.2.1 zu finden.

3.1.4 Abfrageergebnisobjekte

Das Abfrageergebnisobjekt besteht aus einer Referenz zur Ursprungsabfrage und einer Liste, welche die aus der Abfrage resultierenden Daten beinhaltet. Im Abschnitt 4.2.1 wird genauer auf die Implementierung eingegangen.

3.2 Views

Wenn man Balkendiagramme zum einen und Straßenkarten zum anderen betrachtet, lässt sich schnell feststellen, dass es Arten der Darstellung von Daten gibt, die entweder nur schwer oder überhaupt nicht sinnvoll kombinierbar sind. Eine Menge von Darstellungen, bei denen eine Kombination der Anzeige sinnvoll ist, werden hier als *View* definiert. Sie soll dazu dienen aus verschiedenen Daten und deren unterschiedlichen Repräsentationsmöglichkeiten ein aussagekräftiges einheitliches Gesamtbild zu generieren. Die potentiellen Darstellungsarten einer View werden dabei *Layer* genannt. Mit diesen befasst sich der folgenden Abschnitt 3.2.1. Zusätzlich kann es für einige Views sinnvoll sein, eigene Einstellungen zu treffen oder selbst Daten abzufragen, auf die dann all ihre Layer Zugriff besitzen. Dies wird für die Auswahl der für das Geomatching verwendeten Daten benötigt.
Mit der eigentlichen Implementierung der Views befasst sich der Abschnitt 4.4.

3.2.1 Layer

Wie schon im vorherigen Abschnitt 3.2 erwähnt, ist ein Layer eine bestimmte Darstellungsart von Daten auf einer View. Die Darstellung einer Route (Daten = Streckenpunkte, Darstellungsart = Weg) auf einer Straßenkarte wäre hier ein Beispiel. Die Anzeige von *Points of Interests* (Daten = Orte, Darstellungsart = Icon) ein Weiteres. Beide lassen sich gleichzeitig auf ein einer einzigen View (Straßenkarte) anzeigen und es können Aussagen darüber getroffen werden (Liegen die Points of Interests in der nähe des Wegs?). Desweiteren ist hier auch die zeitgleiche Darstellung mehrerer Routen und Points of Interests denkbar.
Layer müssen also die Möglichkeit besitzen, Datenbankanfragen anzupassen (Welche Strecke wird gewählt?) und Einstellungen zu setzen (Wie groß soll die Streckenbreite dargestellt werden?). Anhand der abgefragten Daten und der gesetzen Konfigurationen generiert der Layer anschließend seine Visualisierung. Dabei kann eine View zeitgleich mehrere Layer darstellen. Nähere Informationen zu der Implementierung der Layer finden sich in Abschnitt 4.4

Kapitel 4

Implementierung

Im Kapitel 3 wurden bereits mit den generellen Designentscheidungen behandelt. Die praktische Umsetzung und Implementierung dieser Arbeit wird im Folgenden detailiert erläutert.

Da eine der Hauptaufgaben dieser Arbeit die Darstellung von Informationen auf Kartenmaterial ist, ist einer der wichtigsten Gründe für unsere Entscheidung für die Programmiersprache Java, die mögliche Verwendung der bereits existierenden JXMapKit Bibliothek. Sie ist Teil des Swingx-ws Projekts [swi], welches Funktionen zur Interaktion mit verschiedenen Web-Services zur Verfügung stellt. JXMapKit im Speziellen ermöglicht die Benutzung von Kartenmaterial, wie man sie von der Verwendung von Google Maps [goo] oder OpenStreetMap [ope] her kennt. Ein weiteres wichtiges Kriterium, welches für Java spricht, ist die Eigenschaft, dass die entwickelten Programme ohne Neukompilierung plattformübergreifend genutzt werden können.

Selbst bei gewissenhafter Programmierung größerer Softwareprojekten können Fehler beim Programmablauf auftauchen. Die Lokalisierung der Problemstellen im Quellcode gestaltet sich hier meist schwierig und zeitaufwendig. Um den Lokalisierungsprozess zur Fehlerfindung zu beschleunigen, bieten Loggingframeworks Methoden, die es ermöglichen den Programmablauf mitzuschneiden. Für Java ist Log4j [log] eine weit verbreitete Lösung hierfür. Im Gegensatz zum internen Logging-Framework von Java, bietet Log4j eine größere Auswahl an Konfigurationsmöglichkeiten, welche fein granular einstellbar sind. Dies betrifft unter anderem wie, in welchem Format und welche Nachricht-

en überhaupt mitgeschnitten werden. Die dabei resultierenden Ausgaben können auf unterschiedliche Weise gespeichert oder angezeigt werden. Aus diesen Gründen wird Log4j auch in dieser Arbeit verwendet.

Ein oft unterschätzter, aber nicht unwesentlicher Punkt ist die eigentliche Bedienbarkeit eines Programms. Um die Aufmerksamkeit des Benutzers schnell an die von ihm gewünschten Stellen zu lenken, wurden an den entsprechenden Stellen Icons eingefügt. Zusätzlich wurden GUI-Komponenten wiederverwendet, wenn dies sinnvoll möglich war. Eine weitere wichtige Funktion des Programms, die zu einer angenehmen Bedienbarkeit beiträgt, ist die Möglichkeit den aktuellen Stand des Programms im Dateisystem zu speichern und diesen zu einem späteren Zeitpunkt wieder herzustellen. Da zu Editier- und Debugzwecken eine für den Menschen lesbare Speicherdatei wünschenswert ist, wurde auf die Verwendung des *Serializeable* Interfaces unter Java verzichtet und stattdessen eigene Speicher- und Laderoutinen geschrieben. Diese basieren auf der Javaklasse *java.util.Properties* [javb], bei der alle Eigenschaften als Key-Value Zeichenketten Paare gespeichert werden. Hierfür wurde das Interface *Store* erstellt, welches die *toStore()* Methode zum Speichern und die *fromStore(Store store)* Methode zum Laden voraussetzt. Jedes zu speichernde Objekt muss diese implementieren. Die Schlüssel-Zeichenketten müssen eindeutig sein und werden daher hierarchisch aufgebaut. Wie im folgenden Beispiel gezeigt, werden die verschiedenen Hierarchieebenen mit Punkten getrennt:

EigenschaftDesProgramms
Views.ViewId.EigenschaftDerView
Views.ViewId.Layer.LayerId.EigenschaftDesLayers

Die entwickelte Software besteht aus drei großen Teilbereichen (ähnlich dem Modell-View-Controller Konzept). Zunächst musste eine Schnittstelle geschaffen werden, über die Daten aus einer Quelle ausgelesen und diese dem Rest der Applikation zur Verfügung gestellt werden können. Dieser Bereich wird in Abschnitt 4.2 erarbeitet. Um die Software steuern zu können, sind im Backend entsprechende Funktionen implementiert worden, die in 4.3 beschrieben sind. Der letzte Punkt ist die eigentliche graphische Ausgabe des Programms. Diese lässt sich in zwei Teile gliedern. Die allgemeine GUI, auf die in 4.1 eingegangen wird und die einzelnen Views, die in 4.4 genauer betrachtet werden.

4.1 GUI

Um einen generellen Überblick über das Programm zu geben, werden zunächst die graphische Benutzerschnittstelle und die hierrüber erreichbaren Funktionen erläutert. In Abbildung 4.1 sind die einzelnen Teileinheiten der GUI markiert.

Im Hauptmenü (1) befinden sich drei Untermenüs. Unter dem Punkt *File* lässt sich der gegenwertige Stand sichern, sowie ein bereits gespeicherter Zustand wiederherstellen. Beim Laden werden dabei alle aktuellen Einstellungen verworfen. Der Menüpunkt *View* stellt Funktionen zum Erstellen, Löschen und Umbenennen der Views zur Verfügung. Zudem gibt es hier auch die Möglichkeit, die derzeitige graphische Ausgabe als Bilddatei zu speichern. Abbildung 4.6 zeigt das gespeicherte Bild einer Mapview. Die Views können an dieser Stelle weitere Menüpunkte hinzuzufügen. Bei *Layer* lässt sich der aktuell markierte Layer zwischenspeichern und eine Kopie in derselben oder in einer View des gleichen Typs wieder einfügen.

Jede derzeit verwendete View ist über ihren entsprechenden Tabreiter (2) zu erreichen. Dieser beinhaltet den Namen der View und ein von ihr abhängiges, fest definiertes Icon. Alle Einstellung einer View (Layerverwaltung, Konfigurationen und Datenverwaltung) sind über den entsprechenden Tab (3) im linken Teil des Programmfensters zu erreichen. Die aktive Einstellungskategorie wird anschließend in (4) angezeigt. Auf der rechten Seite (5) befindet sich ein Informationspanel, in welchem die View weitere relevanten Informationen präsentieren kann. Im unteren Teil findet sich der Loggingbereich. Hier werden die geloggten Nachrichten während der Programmnutzung angezeigt.

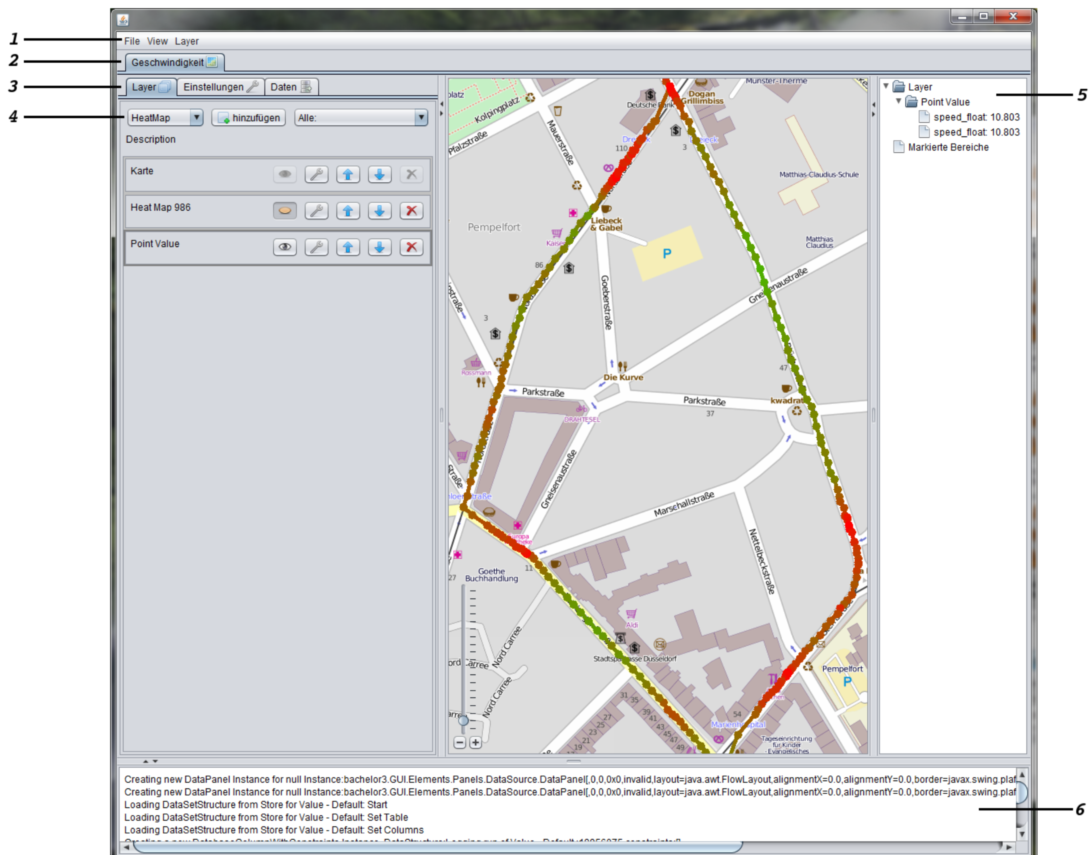


Abbildung 4.1: Überblick GUI

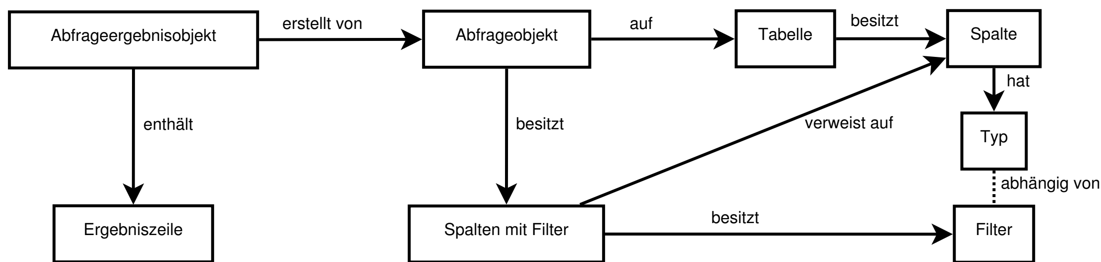


Abbildung 4.2: Vereinfachtes Klassenmodell der Datenabfrage

4.2 Daten

4.2.1 Abfrageklassen

Damit das Programm unabhängig von der eigentlichen Datenquelle ist und trotzdem flexible Anfrage von Daten unterstützt, wurden spezielle Abfrage- und Abfrageergebnis-Klassen entwickelt. Mit deren Hilfe können Views und Layer vom Benutzer gefilterte Datenanfragen senden und die Ergebnisse dieser verarbeiten.

Abbildung 4.2 zeigt ein vereinfachtes Klassendiagramm, welches die Zusammenhänge, der in der Abfrage beteiligten Klassen bildlich darstellt. Das Abfrageobjekt beinhaltet alle nötigen Informationen um daraus eine Abfrage zu erzeugen, die anschließend die gewünschten Daten als Ergebnis liefert. Die Parameter der Abfrageklasse sind vom Nutzer nach Bedarf über die GUI konfigurierbar. Da davon auszugehen ist, dass die meisten Daten in der Tabellenform vorliegen, ist das Abfrageobjekt entsprechend ausgelegt. Hier wird definiert welche Spalten, aus welcher Tabelle, mit welchen Filtereinstellungen abgefragt werden sollen. Bevor ein Abfrageobjekt erstellt wird, sind die Tabellenstrukturen inklusive Spaltentypdefinitionen über einen Aufruf des Dateninterfaces bekannt (Siehe Kapitel 3.1.2 - *getTables*). Bei den Spalten sind je nach Spaltentyp verschiedene Arten von Filter möglich. Bei numerischen Typen sind dies Größeneinschränkungen ($<, \leq, >, \geq, =$) mit anzugebenden Vergleichswerten. Bei Zeichenketten wird dafür der aus SQL bekannte Vergleichsoperator LIKE [sql] genutzt. Somit sind hier auch die SQL-Wildcardzeichen % und _ gestattet. Bei jeder Spalte ist es zudem möglich, mehrere dieser Einschränkungen zu definieren. Dabei können Gruppen aus mehreren Filtern gebildet werden, deren Gruppenelemente mit dem AND-Operator verbunden

werden. Die Gruppen selbst werden anschließend mit dem OR-Operator verknüpft.
Beispiele:

1. $(x > 1 \&\& x \leq 3) || (x \geq 5 \&\& x < 10)$

2. (x LIKE "%hinten"&& x LIKE "vorne%"&& x LIKE "%zwischen%") || (X LIKE "hinten")

In Beispiel 1 sind alle Einträge gesucht, bei denen die Spalte x größer 1 und kleiner gleich 3 oder alternativ x größer gleich 5 und kleiner 10 ist. Beispiel 2 sucht nach einer Zeichenkette, die einen Treffer für den regulärem Ausdruck ((vorne.*zwischen.*)?hinten) erzeugt.

Komplexere Filterkombinationen werden in dieser Arbeit nicht benötigt und wären auch nicht mit vertretbarem Mehraufwand lösbar gewesen.

Weiterhin ist es möglich, eine Spalte eines Abfrageobjekts bei einer Spalte eines anderen Abfrageobjekts zu registrieren. Alle getroffenen Filtereinstellungen werden anschließend bei der registrierten Spalte hinzugefügt und wenn nötig aktualisiert oder wieder entfernt. Falls bei der Abfrage keine Spalten ausgewählt sind, wird davon ausgegangen, dass im Ergebnis alle in der Zieltabelle vorhandenen Spalten erwünscht sind.

Das Abfrageergebnisobjekt besitzt eine Referenz zu der jeweiligen Abfrage und eine Ergebnisliste. Dabei entspricht jeder Eintrag in der Liste einem Ergebnistupel, mit den Spalteninhalten der jeweiligen Ergebniszeile.

4.2.2 Datenbank

Da die zur Zeit auszuwertenden Daten in einer MySQL Datenbank liegen, wurde eine entsprechende Klasse gegen das in Kapitel 3.1.2 beschriebene Datenimport-Interface implementiert. Dafür war es nötig die Tabellen- und Spaltenstrukturdaten aus der Datenbank auszulesen und aus dem Abfrageobjekt 4.2.1 einen SQL-Query zu generieren. Grundsätzlich können durch das Programmieren gegen das Datenimport-Interface Klassen

erstellet werden, die es ermöglichen weitere Datenquellen einzubinden (z.B. andere Datenbanken, CSV-Dateien). Falls dies in Zukunft notwendig wird, kann das Programm dadurch mit geringem Aufwand angepasst werden.

4.3 Backend

Das Backend besteht zum Großteil aus Klassen die von *javax.swing.JPanel* [java] abgeleitet sind und die für das jeweilige Panel zuständige Programmlogik besitzen. Hieraus ergibt sich ein hierarchisches Modell, bei der die Klasse *MainTabPanel* an der Spitze steht. Sie verwaltet die Liste der verwendeten Views und ist für das Speichern und Laden zuständig. Die Kommunikation der einzelnen Komponenten läuft abwärts der Hierarchieebenen mit Methodenaufrufen und aufwärts mit *PropertyChange* [javc] Nachrichten. Diese funktionieren nach dem Prinzip des Observer Entwurfsmusters, bei dem ein Beobachter die Änderungen eines Subjekts registriert und entsprechend reagiert. Da einige Methoden von vielen Stellen im Programmcode zugreifbar sein sollen (wie z.B. Anfragen an die Datenquelle), wurde die Klasse Controller als Singleton implementiert und die erfolderlichen Funktionen zur Verfügung stellt.

4.4 Views

Wie bereits in Kapitel 3.2 beschrieben, vereinigen Views verschiedene Layertypen, bei denen eine gleichzeitige Darstellung möglich und sinnvoll ist. Daher beinhaltet jede View eine Liste von unterstützten Layertypen, die zu ihrer Anzeige hinzugefügt werden können. Im laufendem Programm können mehrere Views über das Menü erstellt, angezeigt, gelöscht und deren Namen geändert werden. Ein Wechsel der aktuellen View ist dabei über Tabs möglich. Die Tabreiter beinhaltet dafür den jeweiligen Namen und ein passendes Icon. Zudem bieten die Views eine Layerverwaltung, mit der Layer hinzugefügt, gelöscht oder ihre Position innerhalb der View geändert werden können.

4.4.1 Einstellungen und Daten

Da sowohl die Views, als auch die Layer Einstellungen treffen und eigene Datenabfragen verwenden, wurde eine gemeinsame Oberklasse implementiert, die die entsprechenden Methoden zur Verfügung stellt. Der Aufbau der Abfrageobjekte wurde bereits in Kapitel 4.2.1 behandelt. Für die Konfigurationen wurde eine Konfigurationsklasse geschrieben, welche für das Setzen, Auslesen, Speichern und Laden von Konfigurationen zuständig ist. Diese muss für die entsprechenden Datentypen abgeleitet und angepasst werden. Spezielle GUI-Klassen sind anschließend für die Anzeige zuständig. Verwendet werden z.B. Werteregler, Eingabefelder, Checkboxen, Auswahllisten und eine Farbauswahl.

4.4.2 Layer

Zusätzlich zu dem allgemeinen Einstellungs- und Dateninterface, welches die Layer mit den Views gemein haben, besitzen sie noch weitere Eigenschaften. Unter anderem erhält jeder zu einer View hinzugefügter Layer eine Priorität, die abhängig von seiner Position in der Layerliste der View ist. Diese ist in der GUI über Pfeil-Buttons einstellbar. Jeder in der Liste tieferliegende Layer zeichnet über die in der Ausgabe Darunterliegenden. Somit gilt, dass je tiefer er sich in der Liste befindet, umso höher ist seine Priorität. Um Layer temporär zu deaktivieren, gibt es weiterhin die Möglichkeit sie unsichtbar zu schalten. Sie werden dabei weder berechnet, noch dargestellt. Da einige Layer umfangreiche Kalkulationen durchführen müssen, besitzen sie zusätzlich eine Methode, die angibt, ob vor der eigentlichen Anzeige Berechnungen notwendig sind. Diese werden dann in jeweils einem eigenen Thread ausgeführt.

4.4.3 Mapview

Die Kartenansicht ist dafür zuständig, Daten auf unterschiedliche Weisen zeitgleich auf einer Karte darzustellen.
Eines der größten Probleme ist hierbei die notwendige Zuweisung der Informationen auf die entsprechenden Geokoordinaten. Da in den untersuchten Messreihen nur in den

seltensten Fällen das darzustellende Datentupel die benötigten Geopositionen enthält, werden zur Zuweisung Zeitstempel verwendet, die sowohl bei den anzuzeigenden Daten, wie auch bei den Geopositionsdaten vorhanden sein müssen. Für die Abbildung der Daten auf Geopositionen stellen wir verschiedene Algorithmen zur Verfügung, die bei dem jeweiligen Layer auswählbar ist. Hierbei ist es möglich die zeitlich vorherige oder folgende Geoposition zu verwenden oder den geographischen Mittelpunkt dieser Beiden zu verwenden. Bei Bedarf lassen sich hier in Zukunft mit geringem Aufwand auch weitere Algorithmen hinzufügen. Aus welcher Datenbanktabelle die Geodaten stammen und welche Filter gelten, lässt sich über das Daten-Tab der Mapview einstellen. Desweiteren wird, falls eine Messreihe in einem Layer nicht in der Geodatenabfrage berücksichtigt wurde, diese automatisch bei der Abfrage hinzugefügt und anschließend aktuell gehalten.

An jedem Punkt der Karte ist es zudem möglich, über ein Kontextmenü, die an diesem Punkt dargestellten Daten aller Layer einzusehen. Zusätzlich wurden Markierungsfunktionen implementiert, mit deren Hilfe es möglich ist, mehrere rechteckige Gebiete auszuwählen. Aggregierte Informationen (wie Maximal- und Minimalwerte je Layer) zu diesen Gebieten sind dabei über das Kontextmenu abrufbar.

Wenn sich der Mauszeiger über der Karte befindet und die ALT-Taste gehalten wird, erscheinen im Informationspanel (vgl. Abb. 4.3) alle Informationen, die die Layer und die markierten Gebiete für diesen Punkt zurückgeben. Bei Letzteren werden dabei zusätzlich zu den aggregierten Informationen alle darin enthaltenen Daten angezeigt. Über die Auswahl im Informationspanel lässt sich dann auf der Karte der Quellort eines Werts anzeigen. Somit können auffällige Daten in einem markierten Gebiet schnell lokalisiert werden.

Mapview-Layer

Da einige Kartenlayer sehr umfangreiche Berechnungen benötigen, hat sich schnell gezeigt, dass ein Zwischenspeichern der Ergebnisse und der graphischen Ausgabe der meisten Layer sinnvoll ist. Dadurch wird, abhängig von dem eigentlichen Layer und den nötigen Berechnungen, der Speicherverbrauch zwar erhöht, die benötigte Rechenzeit bei der

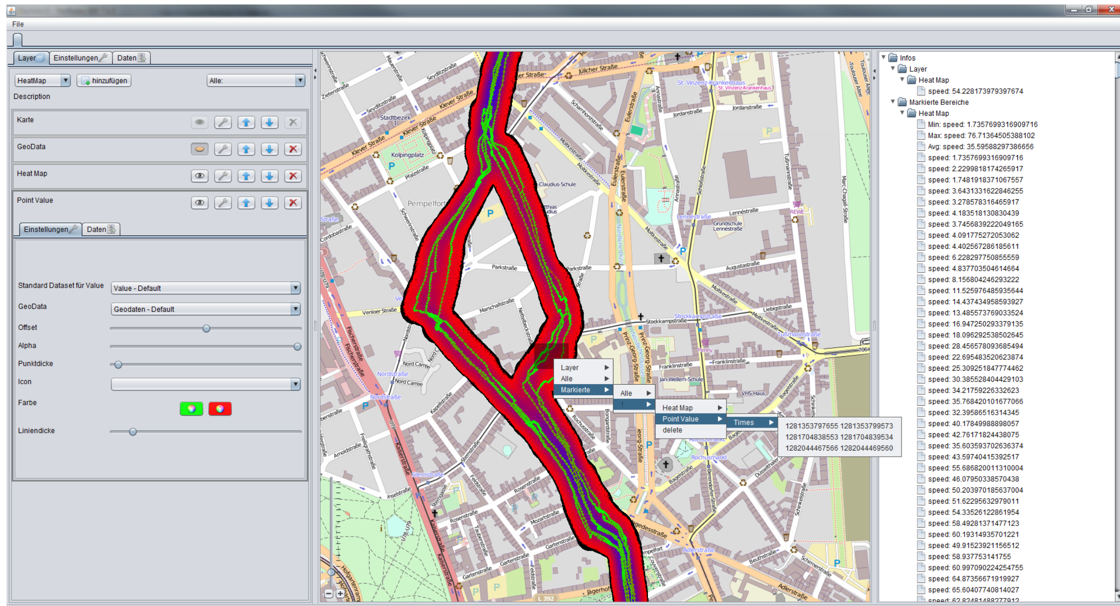


Abbildung 4.3: Mapview mit markiertem Gebiet und geöffnetem Kontextmenü

Benutzung allerdings drastisch verringert. Damit bei geringen Verschiebungen des sichtbaren Kartenbereichs das zwischengespeicherte Bild nicht immer neu berechnet werden muss, ist es möglich bei den Layereinstellungen anzugeben wieviel vom nicht dargestellten Bereich bei der Erstellung der Ausgabe mitberechnet wird.

Um Mapview-Layer zu verschieben bieten diese die Möglichkeit je einen Offset in horizontaler und vertikaler Richtung einzustellen. Desweiteren kann der Alpha-Wert, welcher die Transparenz angibt, frei gewählt werden. Zusätzlich ist es möglich, aus der von der Mapview zur Verfügung gestellten Liste von Geodatenabfragen, eine Entsprechende für den Layer auszuwählen.

Im Folgenden gehen wir auf die existierenden Layer der Mapview ein.

Kartenlayer Dieser Layer stellt die Karte selbst dar. Alle darunterliegende Layer werden weder angezeigt, noch berechnet. Das für die Darstellung verwendete Kartenmaterial ist dabei auswählbar. Aufgrund der freien Verfügbarkeit und der ausreichenden Genauigkeit der Daten wird standardmäßig den Kartenserver und das Kartenmaterial von OpenStreetMap [ope] verwendet. Zusätzlich wird ein Kartenserver des Lehrstuhls

zur Auswahl angeboten (http://tilecn.cs.uni-duesseldorf.de), welcher eine höhere Zoomstufe erlaubt, allerdings nur örtlich beschränktes Kartenmaterial besitzt.

Farblayer Der Farblayer bietet die Möglichkeit, die unter ihm liegenden Layer farblich abzugrenzen. Die Farbe und der Alpha-Wert können dabei angepasst werden. Dies dient vorallem dazu den Hintergrund beliebig abzudunkeln um somit die darüberliegende Layer besser sichtbar zu machen.

Punktlayer Der Punktlayer ist die Oberkategorie für alle Layer die nur Punkte darstellen und bietet somit spezifische Funktionen für die von ihr abgeleiteten Layer. Dabei können unter anderem die Punktfarbe und Punktgröße eingestellt werden. Zusätzlich ist es möglich statt einem farbigen Punkt auch ein Icon auszuwählen. Die Punkt- bzw Icongröße skaliert dabei mit der jeweiligen Zoomstufe.

Geodatenlayer Der Geodatenlayer ist ein Punktlayer, welcher alle Punkte der ausgewählten Geodatenabfrage auf der Karte anzeigt. Die hier dargestellten Werte werden zum Mapmatching der anderen Layer genutzt.

Punktdatenlayer Der Punktedaten-Layer stellt eine Liste von numerischen Werten der gleichen Kategorie auf der Karte dar. Die Farbe legt dabei fest, wie nah sich der jeweilige Wert am Minimum oder Maximum der anzuzeigenden Daten befindet. Desweiteren ist es möglich diese Punkte durch Linien zu verbinden. Die Linie wird dabei mit einem entsprechendem Farbverlauf zwischen zwei in der Liste aufeinanderfolgenden Punkten dargestellt. Die Farben für den kleinsten und größten Wert, sowie auch die Liniendicke sind bei dem Layer frei einstellbar.

Heatmaplayer Der Heatmaplayer hat die gleichen Datenanforderungen wie der Punktdatenlayer. Hierbei werden allerdings nicht nur die Punkte dargestellt, sondern Wertefelder auswählbarer Größe um diese berechnet. Das Ergebnis, welches aus diesen Wertefeldern bzw. deren Überlagerungen entsteht, wird dabei farblich dargestellt.

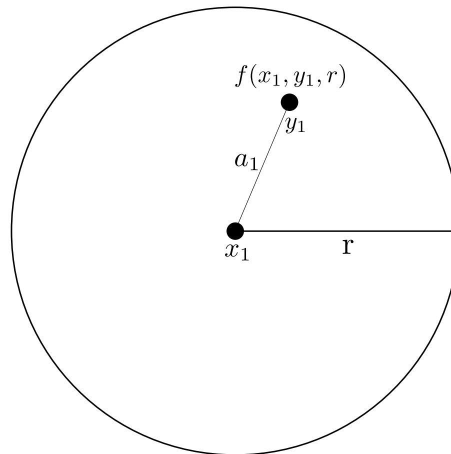


Abbildung 4.4: Berechnung des Wertefelds eines Datenpunkts für den Heatmaplayer

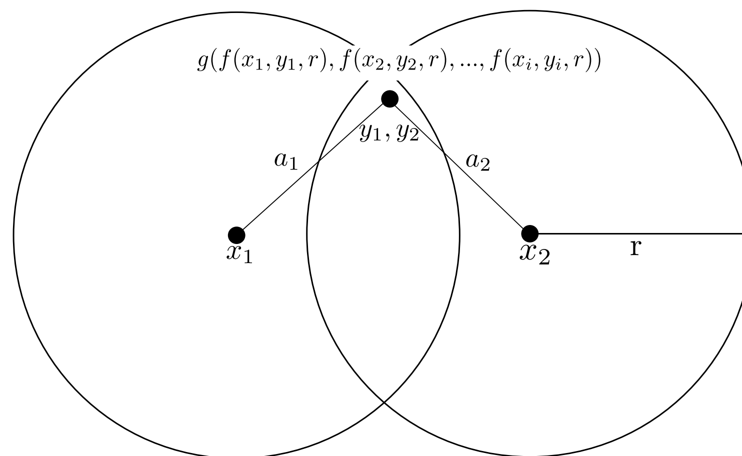


Abbildung 4.5: Berechnung bei Überschneidung von Wertefelder für den Heatmaplayer

Daraus resultiert eine Darstellung, die Aussagen über jeden Punkt auf der Karte in Abhängigkeit von der sich in seiner Nähe befindlichen Daten erlaubt. Ein auszuwählender Algorithmus berechnet für jeden Punkt ein Wertefeld, welches von dem Ursprungswert und der Position des Punkts, sowie auch von einem einstellbaren Radius abhängig ist. Abbildung 4.4 veranschaulicht dies. Hierfür stehen linear fallende und konstante Funktionen zur Verfügung. Das Ergebnis für sich überlagernde Wertefelder wird über einen anderen Algorithmus bestimmt (Abb. 4.5). Dabei können Durchschnitts-, Maximal-, Summen-, Zähl- oder *letzter Wert*-Funktionen ausgewählt werden. Für die Laufzeit der

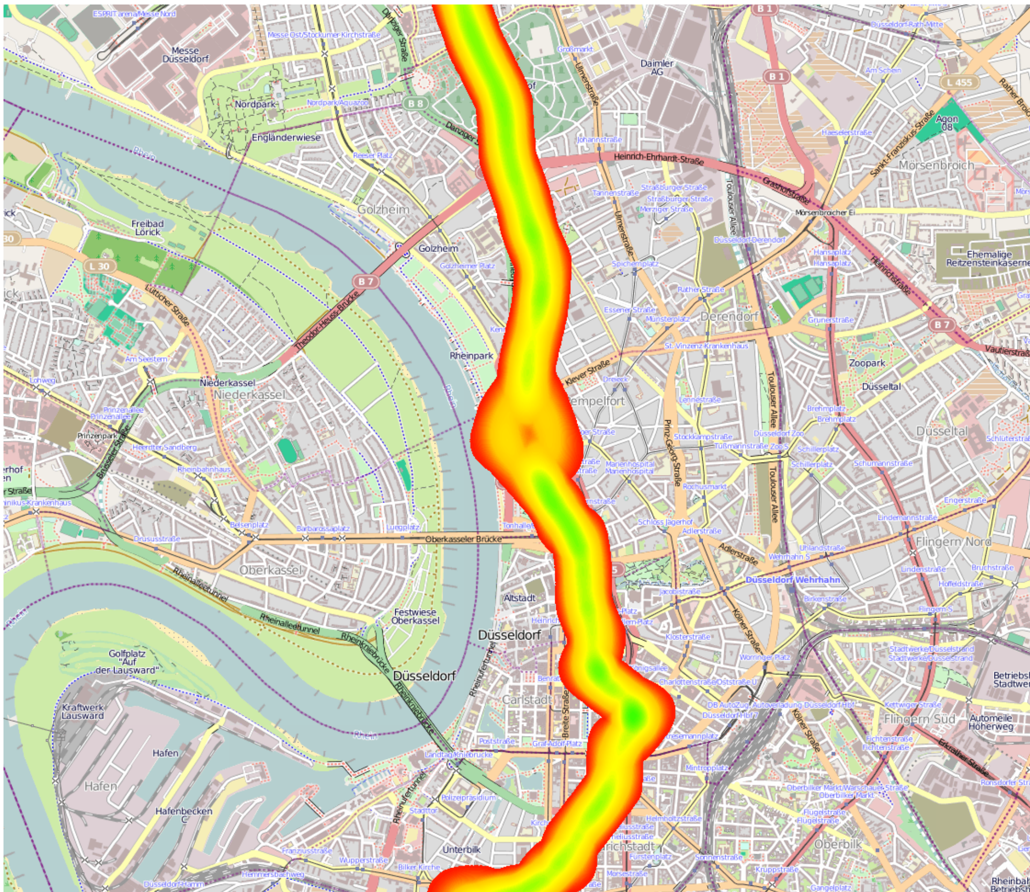


Abbildung 4.6: Beispiel einer Heatmap

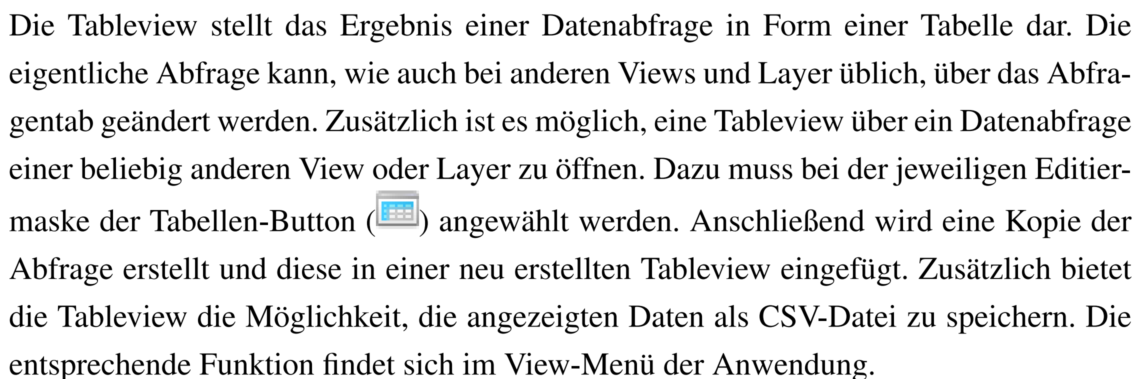
Heatmaplayerberechnung gilt hierbei:

$$f \in \Theta(n * 2r^2 * (x + y)) \tag{4.1}$$

wobei x die Laufzeit des Wertefeldberechnungsalgorithmus und y die Laufzeit des Wertefeldvereinigungsalgorithmus darstellt. Bei den derzeit implementierten Algorithmen sind diese aus $O(1)$. Die für die Karte berechneten Werte werden anschließend farblich dargestellt. Zur Farbgebung lassen sich hier, anders als bei den Punktlayer, Graphiken mit Farbverläufen auswählen. Dabei entspricht der kleinste Wert dem Farbwert an dem linken Ende der Graphik und der größte dem am rechten Ende. Die Zwischenwerte werden entsprechend ausgelesen. Das Umdrehen des verwendeten Gradienten ist dabei auch möglich. Abbildung 4.6 zeigt ein Beispiel für eine Heatmap.

Heatmapdifferenzlayer Dieser Layer subtrahiert für jeden Punkt auf der Karte die Werte aus zwei anzugebenden Heatmaplayer. Daraus lassen sich die Werteunterschiede gut sichtbar hervorheben. Zusätzlich ist es auch möglich, nur die Gebiete der Überschneidungen darzustellen. Die Berechnungen der beiden Heatmaplayer werden hierbei auch durchgeführt, wenn diese auf unsichtbar gestellt sind oder sie sich unterhalb der Karte befinden.

4.4.4 Tableview

Die Tableview stellt das Ergebnis einer Datenabfrage in Form einer Tabelle dar. Die eigentliche Abfrage kann, wie auch bei anderen Views und Layer üblich, über das Abfragentab geändert werden. Zusätzlich ist es möglich, eine Tableview über ein Datenabfrage einer beliebig anderen View oder Layer zu öffnen. Dazu muss bei der jeweiligen Editiermaske der Tabellen-Button () angewählt werden. Anschließend wird eine Kopie der Abfrage erstellt und diese in einer neu erstellten Tableview eingefügt. Zusätzlich bietet die Tableview die Möglichkeit, die angezeigten Daten als CSV-Datei zu speichern. Die entsprechende Funktion findet sich im View-Menü der Anwendung.

4.4.5 2DPlotview

Die 2DPlotview stellt Daten in einem zweidimensionalen Koordinatensystem dar. Ihre Layer werden hierbei, äquivalent zur Mapview, übereinander gezeichnet. Die X und Y Achsengröße wird den anzuzeigenden Daten entsprechend automatisch angepasst. Für die Anzeige der 2DPlotview wird das Framework JFreeChart verwendet. Dieses ist frei verfügbar, bietet eine Vielzahl unterschiedlicher Darstellungstypen mit vielen Funktionen und eine ausführliche Dokumentation. In Abbildung 4.7 sieht man eine 2DPlotview, auf der die Bandbreite und Droprate in up- und down-Richtung gegen die Zeit abgetragen ist.

Punktlayer Der Punktlayer der 2DPlotview stellt Daten als Punkte im Koordinatensystem dar. Diese lassen sich über die entsprechende Konfiguration mit Linien verbinden. Zusätzlich lassen sich Größe, Farbe und Art der Punkte einstellen.

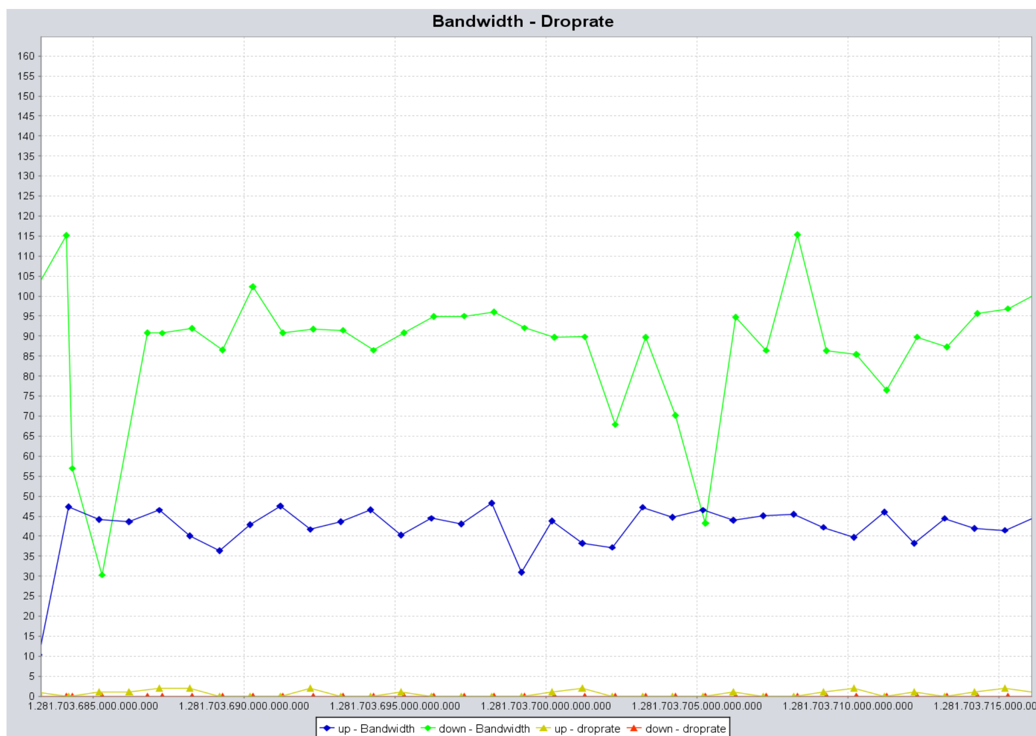


Abbildung 4.7: Beispiel einer 2D Plotview

Kapitel 5

Auswertung

Bevor die Messreihen genauer betrachtet werden, ist es wichtig diese zeitlich einordnen zu können. In der Tabelle 5.1 sind hierfür die verwendeten Messreihen mit Id, Datum, Wochentag, Start- und Endzeit aufgelistet. Dabei fällt auf, dass diese etwa zur gleichen Zeit aber an Unterschiedlichen Wochentagen stattfanden.

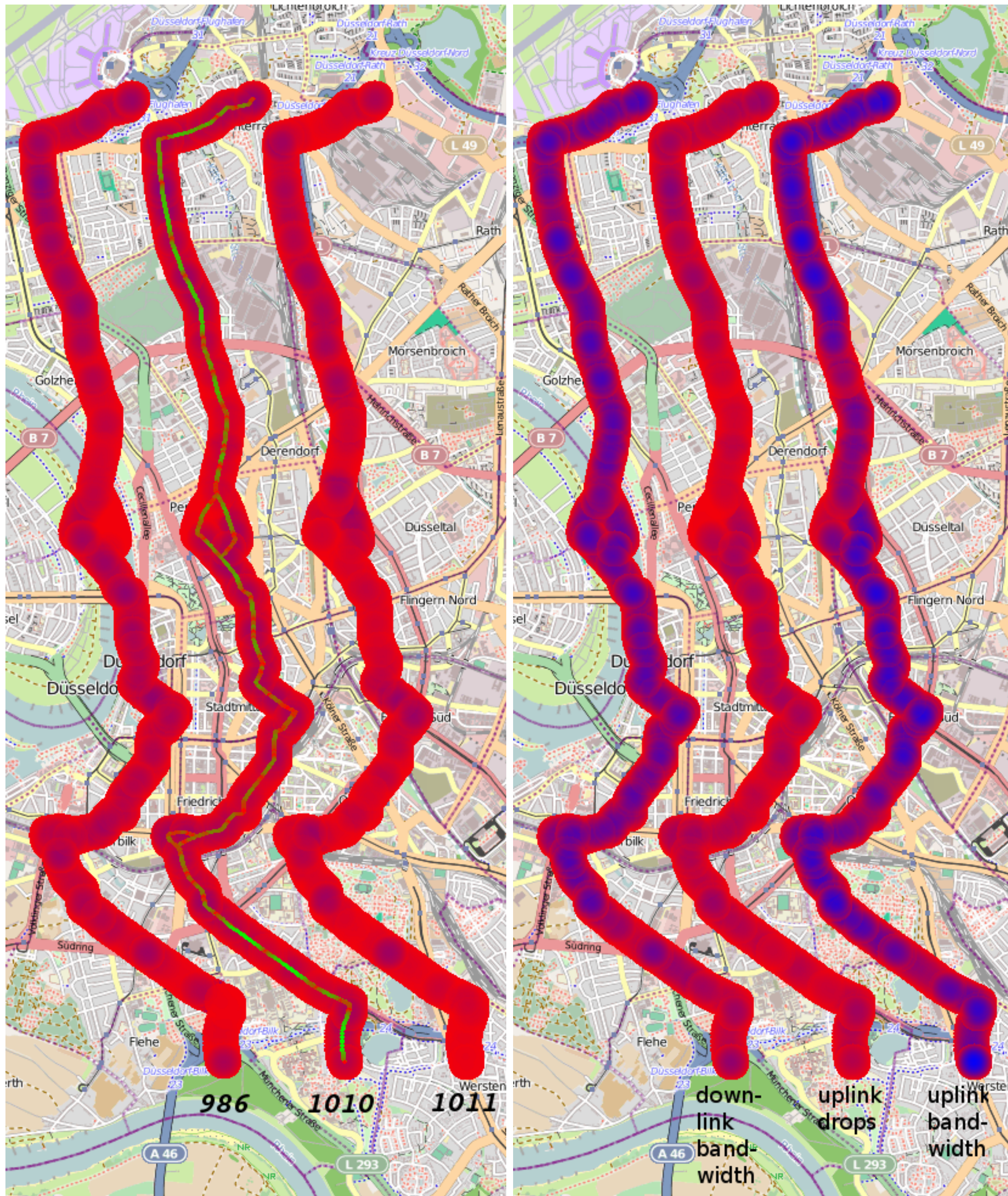
Zunächst bietet die Abbildung 5.1(a) einen Vergleich der gemessenen Dropraten beim Uplink der verschiedenen Messfahrten. Die linke und rechte Heatmap sind hierbei um einen Offset verschoben. Das Erste was hier auffällt, ist dass bei der Messreihe 1011, gerade im Innenstadtbereich die Droprate im Vergleich zu den anderen Fahrten wesentlich geringer ist. Eine Ausnahme bilden hier die Bereiche um die Bahnhöfe Düsseldorf Bilk und Düsseldorf Hbf. Dies wird daran liegen, dass die Messfahrt an einem Sonntag durchgeführt wurde. Hier ist das Personenaufkommen und dadurch auch die Menge an mobilen Endgeräten, die das Mobilfunknetz belasten, in der Innenstadt wesentlich geringer als an Wochentagen. Zusätzlich fällt auf, dass die Dropraten außerhalb der Innenstadt über alle Messreihen hinweg auf einem etwa gleich hohen Niveau liegen. Das könnte unter

Id	Datum	Wochentag	Startzeit	Endzeit
986	09.08.2010	Samstag	12:41	14:04
1010	13.08.2010	Mittwoch	14:04	15:36
1011	17.08.2010	Sonntag	12:24	13:57

Tabelle 5.1: Messreiheninformationen

anderem daran liegen dass sich dort mit eine höhere Geschwindigkeit bewegt wird. Die gemessenen Geschwindigkeiten der Messreihe 1010, lassen sich auf dem auf dem Punktelayer ablesen, der oberhalb der Dropraten-Heatmap dargestellt wird.

In der Abbildung 5.1(b) ist die Messreihe 1010 abgetragen. Links ist die Bandbreite im Downlink, Rechts die Bandbreite im Uplink und in der Mitte die Droprate des Uplinks zum Vergleich dargestellt. Die Droprate des Downlinks beträgt in der Messreihe fast immer 0 und wird daher hier nicht mit dargestellt. Hier ist zu beachten, dass bei unterschiedlichen Heatmaps gleiche Farbwerte nicht zwingend die gleichen Werten repräsentieren. Beim Vergleich der durchschnittlichen Bandbreite und der durchschnittlichen Droprate fällt auf, dass diese positiv korrelieren. Weiterhin ist kein klarer Zusammenhang zwischen dem Up- und Downlink zu erkennen. Hier sind weitere Analysen nötig um genauere Aussagen über die Zusammenhänge treffen zu können.



(a) Vergleich der Dropraten der Messreihen 986, 1010 und 1011

(b) Von Links: Bandbreite im Downlink, Droprate im Uplink, Bandbreite im Uplink

Abbildung 5.1: Bandbreite und Droprate im Up- und Downlink

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

In dieser Bachelorarbeit wurde ein Programm implementiert, um Daten aus Mobilfunkmessreihen auf Karten darzustellen. Zunächst wurde die graphische Ausgabe in zwei logische Einheiten unterteilt. Zum Einen die Views, welche eine Zusammenfassung gleichartiger Darstellungen entsprechen und die Grundlage für deren graphische Ausgabe schaffen. Zum Anderen die Layer, welche die verschiedenen Darstellungsarten der jeweiligen Views sind. Um die für die Anzeige benötigten Daten zu erhalten, wurden anschließend Abfrageobjekte und Abfrageergebnisobjekte erstellt, mit denen die Views und Layer die für sie nötigen Daten definieren und nach einer Anfrage an die Datenbankschnittstelle auch erhalten. Gegen dieses Interface wurde eine Klasse programmiert, die den Datenzugriff auf MySQL-Datenbanken erlaubt. Welche Daten die Abfrageobjekte liefern und wie diese gefiltert werden, ist über die GUI konfigurierbar.

Als Views wurden eine Tabellen, 2D-Koordinatenview und Kartenview erstellt. Mit der Tabellenview lassen sich Abfrageergebnisse in Tabellen darstellen. Diese sind nach Spalten sortierbar und lassen sich als CSV-Datei exportieren. Auf der 2D-Koordinatenview können verschiedene Datensätze zeitgleich auf einem zweidimensionalem Koordinatensystem dargestellt werden. Die Darstellung der einzelnen Datenpunkte, sowie ob diese verbunden werden ist in den jeweiligen Layereinstellungen konfigurierbar. Die Karten-

view bildet Daten auf einer Karte ab und bietet eine Vielahl unterschiedlicher Layertypen. Beim Kartenlayer, der pro Kartenview genau einmal vorhanden ist, lässt sich das verwendete Kartenmaterial auswählen. Für die eigentliche Darstellung der Daten existieren mehrere Möglichkeiten. Zum Einen gestattet es der Punktlayer, die abgefragten Informationen als farbige Punkte oder Icons auf die Karte zu zeichnen. Ob diese verbunden werden sollen und welche Farbe oder Icons sie besitzen, lässt sich bei den Layereinstellungen konfigurieren. Die Farbe der einzelnen Punkte gibt dabei Auskunft über den entsprechenden Wert. Zum Anderen lassen sich Daten als Heatmaps visualisieren. Auch hier gibt es verschiedene Möglichkeiten, die Ausgabe durch die Auswahl der verschiedenen Berechnungsalgorithmen, der gewünschten Größe und des Farbverlaufs anzupassen. Damit bei geringen Verschiebungen des sichtbaren Kartenbereichs die Heatmaps nicht sofort neu berechnet werden müssen, werden die Ergebnisse und die graphische Ausgabe zwischengespeichert. Mit dem Heatmapdifflayer, eine Unterkategorie des Heatmaplayers, lassen sich die Differenz zweier Heatmaps auf der Karte darstellen, um so Unterschiede oder Gemeinsamkeiten schnell erkennen zu können. Um die für das Geopositionsmatching verwendeten Daten sichtbar zu machen, existiert der Geodatenlayer, welcher es erlaubt die Geopositionen als Punkte auf der Karte darzustellen. Damit verschiedene Layer voneinander abgrenzbar sind, existiert ein Farblayer, der eine Farbschicht darstellt. Je nach Transparenz der ausgewählten Farbe lässt sich somit der Hintergrund abdunkeln oder ganz überdecken. Zusätzlich bietet die Kartenview die Möglichkeit, mehrere rechteckige Gebiete zu markieren. Über das Informationspanel können die verwendeten und berechneten Daten an jedem Kartenpunkt angezeigt werden. Falls sich dort Gebietsmarkierungen befinden, werden auch Informationen, wie Minimal-, Maximal- oder Durchschnittswert des entsprechenden Gebiets zur Verfügung gestellt.

6.2 Ausblick

Aufgrund der begrenzten Zeit konnten nicht alle Ideen und Optimierungsmöglichkeiten, die sich während der Bearbeitung ergeben haben, umgesetzt werden. Diese werden im Folgenden aufgelistet:

- Zur Zeit verwendet das Programm für jedes Abfrageobjekt eine eigene Datenbankabfrage und hält die Ergebnisse im Speicher. Falls bei den Views oder den Layers eine Abfrage einer anderen gleicht, wäre hier durch eine entsprechende Programmierung eine Speicherersparnis zu erreichen.

- Momentan wird nur eine Datenquelle für das gesamte Programm verwendet. Hier wäre es wünschenswert, dass Views und Layer diese zwar standardmäßig benutzen, allerdings auch die Möglichkeit besitzen, alternative Quellen anzugeben.

- Das Erstellen einer weiten Auswahl an unterschiedlichen Diagrammarten mit JFreeChart.

- Ein simulierter Ablauf einer Messreihenfahrt, bei der die auftretenden neuen Messdaten als Events auf der Karte angezeigt werden.

- Eine Usability-Analyse und die daraus resultierenden benötigten Anpassungen, würden die graphische Oberfläche dieser Arbeit weiter abrunden.

Literaturverzeichnis

[Buz00] BUZIEK, Gerd [.: *Dynamische Visualisierung : Grundlagen und Anwendungsbeispiele für kartographische Animationen.* 2000

[cor] *How to make heat maps Corunet. El Blog*. http://blog.corunet.com/how-to-make-heat-maps/.

[Goe10] GOEBEL, Norbert: *Trace-basierte Simulation von Mobilfunkcharakteristiken für die Fahrzeug-zu-Fahrzeug Kommunikation*, Department of Computer Science, Heinrich Heine University Düsseldorf, Diplomarbeit, August 2010

[goo] *Google Maps*. http://maps.google.com/.

[java] *JPanel (Java Platform SE 6).* http://docs.oracle.com/javase/6/docs/api/javax/swing/JPanel.html.

[javb] *Properties (Java Platform SE 6).* http://docs.oracle.com/javase/6/docs/api/java/util/Properties.html.

[javc] *PropertyChangeSupport (Java Platform SE 6).* http://docs.oracle.com/javase/6/docs/api/beans/PropertyChangeSupport.html.

[log] *Apache log4j 1.2*. http://logging.apache.org/log4j/1.2/.

[mys] *MySQL :: The world's most popular open source database*. http://www.mysql.com/.

[Nei76] NEISSER, Ulric: *Cognition and reality : principles and implications of cognitive psychology*. 1976

[ope] *OpenStreetMap*. `http://www.openstreetmap.org/`.

[sql] *SQL-Definition*. `http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt`.

[swi] *Swingx-ws — Java.net*. `http://java.net/projects/swingx-ws`.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 16.Oktober 2012 Franz Kary