Hochschule
Kaiserslautern
University of
Applied Sciences

Informatik und
Mikrosystemtechnik
Zweibrücken

Studiengang

Angewandte Informatik

PO Version 2011

Bachelorarbeit

MapViewer - eine Softwarekomponente zur Visualisierung statistischer und georeferenzierter Daten

MapViewer - a software
component for visualizing statistical and georeferenced data

vorgelegt von

Malte Schledjewski

10. August 2015

Betreuung:	Prof. Dr. Manfred Brill
Zweitkorrektur:	Prof. Dr. Jörg Hettel

Zusammenfassung

Für das am Fraunhofer-Institut für Techno- und Wirtschaftsmathematik ITWM entwickelte Softwarepaket *Virtual Measurement Campaign (VMC)* soll das Modul *MapViewer* neu entwickelt werden. In einem Praxisprojekt wurde bereits ein Konzept hierfür erstellt. Ziel der Bachelorarbeit ist es, die Umsetzung dieses Konzepts darzustellen. Dabei werden unter anderem die verfeinerte Architektur erläutert und die Integration des *GeoServers* beschrieben. Für die Modellierung des *MapViewers* wird eine UML State Machine genutzt. In der Arbeit wird daher auf deren Auswirkungen auf Erweiterbarkeit und zukünftige Änderungen eingegangen. Dabei werden auch die Erfahrungen aus der iterativen Entwicklung berücksichtigt. Ebenfalls werden neue Techniken zur Verifikation, wie zum Beispiel statische Analyse und GUI-Automatisierung, erprobt und auf ihre mögliche Integration in den *VMC*-Entwicklungsprozess hin analysiert. In einem abschließenden Fazit werden darüber hinaus noch zusätzliche Optionen für den weiteren Entwicklungsprozess genannt.

Abstract

The *MapViewer* module of the *VMC* software package, which is developed at the Fraunhofer-Institute for Industrial Mathematics ITWM, is to be reengineered. A new concept has already been developed during an internship. The implementation thereof forms the major part of this thesis. Among others the refined architecture and the integration of the *GeoServer* are described. An UML State Machine is used for modelling the new *MapViewer*. The fitness of this approach and future reactions to changes are examined, taking into account the experiences from the incremental development. Also new techniques for verification, for example static analysis and GUI automation, are tested and analyzed in regard to their possible integration into the *VMC* developement process. In addition to the conclusion several options for further development are presented.

Ehrenwörtliche Erklärung

Hiermit erkläre ich, **Malte Schledjewski**, geboren am **21.12.1991 in Kaiserslautern**, ehrenwörtlich, dass ich meine Bachelorarbeit mit dem Titel:
„MapViewer - eine Softwarekomponente zur Visualisierung statistischer und georeferenzierter Daten“
selbstständig und ohne fremde Hilfe angefertigt habe und keine anderen als in der Abhandlung angegebenen Hilfen benutzt habe.

Die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen habe ich innerhalb der Arbeit gekennzeichnet.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

Ort/Datum (Unterschrift Studierender)

Disclaimer

Alle innerhalb des Berichts genannten und ggf. durch Dritte geschützten Marken- und Warenzeichen unterliegen uneingeschränkt den Bestimmungen des jeweils gültigen Kennzeichenrechts und den Besitzrechten der jeweiligen eingetragenen Eigentümer. Allein aufgrund der bloßen Nennung ist nicht der Schluss zu ziehen, dass Markenzeichen nicht durch Rechte Dritter geschützt sind.

VMC® ist eine Marke von Fraunhofer-Gesellschaft zur Förderung der angewandten Forschung e.V..

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Überblick . . . 1
- 1.2 Ausgangssituation . . . 2
  - 1.2.1 Aktueller MapViewer und VMC-Kontext . . . 2
  - 1.2.2 Neues Konzept aus dem Praxisprojekt . . . 6

2 Architektur **11**
- 2.1 Systemkomponenten . . . 11
- 2.2 Leitideen und Architekturmuster . . . 11
- 2.3 Entwickelte Architektur . . . 13
  - 2.3.1 Architektur auf C++-Seite . . . 13
  - 2.3.2 Browser-Client . . . 15

3 GeoServer **17**
- 3.1 Web Map Service . . . 17
- 3.2 Integration mit dem MapViewer . . . 17
- 3.3 Integration in den Installationsvorgang . . . 18
  - 3.3.1 Analyse der Installationsarten . . . 18
  - 3.3.2 Analyse der Installationsorte . . . 18
  - 3.3.3 Wahl von Installationsort und -art . . . 19

4 UML State Machine des MapViewers **21**
- 4.1 Eignung der Modellierung als UML State Machine . . . 21
  - 4.1.1 Endliche Automaten und endliche Maschinen . . . 21
  - 4.1.2 Harel Statecharts und UML State Machines . . . 21
  - 4.1.3 Eignung für GUI-Entwicklung . . . 22
- 4.2 Entwurf der UML State Machines . . . 23
  - 4.2.1 Steuerung des MapViewers . . . 23
  - 4.2.2 Steuerung der Karte . . . 25
- 4.3 Wahl der technischen Umsetzung auf C++-Seite . . . 26
  - 4.3.1 Kriterien und Auswahl der Optionen . . . 26
  - 4.3.2 Analyse der Optionen . . . 26
  - 4.3.3 Wahl der Option . . . 28

4.4 Wahl der technischen Umsetzung auf JavaScript-/TypeScript-Seite . . . 28
4.4.1 Kriterien und Auswahl der untersuchten Optionen . . . 28
4.4.2 Analyse der Optionen . . . 29
4.4.3 Wahl der Option . . . 29

5 Iterative Entwicklung **31**
5.1 Iteration 1 . . . 31
5.2 Iteration 2 . . . 33
5.3 Iteration 3 . . . 33
5.4 Iteration 4 . . . 34

6 Verifikation **37**
6.1 Bereits genutzte Techniken . . . 37
6.1.1 Unit-Tests mit Boost Test . . . 37
6.1.2 TestLink . . . 38
6.2 Neueingeführte Techniken . . . 40
6.2.1 Neue connect-Syntax von Qt . . . 40
6.2.2 TypeScript . . . 40
6.2.3 Statische Analyse mit Cppcheck . . . 40
6.2.4 GUI-Tests mit SikuliX . . . 40
6.3 Weitere Techniken . . . 42
6.3.1 Formale Methoden . . . 42
6.3.2 Möglichkeiten unter Linux . . . 42

7 Fazit **43**

Anhang **49**

Kapitel 1

Einleitung

1.1 Überblick

Am Fraunhofer-Institut für Techno- und Wirtschaftsmathematik ITWM wird das Softwarepaket *Virtual Measurement Campaign (VMC)* [53] entwickelt. Eingesetzt wird es „zur systematischen Analyse der Nutzungsvariabilität von Fahrzeugen auf Basis georeferenzierter Daten“ ([26]). *VMC* dient zum Beispiel der Berechnung von Geschwindigkeitsprofilen sowie der Simulation und Planung von Messfahrten (vgl. [25]). Einen genaueren Überblick geben [1, 53].

Ein Bestandteil des *VMC*-Clients ist das Modul *MapViewer*. Dies dient vorwiegend der Anzeige einer Karte und den damit verbundenen Interaktionen. In der Vergangenheit war der *MapViewer* oft Grund für Programmabstürze. Zudem beeinträchtigten einige subtile Bugs die Nutzerfreundlichkeit, zum Beispiel musste in manchen Fällen erneut auf die Karte geklickt werden, um den Kartenausschnitt vollständig zu aktualisieren. Auch war die Speicherverwaltung nicht korrekt umgesetzt, sodass manche Objekte geleakt wurden. Ebenfalls wurden viele (>10) Datenbank-Verbindungen pro angezeigter Karte verwendet.

Daher wurde beschlosse den *MapViewer* zu überarbeiten. Dabei war nicht nur eine Veränderung der Architektur, sondern auch eine Überarbeitung der grafischen Oberfläche gewünscht. Dies sollte ohne Refactoring durchgeführt werden, da der wiederverwendbare Teil des Quelltexts als sehr gering eingeschätzt wurde. Der neue *MapViewer* wird daher von Grund auf als neues Modul entwickelt.

In einem Praxisprojekt [47] wurde dazu ein neues Konzept für den *MapViewer* erarbeitet. Dabei gab es drei Vorgaben:

1. Alle Funktionen des alten *MapViewers* sollen erhalten bleiben.
2. Die Anzeige der Karte soll leicht austauschbar sein, damit auf eventuelle Kundenwünsche, wie zum Beispiel eine 3D-Ansicht, schnell reagiert werden kann.
3. Neue Abhängigkeiten in Form von benutzten Bibliotheken sollen nach Möglichkeit vermieden werden.

Es wurde zusätzlich entschieden, die eigene Implementierung der Kartenanzeige durch eine Anzeige basierend auf einem *GeoServer* zu ersetzen.

Beim Erstellen des neuen Konzepts fiel auf, dass der *MapViewer* an sich aus vielen Zuständen besteht. Aus diesem Grund wurde in Rücksprache mit dem Entwicklerteam beschlossen, die Umsetzung mittels einer UML State Machine zu modellieren.

Inhalt dieser Arbeit ist es, die UML State Machine zu entwerfen und den neuen *MapViewer* zu entwickeln. Zusätzlich bietet diese Neuentwicklung die Möglichkeit neue Ansätze bei der Verifikation zu erkunden.

Abschnitt 1.2 gibt einen Überblick über den aktuellen *MapViewer* und das neue Konzept.
Kapitel 2 beschreibt die entwickelte Architektur des neuen *MapViewers*.
Kapitel 3 gibt einen kurzen Überblick über den genutzten *GeoServer* und dessen Integration in *VMC*.
Kapitel 4 stellt den theoretischen Hintergrund der UML State Machine sowie die entwickelten Maschinen dar und beinhaltet die Wahl der technischen Umsetzung.
Kapitel 5 schildert die iterative Implementierung des Konzepts in einem kleineren Umfang und die dabei gemachten Erfahrungen.
Kapitel 6 beschäftigt sich mit dem Testen und anderen Möglichkeiten zur Verifikation des neuen *MapViewers*.
Kapitel 7 zieht ein Fazit, ob sich die Modellierung als UML State Machine gelohnt hat und inwieweit die Techniken zur Verifikation nützlich waren. Auch wird ein Ausblick auf die weitere Entwicklung von *VMC* gegeben.

1.2 Ausgangssituation

1.2.1 Aktueller MapViewer und VMC-Kontext

Der aktuelle *MapViewer* besitzt zwei Varianten. Zum einen kann das Modul eine Kartenansicht als Widget liefern, welche dann vom *DataSelection* Modul benutzt wird. Dabei dient die Karte der Selektion von Verwaltungsbereichen (Staaten, Bundesländern, Landkreisen etc.), um eine Region zu erstellen. Die in die Oberfläche aus *DataSelection* eingebettete Karte ist in Abbildung 1 zu sehen. Das *DataSelection* Modul bietet auch eine Möglichkeit die Verwaltungsbereiche über eine Baumstruktur zu wählen, wie in Abbildung 2 gezeigt ist. Dabei ist jede Ebene nach den Anfangsbuchstaben sortiert und Untereinträge stellen Teilbereiche dar. Die beiden Arten, eine Region zu erstellen, lassen sich jedoch nicht kombinieren.

Zum anderen bietet das Modul *MapViewer* die Option ein Fenster zu öffnen, welches eine Karte mit einigen Interaktionsmöglichkeiten bietet. Dieses Fenster wird *MapViewer* genannt. Der *MapViewer* dient in dieser Form dem Erstellen von Routen mittels Routing durch Eingabe oder Laden von Routingpunkten, wie aus Abbildung 3 ersichtlich wird. Neben der Liste der Routingpunkte werden dabei auch Optionen für das Routing angezeigt.

Erstellte Routen und Regionen können, wie in Abbildung 4 zu sehen ist, farbcodiert dargestellt werden, wobei alle berechneten Faktoren zur Verfügung stehen.

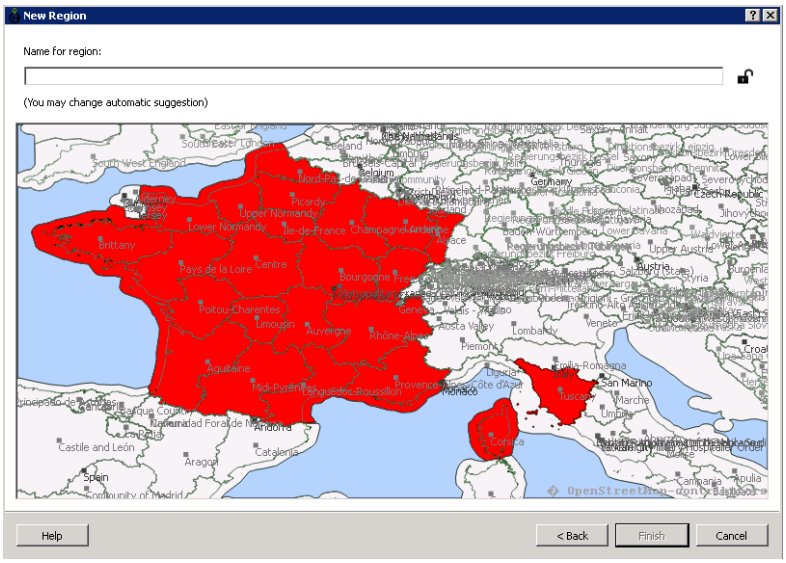


Abbildung 1: Kartenbasierte Regionsauswahl (aus [47])

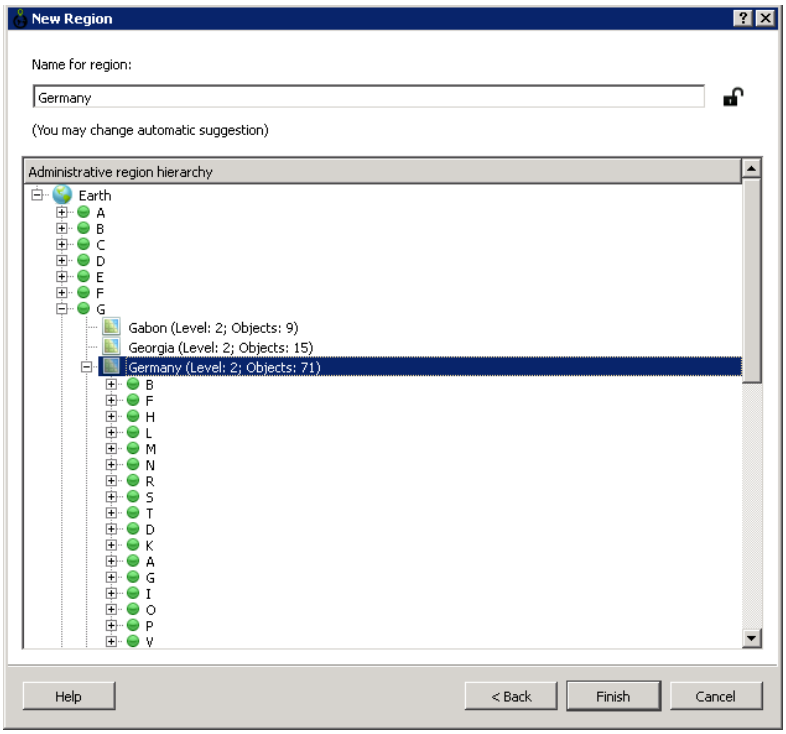


Abbildung 2: Regionsauswahl über Baumansicht (aus [47])

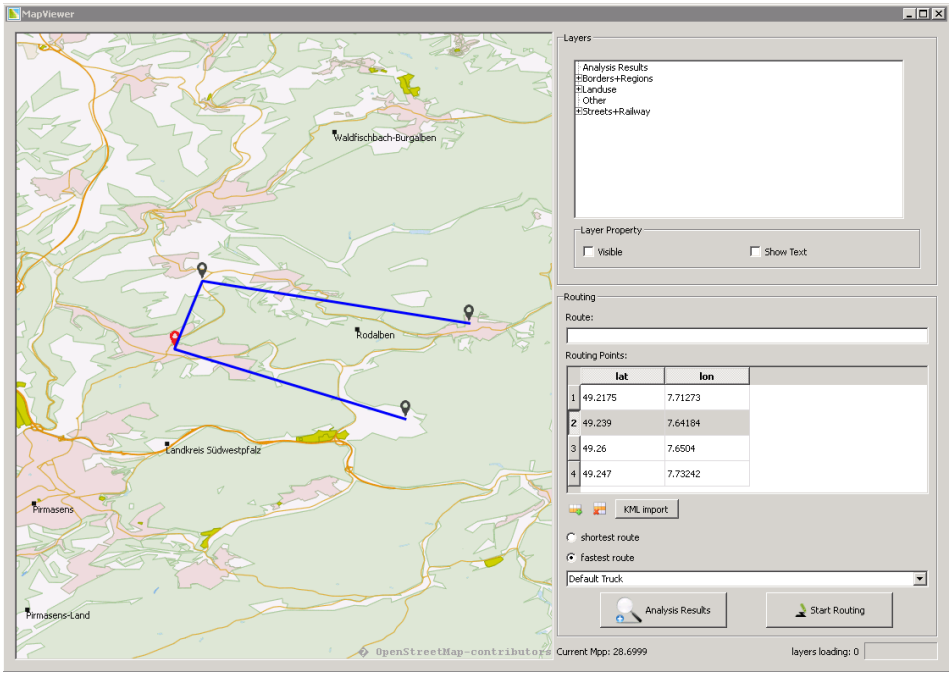


Abbildung 3: Die aktuelle Version des *MapViewers* während des Routings (aus [47])

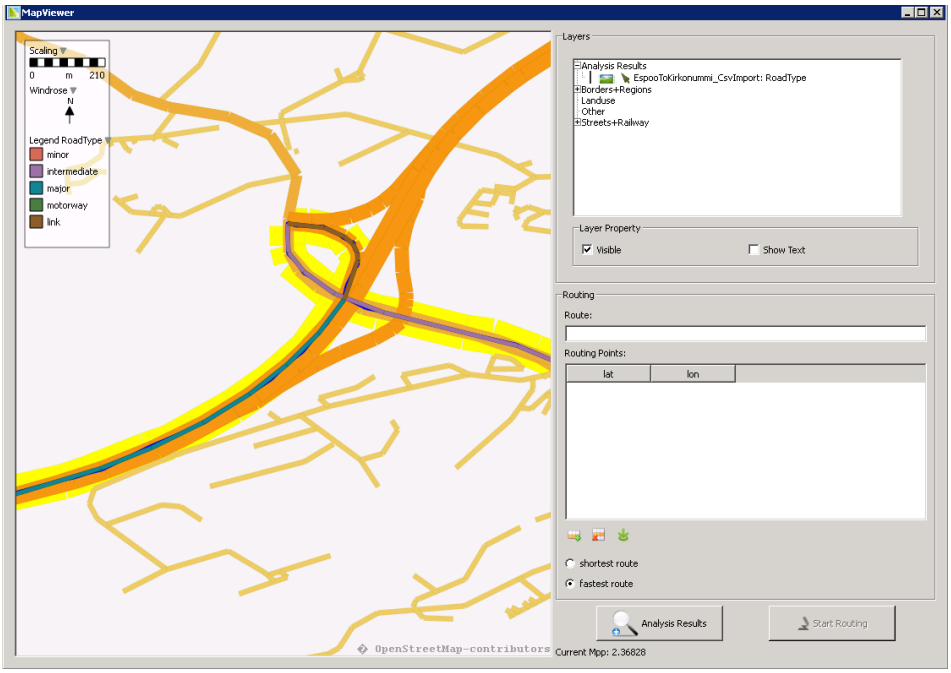


Abbildung 4: Farbcodierte Route in der aktuellen Version des *MapViewers* (aus [47])

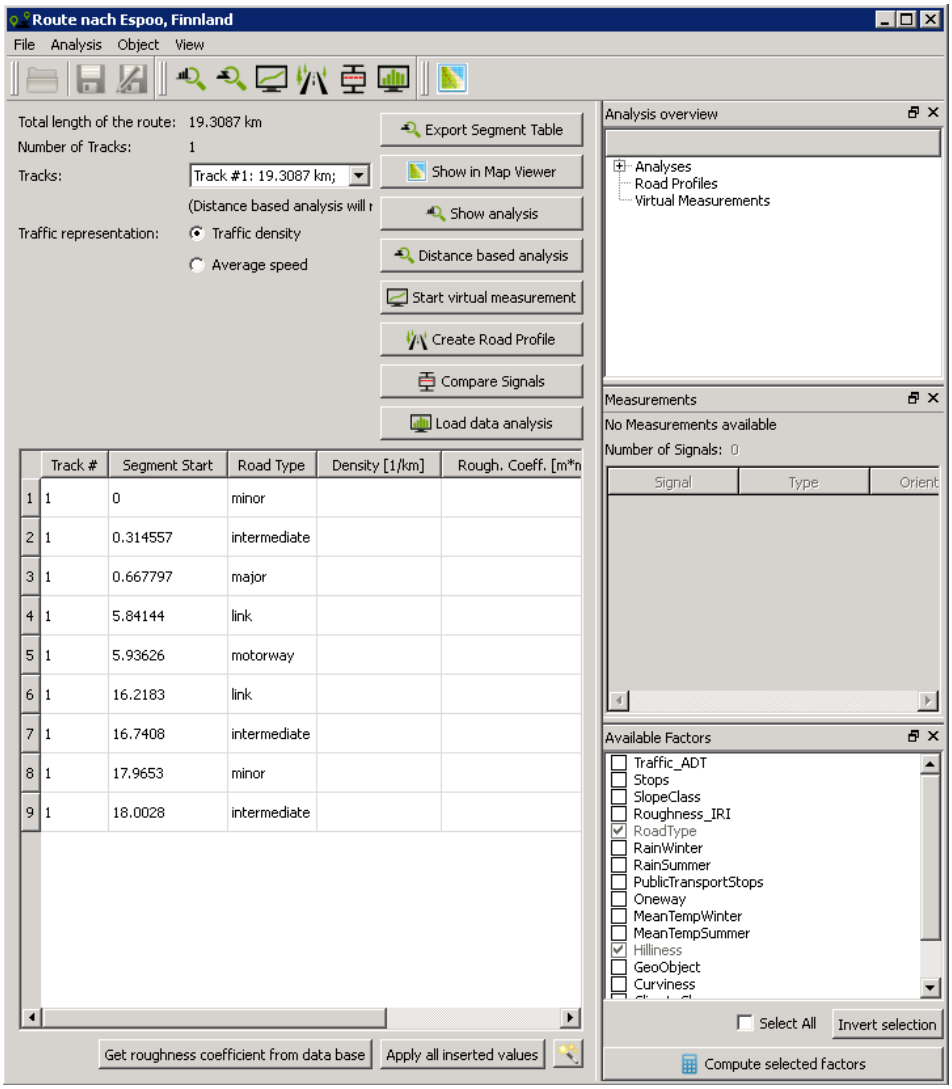


Abbildung 5: Ein *RouteViewer* (aus [47])

Abbildung 5 zeigt einen *RouteViewer*, der für jede Route geöffnet werden kann. Er ist ebenfalls in einem anderen Modul. Der *RouteViewer* bietet alle möglichen Interaktionen mit der Route. Er besitzt jedoch keine integrierte Kartenansicht.

Der *MapViewer* kann auf drei Arten als Fenster geöffnet werden:

- Durch Klicken auf das *MapViewer*-Icon im *VMC*-Hauptfenster.
- Durch Klicken auf das *MapViewer*-Icon im *RouteViewer* einer Route, wobei auf die Route gezoomt wird und sie einfarbig auf der Karte angezeigt wird.
- Durch Klicken auf „Route planning“ im *VMC*-Hauptfenster.

Das Routing ist nur möglich beim Starten durch die letztgenannte Art, das Anzeigen von Routen und Regionen jedoch immer.

1.2.2 Neues Konzept aus dem Praxisprojekt

Im Praxisprojekt wurde ein neues Konzept für den *MapViewer* entwickelt (vgl. [39, 40, 47]). Es bildet das langfristige Ziel und wird zunächst schrittweise in kleinerem Umfang umgesetzt.

Dazu wurden drei Use Cases entwickelt, die den *MapViewer* direkt betreffen. Die einfache Ansicht der Karte ist durch den Use Case „Visualisierung von Messdaten“ in Abbildung 7 dargestellt. Der Anwender kann wie gewohnt den Kartenausschnitt verändern und sich Routen und Regionen farbcodiert darstellen lassen. Dabei sind zum einen Lesezeichen für Kartenausschnitte hinzugekommen, um schnell zu häufig benutzten Ausschnitten zu gelangen,. Zum anderen gibt es die Möglichkeit einen so genannten *SplitView* zu starten. Dabei wird eine Route oder Region einzeln auf einer zweiten Karte gezeigt. Zusätzlich gibt es eine Export-Funktion für den aktuellen Kartenausschnitt.

Der Use Case in Abbildung 6 repräsentiert das Anlegen einer Region. Die Auswahl auf der Karte und die Auswahl mittels Baumansicht werden kombiniert und um eine Suche mittels Namen erweitert.

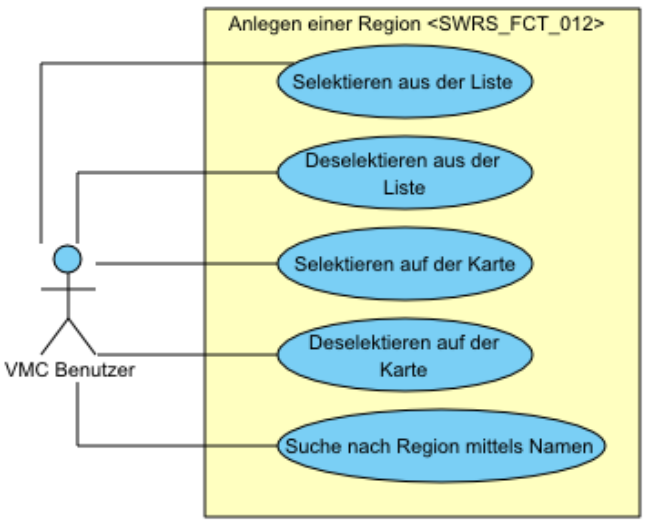


Abbildung 6: Use Case „Anlegen einer Region“ (aus [39])

Der Use Case zum Anlegen einer Route hat sich kaum geändert. Die einzige Ergänzung ist die Möglichkeit, für einen Routingpunkt festzulegen, wo er auf das Straßennetz projiziert wird.

Bisher hat jede Route und Region einen eigenen *Route-/RegionViewer*. Der *RegionViewer* besitzt keine Kartendarstellung. Der *RouteViewer* hingegen öffnet einen passenden *MapViewer* als neues Fenster. Das Entwicklerteam findet dieses System bereits mit wenigen Routen und den damit verbundenen Fenstern für den Anwender zu unübersichtlich, da die Fenster sich gegenseitig überlagern und die zueinander gehörenden Fenster sich nur schwer zuordnen lassen.

Der neue *MapViewer* integriert daher die bisherigen *Route-/RegionViewer*. Es ist nun nicht mehr nötig mehrere Fenster zu öffnen. Abbildung 8 zeigt einen Wireframe des neuen *MapViewers*.

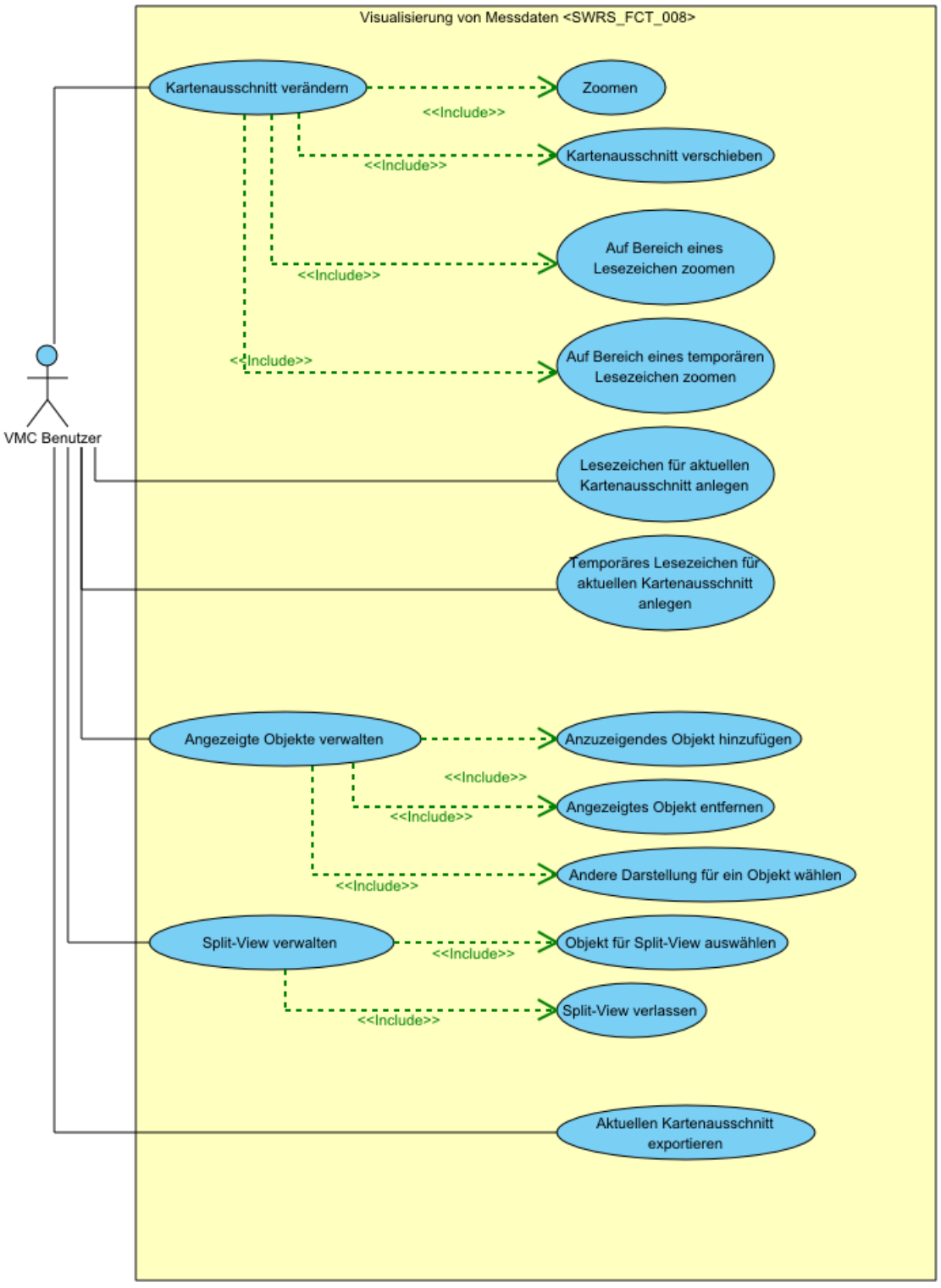


Abbildung 7: Use Case „Visualisierung von Messdaten“ (aus [39])

Dabei besteht die neue GUI aus drei Teilen:

1. Auf der linken Seite gibt es eine Übersicht über alle Routen und Regionen mit ihren verfügbaren Faktoren. Hier lässt sich einstellen, welche Objekte angezeigt werden.
2. In der Mitte befindet sich die eigentliche Karte.
3. Auf der rechten Seite existiert ein kontextspezifischer Bereich.

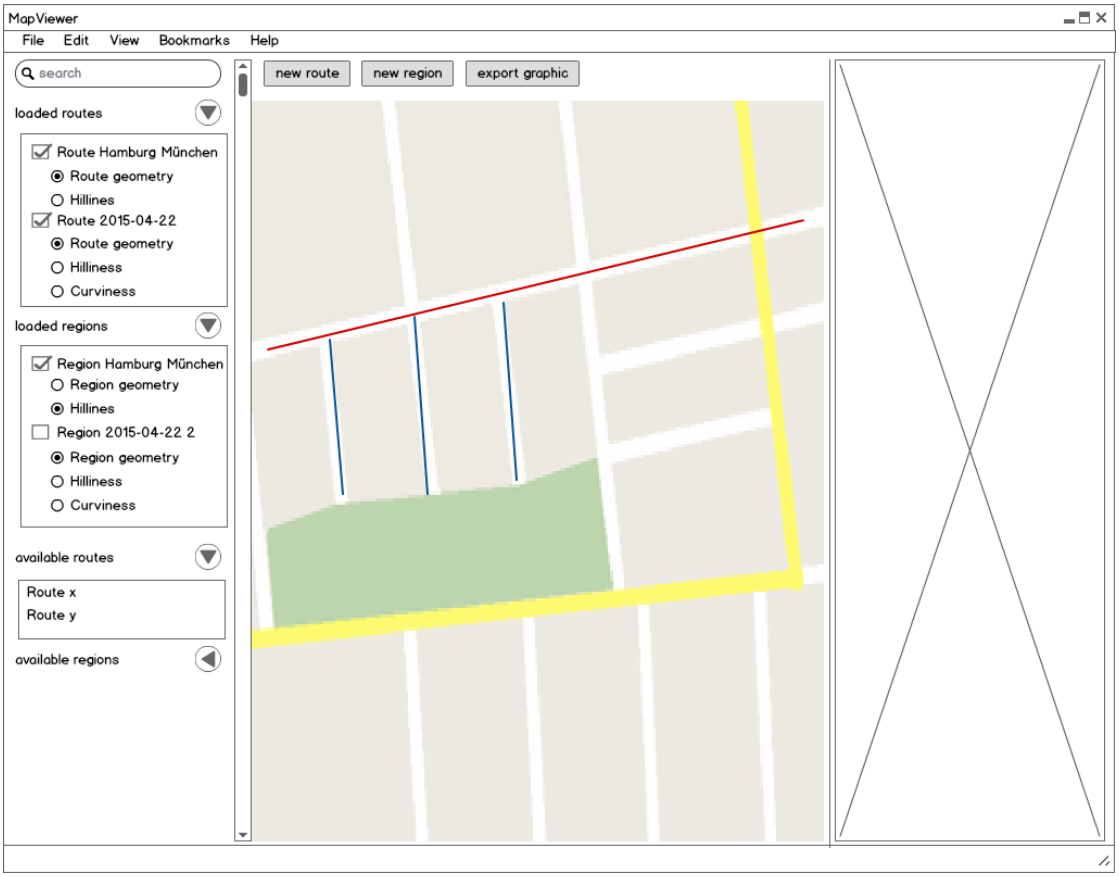


Abbildung 8: Wireframe des neuen *MapViewers* (aus [47])

Dieser Bereich auf der rechten Seite sieht je nach gewünschter Aktion anders aus. So kann er zum Beispiel dem bisherigen Widget zum Routen oder dem *RouteViewer* ähneln. Auch werden hier andere Use Cases, die zum Beispiel bisher durch den *RouteViewer* abgedeckt wurden, unterstützt.

Da die unterschiedlichen Aktionen nicht nur andere angezeigte Widgets, sondern auch unterschiedliche Interaktionen mit der eigentlichen Karte bewirken, wurde der Vorschlag, den neuen *MapViewer* durch eine UML State Machine zu steuern, vom Entwicklerteam angenommen. Somit kann zum Beispiel den unterschiedlichen Aktionen jeweils ein Zustand zugeordnet werden. Wie einfach diese Zuordnung ist, ist in Unterabschnitt 4.2.1 beschrieben.

Um auf mögliche Kundenwünsche reagieren zu können soll die Karte leicht austauschbar sein. Es wird hier nur eine Version implementiert. Das aktuelle System zur Anzeige wird durch einen *Web Map Service (WMS)* ersetzt, welcher die gerenderte Karte als Bild liefert. Als *Web Map Service* kommt *GeoServer* zum Einsatz. Dies wird in Kapitel 3 genauer beschrieben. In einem integrierten Browser dient *OpenLayers* als *WMS*-Client zur Anzeige der Karte (vgl. Abschnitt 2.2).

Kapitel 2

Architektur

2.1 Systemkomponenten

Bisher besteht *VMC* aus einer zentralen Datenbank und einem Client. In der Datenbank sind, neben dem Straßennetz mit seinen zugehörigen Daten, auch die meisten Nutzerdaten hinterlegt. Der Client ist eine in C++ realisierte 64bit Windows-Anwendung. Dabei werden unter anderem *Qt* 5.5 und einige *Boost*-Bibliotheken in Version 1.57 benutzt. Der Client übernimmt den Großteil der Berechnungen und führt sie lokal aus. Die dazu benötigten Daten werden jedes Mal aus der Datenbank ausgelesen und die Ergebnisse dort persistiert.

Die bestehende Konstellation wird nun um einen *Web Map Service (WMS)* Server erweitert (vgl. Kapitel 3), der die Karte rendert. Dieser *WMS* kommuniziert mit der Datenbank, um die benötigten Geometrien zu erhalten. Die neue Version des Moduls *MapViewer* wird hauptsächlich mit diesem *WMS*-Server kommunizieren. Nur in einigen wenigen Fällen ist eine Kommunikation mit der Datenbank notwendig.

2.2 Leitideen und Architekturmuster

Die Entwicklung der Architektur orientiert sich an folgenden Ideen und Architekturmustern:

Verwendung von fertiger Software Das Zeichnen der Karte ist keine Kernkompetenz von *VMC*. Der Einsatz von bereits existierender Software erlaubt somit, Entwicklungsressourcen zu sparen.
So ersetzt der *WMS* die eigene Implementierung der Kartenanzeige. Bisher benutzt der *MapViewer* direkt einen `QPainter` aus *Qt*, um die Karte lokal zu zeichnen. Dabei wird bei jeder Änderung der Karte die noch benötigte Geometrie aus der Datenbank geladen und gezeichnet. Die nicht mehr genutzte Geometrie wird dann gelöscht. Es gibt keinen Cache. Auch muss darauf geachtet werden, dass das Nachladen der Geometrie den GUI-Thread nicht blockiert.
Damit kein *WMS*-Client implementiert werden muss, wurde bereits während des Praxisprojekts beschlossen, dass *OpenLayers* 3 [37] genutzt wird (vgl. [47]). Dabei handelt es sich

um eine JavaScript-Bibliothek. Diese wird in einem integrierten Browser angezeigt. Die Kommunikation zwischen dem JavaScript-Code und dem in C++ geschriebenen *VMC* ermöglicht das *Qt WebChannel* System [42].
Das spekulative Laden von höheren Zoomstufen durch *OpenLayers* 3 hat zusammen mit dem Einsatz des Caches in ersten Versuchen im Vergleich zur Nutzung des aktuellen *MapViewers* eine deutlich flüssigere Kartendarstellung ermöglicht.

Model-View-Controller in VMC Da es viele unterschiedliche Definitionen des Model-View-Controller (MVC) Patterns gibt (zum Beispiel in [52, Seite 5ff.]), wird zunächst die in *VMC* benutzte Variante definiert. Sie beschreibt gleichzeitig den groben Aufbau des Moduls.
Das Model repräsentiert die Daten und kapselt die grundlegenden Operationen. Der View stellt die Daten dar. Es gibt keine direkte Abhängigkeit zwischen Model und View. Die Kommunikation erfolgt über den Controller. Dieser reagiert auf Ereignisse und steuert Model und View. In jedem VMC-Modul gibt es einen speziellen Modul-Controller, welcher die Kommunikation mit den anderen Modulen übernimmt.

Model-View Architektur von Qt In *Qt* kommt eine Model-View-Architektur zum Einsatz. Dabei wird der Controller in den View kombiniert. Zusätzlich werden jedoch Delegates eingeführt. Diese erlauben die Darstellung der Daten anzupassen und andere Bearbeitungsmodi zu verwenden. Einen genaueren Überblick gibt *Qt Documentation - Model/View Programming* [41].

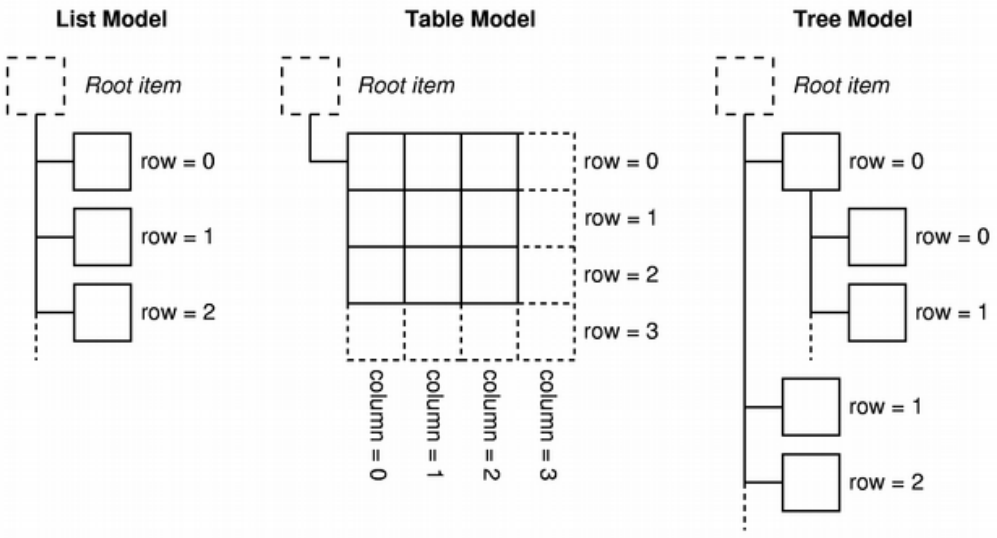


Abbildung 9: `QAbstractItemModel` in drei Varianten (aus [41])

Das Model wird durch die abstrakte Klasse `QAbstractItemModel` beschrieben. Dieses Model ist hierarchisch. Jede Ebene besteht aus Zeilen und Spalten und jede Zelle kann wiederum eine Unterebene besitzen. Einige Beispiele für typische Varianten zeigt Abbildung 9. Änderungen am Model werden den Views durch den *Qt*-Signal-Slot-Mechanismus mitgeteilt.

UML State Machines Eine UML State Machine steuert den *MapViewer*. Die entwickelte Maschine für die C++-Seite sowie die Maschine, die im integrierten Browser läuft, sind in Abschnitt 4.2 beschrieben.

Single Responsibility Principle Der neue *MapViewer* orientiert sich am Single Responsibility Principle. Demnach soll es niemals mehr als einen Grund pro Klasse geben, diese zu ändern [29]. In Anlehnung an dieses Prinzip wurde versucht, für jede Klasse genau eine klare Aufgabe zu definieren.

2.3 Entwickelte Architektur

Der *MapViewer* ist an sich ein reines C++-Modul. Die implementierte Kartendarstellung benutzt jedoch *OpenLayers* 3 in einem integrierten Browser als *WMS*-Client. Dieser Teil, der im Browser läuft, wird im Folgenden als Browser-Client bezeichnet.

Der integrierte Browser wird unter anderem durch die Klassen `QWebView` und `QWebEngine` realisiert. Die Kommunikation zwischen dem in C++ geschriebenen *MapViewer* und dem im integrierten Browser laufenden JavaScript ermöglicht ein `QWebChannel` [42].

Zunächst wird die Architektur auf C++-Seite vorgestellt. Danach wird die Entwicklung und grobe Struktur des Browser-Clients beschrieben.

Die technische Umsetzung der jeweiligen UML State Machines wird in Abschnitt 4.3 und Abschnitt 4.4 festgelegt.

2.3.1 Architektur auf C++-Seite

Abbildung 10 beinhaltet das Klassendiagramm mit der Einteilung in Model/View/Controller. Dabei leiten die grün markierten Klassen von `QObject` ab und können somit den Signal-Slot-Mechanismus von *Qt* benutzen. Rot markierte Klassen wurden noch nicht implementiert, alle anderen zumindest teilweise.

Der Modul-Controller im *MapViewer*-Modul ist der `CMapViewerControl`. Er koordiniert die Kommunikation mit den anderen Modulen und verwaltet alle *MapViewer*-Fenster.

Ein *MapViewer*-Fenster wird durch einen `CMapViewer` verwaltet. Dieser wird durch eine UML State Machine gesteuert. Die Klassen zur Umsetzung der UML State Machine sind in diesem Diagramm nicht enthalten, da die technische Umsetzung erst in Kapitel 4 entschieden wird.

`CBookmarksManager` und `CConfigManager` verwalten die Lesezeichen für Kartenausschnitte und die allgemeine Konfiguration und gehören somit zum Model.

Die eigentlichen Routen und Regionen werden in einem anderen Modul verwaltet. Es wird jeweils eine Proxy-Klasse eingeführt. Bisher sind alle Routen und Regionen einer Session geladen. Dies kann sich in Zukunft ändern, da mittlerweile Sessions mit mehr als 1000 Routen und Regionen existieren. Zum einen kommt also ein Proxy-Pattern [52] zum Einsatz, um das gewünschte Objekt erst bei Bedarf zu erstellen, und zum anderen wird hier vom Interface der modulexternen Klassen entkoppelt. Es handelt sich also auch um eine Art Brücken-Pattern.

Die beiden Proxy-Klassen `CRegionProxy` und `CRouteProxy` werden dann von einem sitzungsweitem `CSessionLayerModel` verwaltet.

Jeder offene *MapViewer* erzeugt ein `CLayerModel` basierend auf dem `CSessionLayerModel`. Es implementiert ein `QAbstractItemModel`, um später von der in *Qt* implementierten GUI benutzt

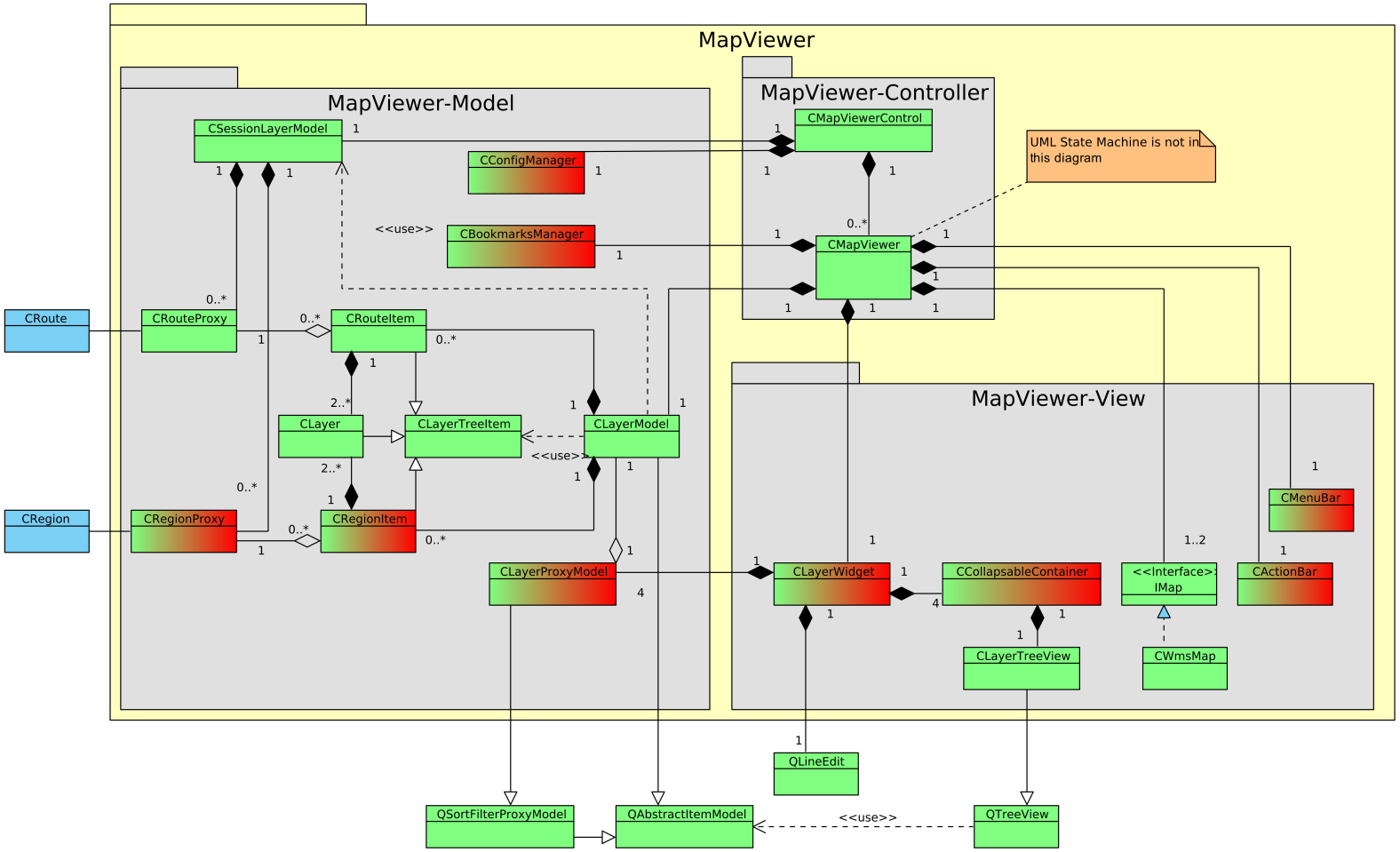


Abbildung 10: Klassendiagramm des neuen *MapViewer*-Moduls

zu werden. Jedes `CRouteProxy` Objekt wird durch ein `CRouteItem` Objekt repräsentiert. Analog wird mit dem `CRegionProxy` verfahren.

`CLayer` repräsentiert eine anzeigbare Ebene und entspricht somit zum Beispiel einem berechneten Faktor einer Route.

`CLayer`, `CRegionItem` und `CRouteItem` bilden also eine Baumstruktur, die durch das `CLayerModel` dargestellt wird.

Da das `QAbstractItemModel` mit Pointern arbeitet, bekommen alle drei Klassen der Baumstruktur eine gemeinsame Oberklasse `CLayerTreeItem`, die lediglich einen Aufzählungstypen besitzt, der die Klassen unterscheidet.

Die Liste der Routen und Regionen inklusive ihrer Ebenen kann in der GUI gefiltert werden. Daher implementiert `CLayerProxyModel` ein `QSortFilterProxyModel`, welches dem `CLayerModel` vorangestellt wird, wie es in *Qt* üblich ist.

Mehrere `CLayerProxyModel` Objekte gehören dann einem `CLayerWidget`, das die linke Seite der GUI repräsentiert. Es besitzt mehrere `CCollapsableContainer`, die jeweils eine aufklappbare Sektion darstellen. In jeder Sektion gibt es dann einen `CLayerTreeView`, welcher eines der `CLayerProxyModel` Objekte in der GUI repräsentiert. Dabei erbt `CLayerTreeView` von `QTreeView` und ändert nur ein paar kleine Interaktionen, wie zum Beispiel das Verhalten bei einem Doppelklick.

Eine `CMenuBar` repräsentiert das Menü eines *MapViewers*. Über der Karte wird eine `CActionBar` angezeigt, die mehrere Aktionen, wie zum Beispiel das Anlegen einer Route, ermöglicht.

Da die Anzeige der Karte austauschbar sein soll, wird das Interface `IMap` eingeführt. Die Erstellung einer Instanz, die `IMap` implementiert, wird durch eine Fabrik-Methode vorgenommen. Dabei wird die passende Implementierung durch einen Aufzählungstypen bestimmt. Dies entspricht ungefähr dem Pattern der abstrakten Fabrik [52].

In dieser Bachelorarbeit wird eine Implementierung von `IMap` in Form von `CWmsMap` erarbeitet. Die von `CWmsMap` genutzten Klassen sind jedoch in diesem Diagramm nicht aufgelistet, um es übersichtlich zu halten.

2.3.2 Browser-Client

JavaScript ist nicht klassenbasiert, sondern verfügt über ein Prototypen-System [6]. Auch ist `this` nicht lexikalisch gebunden, sondern wird dynamisch gesetzt [49]. Beides ist für C++-Entwickler ungewohnt.

Dies ist einer der Gründe warum anstatt in JavaScript in TypeScript [50] entwickelt wird. TypeScript ist eine Übermenge von JavaScript. JavaScript-Code ist also automatisch gültiger TypeScript-Code. TypeScript-Code wird in JavaScript-Code kompiliert, welcher dann wie gewohnt im Browser läuft. Klassen und lexikalisches Binden sind in TypeScript möglich. Einige der zusätzlichen Sprach-Features gibt es auch mit dem neuen ECMAScript 6, welches jedoch erst während der Entwicklung offiziell erschien. Auch wird es noch nicht von den Browsern unterstützt [11].

Die wichtigste Änderung ist jedoch ein Typsystem. Dieses ermöglicht das Erstellen von Typen und das Annotieren von Variablen mit Typinformationen. So kann der Compiler zum Beispiel darauf hinweisen, dass anstelle eines `int` ein `string` übergeben wurde. Auch können anhand des bestimmten Typs die Vervollständigungsvorschläge gefiltert werden, womit die Entwicklung erleichtert wird. Zur Entwicklung wird TypeScript 1.5 benutzt.

Die Architektur besteht aus zwei Teilen. Die eigentliche Karte, die durch *OpenLayers* 3 dargestellt wird, wird durch eine Klasse gekapselt. Daneben gibt es die UML State Machine, die die Karte steuert. Beide Teile werden mittels *QWebChannel.js* an den C++-Teil angeschlossen.

Kapitel 3

GeoServer

3.1 Web Map Service

Web Map Service (WMS) [33] ist ein Standard des Open Geospatial Consortiums (OGC). Ein *WMS* liefert eine Karte codiert als Bildformat. Die Anfrage an den *WMS* erfolgt mittels eines URL-Aufrufs. Der *WMS* bietet dazu unterschiedliche Styles an. Einzelne Layer können dabei einen transparenten Hintergrund haben. Mit der optionalen Erweiterung *Styled Layer Descriptor (SLD)* [35] ist es möglich, selbst definierte Styles bei der Anfrage mitzugeben.

Neben *WMS* gibt es den *Web Map Tile Service Standard (WMTS)* [34]. Dabei wird nicht für jeden frei wählbaren, angefragten Kartenausschnitt ein Bild generiert, sondern es werden Ebenen aus vordefinierten Tiles bereitgestellt. Ein *WMS*-Server kann intern auch Tiles benutzen. Aus diesem Grund gibt es eine inoffizielle Erweiterung [38] durch OSGeo. Diese legt fest, wie der *WMS*-Server dem Client mitteilen kann, welche Tiles verfügbar sind.

Die Entscheidung für den Einsatz eines *WMS* fiel während des Praxisprojekts und ist in [40] erläutert. Einen Überblick über weitere Rendermöglichkeiten gibt [44].

3.2 Integration mit dem MapViewer

Der *GeoServer* läuft als eigenständiger Server. Daher steht die *GNU General Public License (v2)* [16] des *GeoServers* nicht im Widerspruch zur *VMC*-Lizenz. Der *MapViewer* benutzt *OpenLayers* 3, um mit dem *WMS*-Server zu kommunizieren. Dabei wird gegebenenfalls ein angepasster Style in der Anfrage mit übertragen.

Nach der Installation des *GeoServers* übernimmt der *MapViewer* die weitere Konfiguration. Dabei wird die REST-API des *GeoServers* benutzt. Zunächst muss unter anderem ein Workspace mit einer zugehörigen Datenquelle eingerichtet werden. Neben den grundlegenden Layern, wie zum Beispiel den unterschiedlichen Straßenarten, wird auch für jede Route und Region jeweils ein eigener Layer angelegt.

3.3 Integration in den Installationsvorgang

Der *GeoServer* bietet die folgenden drei Installationsarten an:

- Installation mittels der plattformunabhängigen Variante
- Installation mittels Installer
- Installation mittels *Web Application Archive* (.war)

Auch ist zu klären, wo der *GeoServer* installiert wird. Dazu werden folgende Varianten betrachtet:

- Installation auf dem Datenbank-Server
- Installation auf dem Client
- Installation auf einem anderen Server

Beide Aspekte werden zunächst getrennt untersucht. Danach wird entschieden welche Kombination gewählt wird.

3.3.1 Analyse der Installationsarten

Laut Dokumentation des *GeoServers* wird unter Windows die Installation mittels Installer bevorzugt [15]. Sie ist nutzerorientiert und führt den Anwender in wenigen Schritten durch die Installation. Bei der plattformunabhängigen Variante müssen manuell Umgebungsvariablen gesetzt werden. Auch ist hierbei die Installation als Service nicht unterstützt. Der *GeoServer* muss dann jedes Mal manuell gestartet werden. Bei einem vorhandenen *Java Servlet Container* ist die Installation mittels *Web Application Archive* einfacher, da lediglich das *Web Application Archive* geladen werden muss.

3.3.2 Analyse der Installationsorte

Der Installationsort des *GeoServers* hat keine direkte Auswirkung auf den *MapViewer*. Latenzen aufgrund des Netzwerks sind im lokalen Netzwerk vernachlässigbar, da sie sehr klein sind. Eine Instanz lokal auf dem Client ist somit von einer Instanz auf einem lokalen Server kaum zu unterscheiden. Jedoch hat der Installationsort unter anderem Auswirkung auf den Installationsvorgang und den Speicherverbrauch.

Zunächst werden die untersuchten Kriterien ausgewählt. Die Geschwindigkeit des eigentlichen Renderns auf den unterschiedlichen Rechnern sollte sich kaum unterscheiden. Da die meisten Anfragen aus dem Cache beantwortet werden sollen, ist die Geschwindigkeit des Renderns vernachlässigbar.

In Rücksprache mit dem restlichen Entwicklerteam wurden folgende Kriterien gewählt:

- Java-Abhängigkeit
- Benötigte Portfreigabe
- Verwaltungsaufwand
- Effizienz der Speichernutzung
- Effizienz der Netzwerknutzung

Die ersten drei Kriterien hängen nicht vom Anwenderverhalten ab.

Java-Abhängigkeit Java bietet eine zusätzliche Angriffsfläche. Aufgrund der hohen Verbreitung ist Java ein beliebtes Ziel von Angriffen und die Anzahl an Exploits ist in den vergangenen Jahren stark gestiegen [28]. Daher ist es sinnvoll, die Java-Abhängigkeit auf möglichst wenige Computer zu beschränken. Wenn der GeoServer auf dem Client läuft, ist die Zahl der benötigten Java-Installationen deutlich höher. In den beiden anderen Optionen wird Java nur auf dem jeweiligen Server benötigt.

Benötigte Portfreigabe Bei beiden Server-Optionen muss der Port, unter dem der *GeoServer* erreichbar ist, für das lokale Netz freigegeben werden und ist somit eine potenzielle Gefahr. Bei der Installation auf dem Client muss der Port jeweils nur lokal erreichbar sein und nicht für das Netzwerk freigegeben werden.

Verwaltungsaufwand Bei den beiden Server-Optionen muss jeweils nur eine *GeoServer*-Installation verwaltet werden anstatt eine pro Client. Die Verwaltung umfasst zum einen die Konfiguration von Layern und sonstigen Objekten im *GeoServer*, und zum anderen das Updaten des *GeoServers* an sich.

Zur Untersuchung von Speichernutzung und Netzwerknutzung ist Wissen über das Anwenderverhalten nötig. Die Ergebnisse können sehr unterschiedlich ausfallen. Wenn die Anwender die gleichen Regionen betrachten, wird der Cache gemeinsam genutzt. Falls die Anwender unterschiedliche Regionen betrachten, werden bei einem kleinen Cache jeweils die bisherigen Tiles verdrängt. Daher werden die beiden Extreme betrachtet. Im einen Fall wird angenommen, dass alle Anwender die gleichen Regionen betrachten und im anderen, dass alle Anwender unterschiedliche Regionen betrachten.

Szenario 1 - gleiche Region Bei der Client-Option muss jede *GeoServer*-Instanz die betrachtete Region rendern. Der Cache der anderen kann nicht benutzt werden. Die Geometrie muss also an alle Clients übertragen werden. Auch wird auf jedem Client der Cache wachsen. Bei der Instanz auf dem Datenbank-Server muss die Geometrie nicht über das Netzwerk übertragen werden. Die Tile muss nur einmal gerendert werden und nachfolgende Anfragen können aus dem Cache bedient werden. Die gerenderten Tiles werden dann auch lokal im Client gecached, wodurch der Netzwerkverkehr weiter reduziert wird. Wenn der *GeoServer* auf einem anderen Server installiert wird, kommt lediglich der Netzwerkverkehr zur Datenbank für das einmalige Rendern hinzu.

Szenario 2 - unterschiedliche Regionen Bei der Client-Option kann es nicht dazu kommen, dass Anfragen zu Regionen eines Anwenders die bereits gerenderten Tiles einer anderen Region eines anderen Anwenders aus dem Cache verdrängen, da diese in unterschiedlichen Caches liegen. Falls bei den beiden Server-Optionen der Cache zu klein ist, müssen andauernd Tiles neu gerendert werden, da sie vorher aus dem Cache verdrängt wurden.

3.3.3 Wahl von Installationsort und -art

Damit der Cache gut genutzt werden kann, muss er groß sein. Ein Versuch mit drei einfachen Straßentypen und der Landmasse brauchte für die oberen Zoom-Ebenen etwa 5GB. Dies ist für die einzelnen Clients unangemessen. Daher wird eine zentrale Installation auf einem Server gewählt. Dabei fällt die Wahl auf den Datenbank-Server, da somit keine weitere Adresse in der Konfiguration eingetragen werden muss. Auch kann die Installation des GeoServers mit der Installation der Datenbank kombiniert werden. Der Verwaltungsaufwand ist gering, da nur eine

Installation gewartet werden muss. Die Portfreigabe für das Intranet und die Java-Abhängigkeit stellen nach Absprache auch kein Hindernis dar. Da der *GeoServer* durchgehend auf dem Server laufen und der Installationsvorgang einheitlich gehalten werden soll, ist die Installation als Service mittels Installer vorgesehen.

Kapitel 4

UML State Machine des MapViewers

4.1 Eignung der Modellierung als UML State Machine

4.1.1 Endliche Automaten und endliche Maschinen

Endliche Automaten sind ein bewährtes Modellierungsmittel und werden zum Beispiel im Zusammenhang mit formalen Sprachen angewendet. Ein endlicher Automat ist ein Tupel $M = (Q, \Sigma, \delta, q_0, E)$, bestehend aus einer endlichen Menge Q an Zuständen, einem Eingabealphabet Σ, einem designierten Startzustand q_0 aus der Menge der Zustände (also $q_0 \in Q$), einer endlichen, nicht leeren Menge E an akzeptierten Endzuständen und einer Überführungsfunktion δ, die einem aktuellen Zustand und einer Eingabe einen Nachfolgezustand zuordnet [55].

Endliche Automaten lassen sich auch mit Ausgabe kombinieren und werden dann als endliche Maschinen bezeichnet. Dabei werden meist zwei Varianten betrachtet: Bei den Moore-Maschinen ist die Ausgabe nur vom Zustand und bei den Mealy-Maschinen nur vom Übergang abhängig.

4.1.2 Harel Statecharts und UML State Machines

Harel zeigt in „Statecharts: a visual formalism for complex systems“ [20] eine erweiterte Modellierungsmöglichkeit und visuelle Darstellung für Maschinen. Zustände können nun in Hierarchien angeordnet werden. Dies erlaubt es, gemeinsames Verhalten beziehungsweise Standardverhalten im Oberzustand zu definieren und in den Unterzuständen nur die lokalen Abweichungen anzugeben.

Ein Zustand kann auch mehrere parallele Unterregionen besitzen. Dabei ist in jeder Unterregion genau ein Unterzustand aktiv. Beim Verlassen eines Zustands werden automatisch alle Unterzustände verlassen. Dabei wird der unterste Zustand zuerst verlassen. Das Eintreten in einen zusammengesetzten Zustand erfolgt in inverser Reihenfolge.

Um die Konstellation an Unterzuständen wiederherstellen zu können, kann auch ein history-State hinzugefügt werden. Dabei wird zwischen deep und shallow history unterschieden. Bei shallow history wird nur der zuletzt aktive Unterzustand des Oberzustands wiederhergestellt. Bei deep history wird dies rekursiv angewendet.

Neben den Zuständen können auch die Transitionen zwischen ihnen Aktionen auslösen. Diese Aktionen entsprechen der Ausgabe in den endlichen Maschinen und bilden somit eine Kombination aus Mealy- und Moore-Maschine. Transitionen werden durch Ereignisse ausgelöst. Falls ein aktiver Unterzustand das Ereignis nicht behandelt, wird es an seinen Oberzustand weitergereicht. Sogenannte Guards können zu Transitionen hinzugefügt werden. Diese bilden eine Bedingung, die bestimmt ob die Transition durchgeführt werden kann. Dabei sollte darauf geachtet werden, dass die Guards für ein Ereignis sich in jedem Zustand gegenseitig ausschließen, damit die Wahl der Transition eindeutig ist.

Auch gibt es die Möglichkeit, beim Betreten oder Verlassen eines Zustands eine Aktion auszulösen. Diese werden meist Entry/Exit-Aktionen genannt.

Die UML State Machine Spezifikation basiert auf den Harel Statecharts [36]. UML hat keine eindeutige Semantik und Syntax (vgl. [46] und [12]). Die hier benutzte Schreibweise für Ereignisse orientiert sich am Signal-Syntax von *Qt*: `ereignisName( Parameter 1, ...)` – also zum Beispiel `routeCreated(string name)`. Auch unterscheidet UML ab Version 2.0 bei Transitionen zwischen drei Varianten: Interne Transitionen führen keine Zustandsänderung durch. Somit wird kein Zustand verlassen und kein Zustand betreten. Externe Transitionen verlassen einen Zustand und alle seine Oberzustände und betreten dann alle Oberzustände des Ziel-Zustands und diesen selbst. Lokale Transitionen sind wie externe Transitionen, verhalten sich aber bei einem gemeinsamen Oberzustand anders. In diesem Fall werden nur die Zustände unterhalb des gemeinsamen Oberzustands verlassen beziehungsweise betreten.

4.1.3 Eignung für GUI-Entwicklung

Grafische Oberflächen müssen oft auf Ereignisse reagieren. So muss auf Interaktion durch Maus, Tastatur und sonstige Eingabegeräte reagiert werden. Ebenfalls können Änderungen an zugrundeliegenden Models auftreten. Außerdem ist eine grafische Oberfläche meist zustandsbehaftet. Zum Beispiel muss verwaltet werden, welcher Reiter gerade aktiv ist oder ob ein Button aktiviert oder deaktiviert ist. Oft kommen dabei boolesche Variablen oder Aufzählungstypen zum Einsatz. Anstatt diesen Zustand durch solche Flags abzuspeichern, wird er explizit in der Maschine dargestellt. Dies soll zu einer besseren Strukturierung führen. Auch ist die Darstellung als Diagramm meist leichter zu überschauen, als die über mehrere Klassen verteilten Informationen. Zusammen mit Entry- und Exit-Aktionen sollten sich die Zustände gut kapseln lassen, damit sie nicht von anderen Zuständen abhängig sind. Die eigentlichen Teil-Funktionen sollen also von ihrer Kombination getrennt werden. Der Zustand, der zur Ansicht von Routen-Informationen dient, soll zum Beispiel nur wissen, wie er diese Informationen darstellt. Wodurch der Zustand ausgelöst wird, wird durch die Konstruktion der endlichen Maschine bestimmt.

4.2 Entwurf der UML State Machines

4.2.1 Steuerung des MapViewers

Abbildung 11 zeigt den aktuellen Planungsstand der Maschine. Dabei ist jede Hierarchie-Ebene anders gefärbt.

Der neue *MapViewer* bietet die Möglichkeit, eine Route oder Region einzeln auf einer weiteren Karte anzuzeigen. Dieser so genannte *SplitView* hat keinerlei Auswirkung auf das restliche Verhalten des *MapViewers* und wird daher in einer parallelen Region modelliert.

Erstellende und bearbeitende Zustände, wie zum Beispiel das Anlegen einer Route, sollen so gestaltet sein, dass sie vom Benutzer explizit abgebrochen werden müssen. Während man in einem dieser Zustände ist, sollen die anderen Buttons und sonstigen Interaktionen gesperrt werden. Um diese Sperren der Oberfläche in allen bearbeitenden oder erstellenden Zuständen zu realisieren, wird zwischen einem `Editing mode` und einem `Normal mode` unterschieden. Für den `Editing mode` ergeben sich dann zunächst die beiden Unterzustände `Create route` und `Create region`. Die drei möglichen Varianten, eine Route anzulegen, bekommen dann jeweils ihren eigenen Unterzustand.

Im `Normal mode` gibt es zum einen den `Normal view`, welcher zu Beginn angezeigt wird. Daneben gibt es jeweils einen Zustand, der Informationen und weitere Aktionen für eine Route beziehungsweise Region anzeigt. Diese beiden Ansichten lassen sich in `Normal mode` öffnen, indem auf der linken Seite des *MapViewers* auf den passenden Eintrag geklickt wird. Damit nicht jeder Unterzustand von `Normal mode` einen Übergang zu `Route view` und `Region view` benötigt, wird die Hierarchie benutzt und jeweils nur eine lokale Transition vom `Normal mode` aus verwendet. Dies hält die Anzahl der Transitionen klein und somit übersichtlicher. Auch besteht keine Gefahr, dass man, wenn man zu `Normal mode` einen weiteren Unterzustand hinzufügt, diese Transitionen vergisst.

Das gleiche Prinzip wurde genutzt, um beim Abbruch aus dem `Editing mode` in den `Normal View` zu wechseln und beim erfolgreichen Erstellen einer Route, egal auf welche Weise, den passenden `Route View` zu öffnen.

Die Migration des bisherigen *Route-/RegionViewers* ist noch nicht genau abgesprochen, aber geplant. Der Automat ist daher noch nicht vollständig. `Route view` und `Region view` werden beide Unterzustände bekommen. Auch wird es zum Beispiel für den `Route view` voraussichtlich einen weiteren Unterzustand von `Editing mode` geben, da für die Segmente einer erstellten Route Parameter eingetragen werden können und dieser Prozess ebenfalls explizit abgebrochen werden soll.

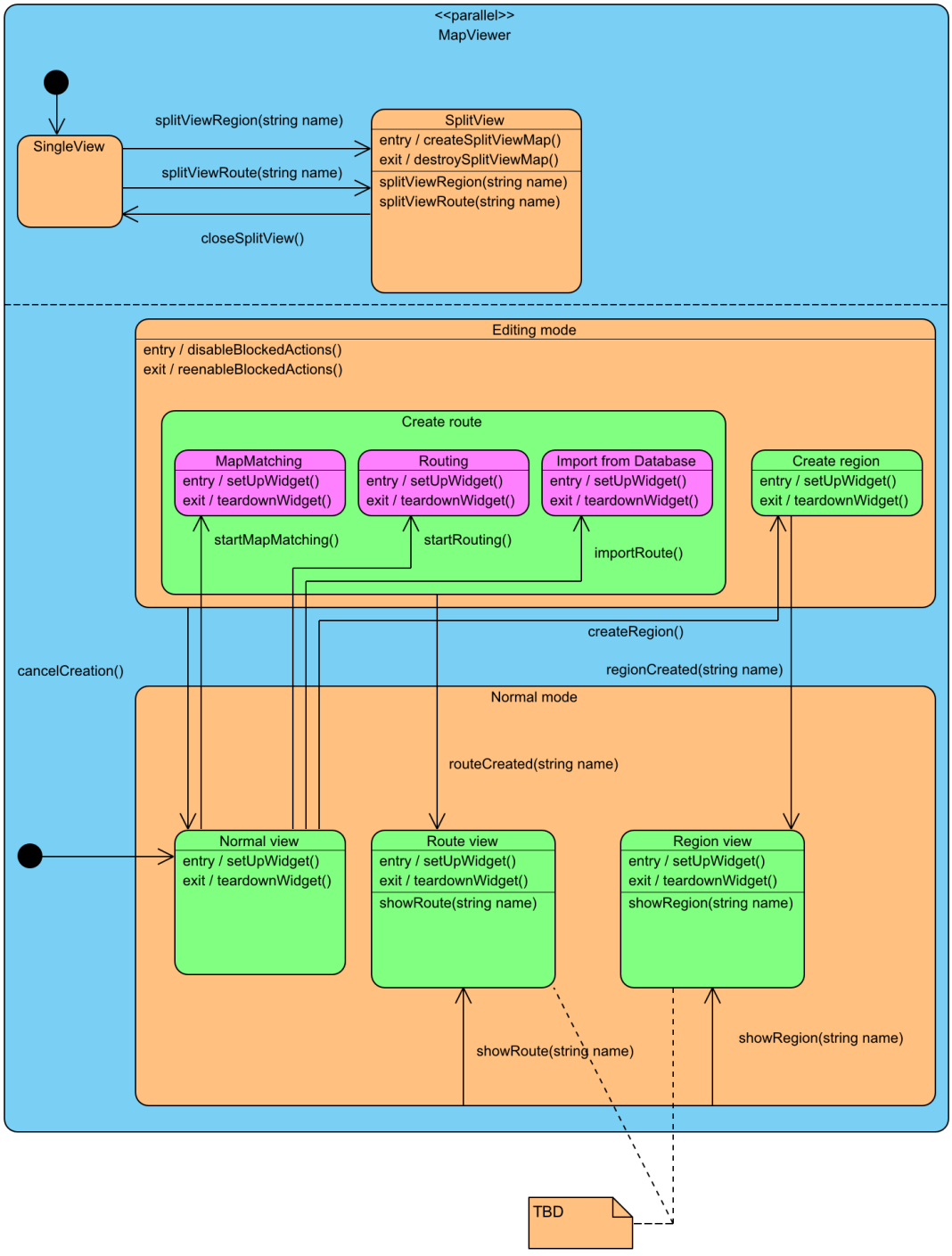


Abbildung 11: Aktueller Planungsstand für die State Machine auf C++-Seite

4.2.2 Steuerung der Karte

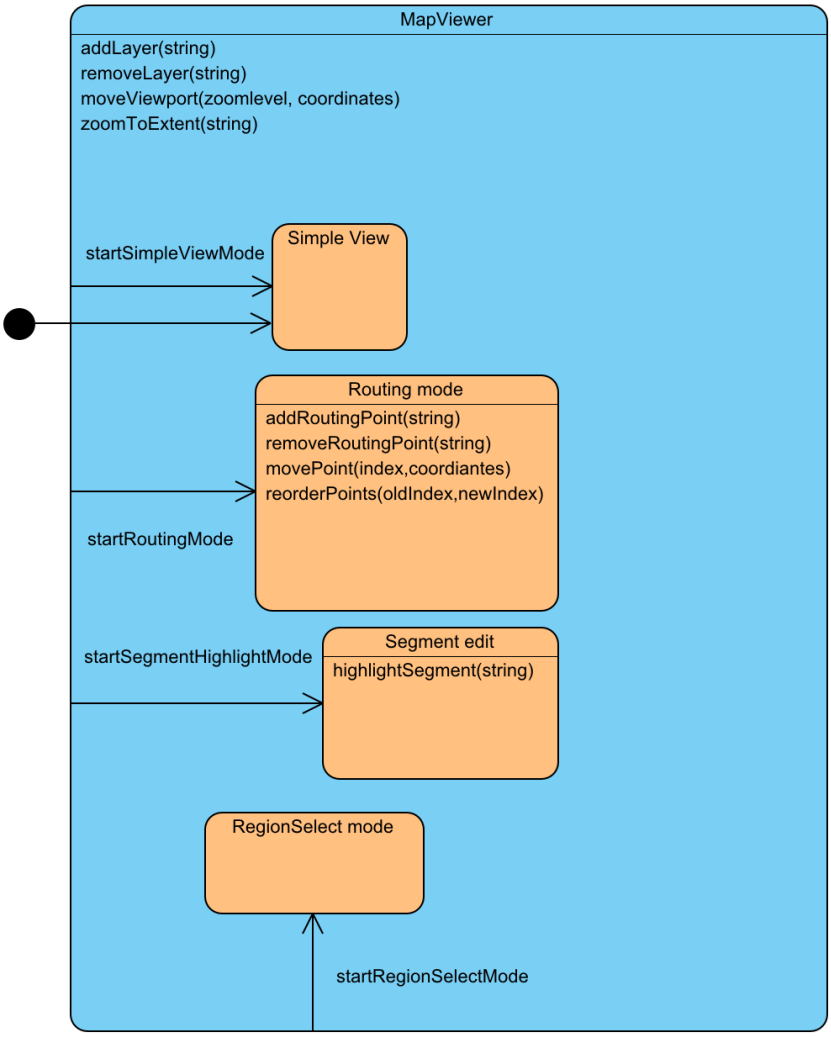


Abbildung 12: Aktueller Planungsstand für die State Machine auf TypeScript-Seite

Die endliche Maschine zur Steuerung der Karte ist deutlich kleiner (vgl. Abbildung 12), da sie nur die unterschiedlichen Interaktionsmodi abdecken muss. Bisher gibt es vier Modi: eine einfache Ansicht der Karte (`Simple View`), einen `Routing mode`, einen `RegionSelect mode` und den `Segment edit` Modus.

Im `Routing mode` können Routing-Punkte gesetzt und bearbeitet werden. Das Erstellen von Regionen erfolgt unter anderem durch die Auswahl von administrativen Bereichen auf der Karte. Hierfür ist der `RegionSelect mode` gedacht. Im `Segment edit` Zustand wird jeweils das Segment hervorgehoben, welches im neuen *RouteViewer* gerade bearbeitet wird.

4.3 Wahl der technischen Umsetzung auf C++-Seite

4.3.1 Kriterien und Auswahl der Optionen

Es besteht die Wahl zwischen bestehenden Bibliotheken/Frameworks und einer eigenen Implementierung der State Machine Logik. In Absprache mit dem Entwicklerteam wurden folgende Kriterien gewählt:

Kostenfrei Es dürfen keine zusätzlichen Kosten entstehen.

Kompatible Lizenz Die Lizenz darf keine Auswirkung auf die VMC-Lizenz haben. Die Copyleft-Lizenzen, wie zum Beispiel die *GNU General Public License* [18], sind also zum Beispiel nicht nutzbar.

Unterstützung von UML State Machine Die Unterstützung von Unterzuständen und parallelen Regionen ist für die einfache Modellierung wichtig. So erlauben Unterzustände zum Beispiel das Auslagern von Gemeinsamkeiten in einen Oberzustand.

Wartung und Zukunftssicherheit Die benutzte Variante muss noch aktiv gewartet werden. Das Risiko wäre sonst zu hoch, dass beim Entdecken von Fehlern diese entweder selbst behoben oder dass auf eine andere Variante migriert werden muss.

Support und Dokumentation Die Variante muss ausreichend dokumentiert sein oder es muss zumindest eine zuverlässige Möglichkeit geben, um an Informationen zu gelangen.

Zusätzliche Abhängigkeiten Zusätzliche Abhängigkeiten, zum Beispiel in Form von Bibliotheken, sollen laut Vorgabe möglichst vermieden werden.

Einarbeitungsaufwand Da an *VMC* mehrere Entwickler arbeiten, sollte der Einarbeitungsaufwand gering sein.

Bei der Auswahl der potenziellen Optionen muss darauf geachtet werden, dass keine Lizenzkosten verursacht werden dürfen, womit alle kommerziellen Angebote (außer *Qt*) entfallen. Auch sollten laut Vorgabe neue Abhängigkeiten, wenn möglich, vermieden werden. Dies liegt unter anderem an der nötigen aufwendigen Prüfung der Lizenz sowie Ungewissheit über die weitere Entwicklung. Da bereits einige *Boost* Bibliotheken verwendet werden und für *Qt* eine Lizenz vorhanden ist, werden folgende Optionen in Betracht gezogen und untersucht:

- *Boost Meta State Machine (MSM)* [22]
- *Boost Statechart* [8]
- *Qt State Machine Framework* [43]
- *State Pattern* [52]

4.3.2 Analyse der Optionen

Boost Meta State Machine (MSM) Laut Dokumentation [21] besteht *MSM* aus einem back-end und einem front-end. Für das back-end gibt es nur eine Umsetzung. Beim front-end stehen drei Optionen zur Verfügung. Zum einen gibt es ein einfaches front-end, welches jedoch nur für kleine State Machines empfohlen wird. Zum anderen gibt es ein *eUML* front-end, das jedoch als experimentell bezeichnet wird. Im Allgemeinen wird das Funktor front-end empfohlen. Alle drei front-ends benutzen eine Liste aller Transitionen zur Definierung der State Machine.

MSM unterstützt, laut Dokumentation, einen Großteil der UML 2 Standards. Auch ist es als aktives *Boost*-Projekt gewartet und auf absehbare Zeit zukunftssicher. Die Dokumentation ist umfangreich und über die E-Mailliste ist Support verfügbar.

Allgemein verwendet *MSM* viele Templates. Dabei benutzt *MSM* auch viele weitere *Boost*-Bibliotheken. Der Einsatz von *MSM* würde also zu einigen neuen Abhängigkeiten führen. Templates können schwer lesbare Fehlermeldungen hervorrufen [14].

Boost Statechart Laut Dokumentation [9] unterstützt *Boost Statechart* kein UML 2.0. Somit sind alle Transitionen extern. Dies grenzt die Modellierung stark ein, da jedes Mal der aktuelle Zustand verlassen wird. Interne Transitionen würden es erlauben, beim Eintreten in den Zustand GUI-Elemente zu erstellen und beim Austreten diese wieder zu zerstören. Die erstellte Maschine lässt sich somit nicht in der entwickelten Form umsetzen. Aus diesem Grund wird *Boost Statechart* nicht weiter betrachtet.

Qt State Machine Framework Laut Dokumentation [43] orientieren sich die Notation und die Konzepte an Harels Statechart [20]. Die Semantik orientiert sich jedoch an *State Chart XML (SCXML)* [24]. Zwischen SCXML und UML gibt es kleinere Unterschiede. Einen Überblick darüber gibt „Comparing UML, SCXML and the Apache implementation of SCXML engine“[31].

Alle benötigten Modellierungsmöglichkeiten, wie zum Beispiel Hierarchien, parallele Regionen und Entry/Exit-Aktionen, sind aber vorhanden. Da *VMC* mittels *Qt* realisiert ist, ist die Zukunftssicherheit nicht gesondert zu betrachten. *Qt* ist ohnehin ein bekanntes und aktives Projekt, das auch kommerziell weiterentwickelt wird. Die Dokumentation ist detailliert und auch Support ist vorhanden. Da *Qt* bereits benutzt wird, würde nur ein weiteres Modul hinzukommen. Dies ist eine recht kleine Abhängigkeit, welche aber bei jedem Release auch auf Kompatibilität mit den restlichen *Qt*-Modulen getestet wird.

Das *Qt State Machine Framework* benutzt den Signal-Slot-Mechanismus sowie das Event-System von *Qt*, welche auch im restlichen *VMC* benutzt werden. So kann eine Transition zum Beispiel durch ein Signal ausgelöst werden. Dies bietet sich bei kleinen Maschinen an, bei denen keine Parameter übergeben werden sollen. Für komplexere Maschinen bietet sich das Event-System an, wobei jedes unterschiedliche Event als eine eigene Unterklasse von `QEvent` realisiert wird. Somit können an das Event alle benötigten Informationen angehängt werden. Dieses Event wird dann an die `QStateMachine` gesendet.

Die Einarbeitung eines weiteren Entwicklers ist also aufgrund der bekannten Mechanismen und der guten Dokumentation recht einfach.

State Pattern Das *State Pattern*, wie in [52] beschrieben, bietet ein Muster zur Konstruktion von Zuständen. Dabei sind Hierarchien und parallele Regionen aber noch nicht berücksichtigt und müssten selbst geschrieben werden. Auch muss bei der Änderung der unterstützten Ereignisse die gesamte State Machine erneut kompiliert werden. Es wären keine zusätzlichen Abhängigkeiten nötig. Die Dokumentation müsste selbst erstellt werden, um den Einarbeitungsaufwand gering zu halten.

4.3.3 Wahl der Option

Basierend auf den oben beschriebenen Erkenntnissen wurde eine Einteilung in positiv(+), neutral(o) und negativ(-) vorgenommen (vgl. Tabelle 1). Dabei bedeutet eine negative Bewertung, dass diese Option nicht benutzt werden sollte.

Name	UML-Unterstützung	Wartung und Zukunftssicherheit	Support und Dokumentation	Abhängigkeiten	Einarbeitungsaufwand	Kommentar
Boost Meta State Machine	+	o	+	o	o	
Boost Statechart	-	o	+	o	o	Untersützt keine UML 2.0.
Qt State Machine Framework	+	+	+	+	+	
State Pattern	-	+	-	+	-	Hoher Aufwand, da alles selbst implementiert werden muss.

Tabelle 1: Zusammenfassende Bewertung der C++ Optionen

Wie man in Tabelle 1 erkennen kann, scheiden *Boost Statechart* und das *State Pattern* aufgrund der geringen UML Unterstützung aus.

Boost Meta State Machine und das *Qt State Machine Framework* haben beide keine negative Bewertung. Die Wahl fällt auf das *Qt State Machine Framework*, da die Integration mit dem restlichen Programm, welches ebenfalls auf *Qt* basiert, detaillierter ist und auch aufgrund der gewohnten Dokumentation zu einem geringeren Einarbeitungsaufwand führt.

4.4 Wahl der technischen Umsetzung auf JavaScript-/TypeScript-Seite

4.4.1 Kriterien und Auswahl der untersuchten Optionen

Es gelten die gleichen Kriterien wie für die C++-Seite. Da eine kostenlose Option gesucht wird und die Lizenz zu *VMC* kompatibel sein muss, werden nur Open Source Projekte in Betracht gezogen. Der wohl prominenteste Hoster für Open Source Projekte ist Github [17]. So schließt Google sein eigenes Angebot Google Code und bietet eine Migration nach Github an [7]. Auch Microsoft migriert einige eigene Projekte nach Github [2]. Daher wurde mittels Google nach „*hierarchical state machine javascript site:github.com*“ (8. Juni 2015) gesucht. Es wurde nach „javascript“

gesucht anstatt „typescript“, da TypeScript noch recht jung ist und die Wahrscheinlichkeit, dass „javascript“ auf möglichen TypeScript-Seiten vorhanden ist, recht hoch eingeschätzt wurde.

Die ersten beiden Seiten an Treffern wurden durchsucht. Einige Projekte sind aufgrund von Inaktivität, nicht vorhandener Dokumentation oder Lizenzproblemen direkt ausgeschieden. Die verbleibenden Optionen sind:

- *machina* [4]
- *state.js* [30]
- *Statechart* [10]

4.4.2 Analyse der Optionen

machina *machina* ist ein aktives Projekt. Bisher haben 15 Personen daran mitgearbeitet. Es gibt mehrere größere Beispiele und eine ausführliche Dokumentation. Die Unterstützung der UML Spezifikation ist nicht erwähnt. Es gibt zwar Hierarchien und Entry/Exit-Aktionen, jedoch ist nicht erklärt, ob Transitionen lokal oder extern sind.

state.js *state.js* ist auch ein aktives Projekt, welches hauptsächlich von einer Person entwickelt wird. Die aktuelle Versionsnummer beträgt 5.4.4, was auf andauernde Aktivität hindeutet. Neben einer ausführlichen Dokumentation gibt es auch zwei Beispiele. Hierarchien, Entry/Exit-Aktionen sowie lokale und externe Transitionen werden unterstützt. *state.js* wird mit TypeScript entwickelt und bietet somit Typ-Informationen. Die dadurch ermöglichte Vervollständigung durch Visual Studio erleichtert die Einarbeitung deutlich.

Statechart *Statechart* ist seit 9 Monaten inaktiv. Der letzte Commit war am 23. September 2014. Auch ist die Dokumentation sehr knapp. Hierarchien und Entry/Exit-Aktionen werden unterstützt. Bezüglich lokalen und externen Transitionen gibt es keine Aussage.

4.4.3 Wahl der Option

Wie bei der C++-Seite wurde auch hier eine Einteilung in positiv(+), neutral(o) und negativ(-) vorgenommen.

Wie aus Tabelle 2 ersichtlich wird, wird *Statechart* aufgrund der Inaktivität und der knappen Dokumentation ausgeschlossen. *machina* und *state.js* sind beide aktive Projekte mit guter Dokumentation. Jedoch ist die von *state.js* ausführlicher. Die zusätzlichen TypeScript-Typ-Informationen von *state.js* sind bei der Entwicklung hilfreich, da so einige Fehler direkt gefunden werden können. Außerdem ist aufgrund der möglichen Vervollständigung die Einarbeitung von neuen Entwicklern leichter. Deshalb wird *state.js* verwendet.

Name	UML-Unterstützung	Wartung und Zukunftssicherheit	Support und Dokumentation	Abhängigkeiten	Einarbeitungsaufwand	Kommentar
machina	o	o	+	o	o	
state.js	+	o	+	o	+	Wird in TypeScript entwickelt.
Statechart	o	-	o	o	o	Seit 9 Monaten inaktiv.

Tabelle 2: Zusammenfassende Bewertung der TypeScript Optionen

Kapitel 5

Iterative Entwicklung

Die Entwicklung erfolgt in mehreren Iterationen. Dabei wird die Maschine nach und nach um Zustände erweitert. Im Folgenden wird jeweils der Inhalt der Iteration und die dabei gemachten Erfahrungen beschrieben.

5.1 Iteration 1

Bisher wurden weder das *Qt State Machine Framwork*, noch `QWebEngine` benutzt. Ziel der ersten Iteration ist daher ein fast leeres Modul in *VMC* zu integrieren und die beiden Maschinen mittels `QWebChannel` miteinander zu verbinden. Der eigentliche *MapViewer* besteht im `Normal view` Zustand, wie in Abbildung 13 gezeigt, nur aus einer Karte und auf der rechten, kontextspezifischen Seite einem Button, der eine Transition in den (noch leeren) Routing-Zustand auslöst, wie er in Abbildung 14 zu sehen ist. Es wird keine Konfiguration geladen und die angezeigte Karte benutzt fest eingetragene Ebenen.

Während der Implementierung fiel zum einen auf, dass der `QWebChannel` noch nicht umfassend dokumentiert ist. `QWebChannel` wurde erst mit *Qt* 5.4 (10. Dezember 2014) eingeführt und ist somit recht neu. Die Dokumentation gibt zwar ein Beispiel, dabei werden jedoch nur einfache Datentypen wie `string` und `int` übergeben. Ob und wie zum Beispiel Arrays oder Vektoren übergeben werden können, ist nicht gut dokumentiert.

Ein anderes Problem ist die Entwicklung der TypeScript-Komponente. Aufgrund der `QWebChannel`-Abhängigkeit kann diese Komponente nur in der `QWebEngine` ausgeführt werden. Somit kann aber nicht die Debug-Funktion von Visual Studio genutzt werden.

Eine Möglichkeit ist es, ein weiteres Visual Studio Web-Projekt anzulegen und darin zu entwickeln. Dabei muss die Kommunikation mittels *QWebChannel.js* auskommentiert werden. Dieses Projekt würde aber auch bedeuten, dass auf allen Rechnern die TypeScript-Unterstützung installiert werden muss.

Eine andere Option, die in der Zeit nach der Bachelorarbeit untersucht werden wird, ist der Einsatz von *Vorlonjs* [54]. Dabei handelt es sich um eine remote debugging platform die mittels

eines *node.js* [32] Servers funktioniert. Somit müsste nur ein simples JavaScript-Skript in der `QWebEngine` eingefügt werden. Das Debuggen mittels Konsole kann dann über einen Client im Browser erfolgen. Der *node.js* Server kann zentral installiert werden.

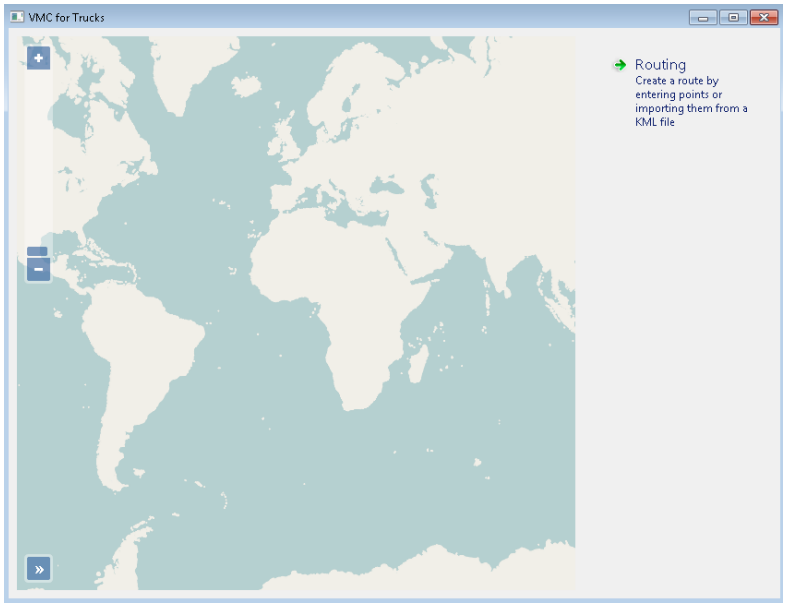


Abbildung 13: `Normal view` Zustand – Iteration 1

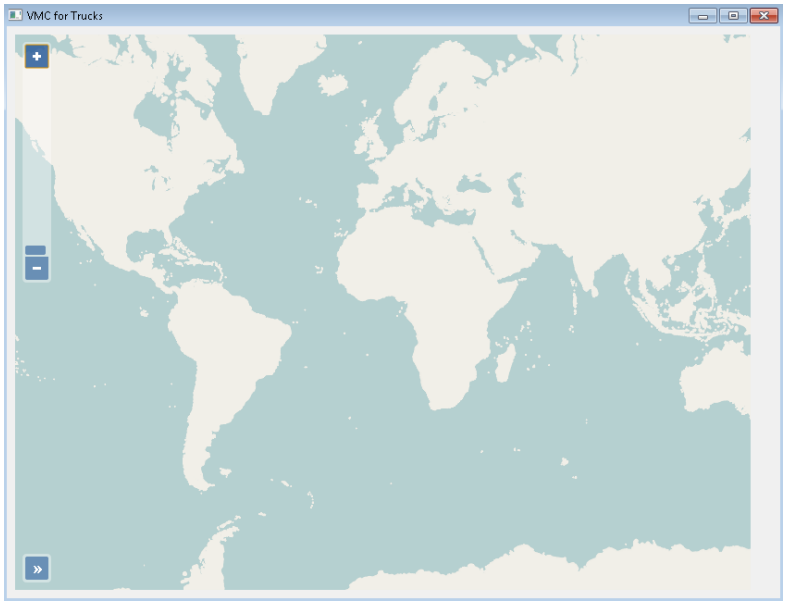


Abbildung 14: leerer `Routing` Zustand – Iteration 1

5.2 Iteration 2

In dieser Iteration geht es darum, den Routing-Zustand so auszubauen, wie er in Abbildung 15 zu sehen ist. Zum einen also die C++-Seite, welche hauptsächlich das Model für die Routing-Punkte sowie die GUI-Elemente umfasst, und zum anderen die JavaScript-Seite, bei der die Routing-Punkte auf der Karte angelegt, verschoben und gelöscht werden können. Nach dem erfolgreichen Routing wird dann in den noch leeren `Route view` gewechselt. Da bei dieser Transition der Name der erstellten Route übergeben werden soll, wird ein passendes Event von `QEvent` abgeleitet.

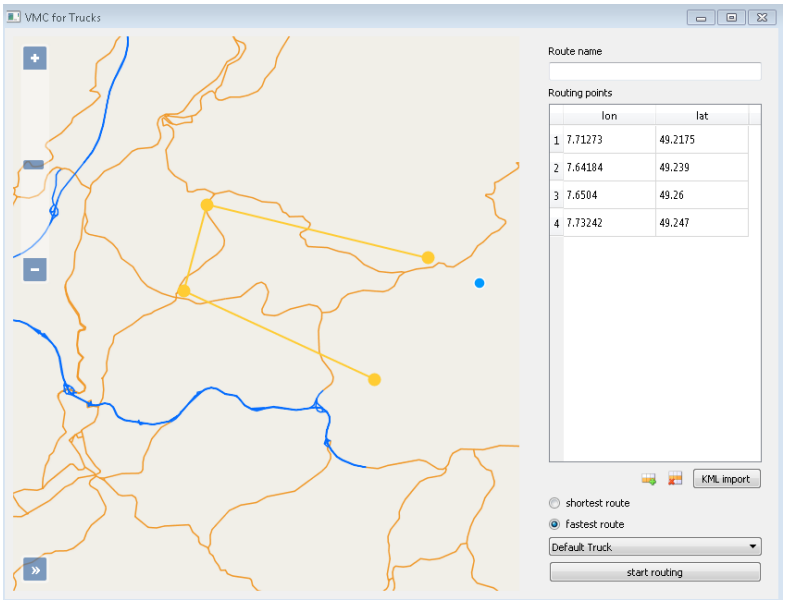


Abbildung 15: Funktionsfähiger `Routing` Zustand – Iteration 2

5.3 Iteration 3

In dieser Iteration soll die erstellte Route dann auch auf der Karte sichtbar sein. Dazu musste zum einen das Model der Routen und Layer implementiert werden. Dabei werden zunächst für jede Route nur die beiden Layer *Geometry* und *GPS points* angelegt. Zum anderen wurde auf der linken Seite ein `QTreeView` eingefügt, der zur Überprüfung des Models dient. Der simple *RouteViewer* auf der rechten Seite zeigt lediglich den Routen-Namen und bewirkt, dass beim Eintreten in den `Route view` Zustand die Route sichtbar ist. Auf der Karte wird die Route mit einem Standard-Style angezeigt. Der somit entstandene `Route view` Zustand ist in Abbildung 16 dargestellt.

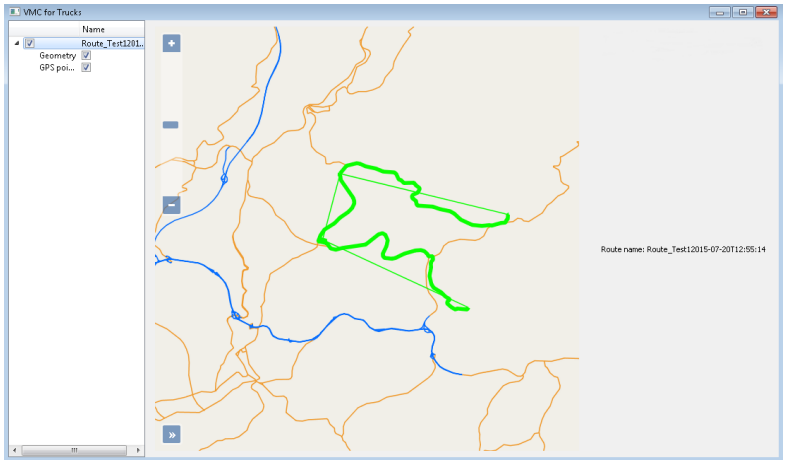


Abbildung 16: `Route view` Zustand – Iteration 3

5.4 Iteration 4

In dieser Iteration soll der bisher entstandene *MapViewer* in sich abgeschlossen werden: Zum einen ist das Routing abbrechbar und auch aus dem `Route view` kann man in den `Normal view` wechseln.

Für das Abbrechen des Routings wird ein weiteres Event erstellt. Wie in Abbildung 17 zu sehen ist, wird in die GUI des Routing-Zustands ein *cancel* Button integriert, der dieses Event an die State Machine schickt. Zusätzlich wird die Transition vom `Editing mode` in den `Normal view` Zustand hinzugefügt.

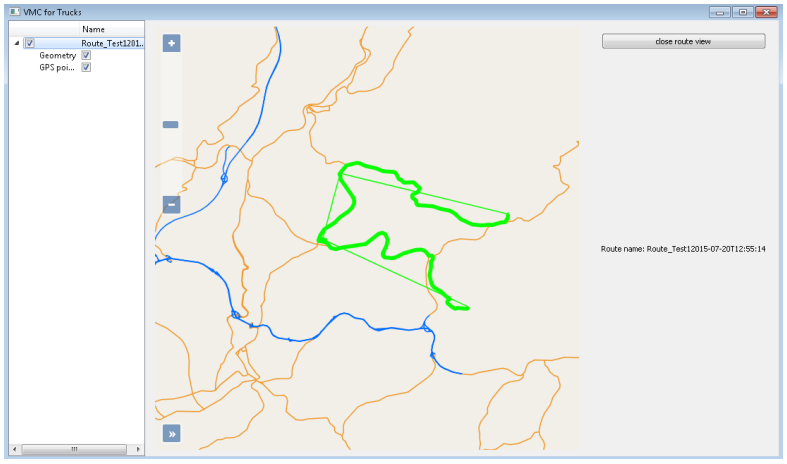


Abbildung 17: Schließbarer `Route view` Zustand – Iteration 4

Analog wird auch für den `Route view` Zustand vorgegangen.

Zusätzlich wird der `QTreeView` durch eine davon abgeleitete Klasse `CLayerTreeView` ersetzt.

Diese reagiert auf Doppelklicks und sendet ein passendes *showRoute(string name)* Event.

Die Änderungen in dieser Iteration waren jeweils nur klein und klar ersichtlich. An sich ging es nur um das Hinzufügen der benötigten Transition und das Auslösen des Ereignisses.

Kapitel 6

Verifikation

6.1 Bereits genutzte Techniken

6.1.1 Unit-Tests mit Boost Test

Für VMC und die abteilungsinternen Bibliotheken gibt es Unit-Tests. Diese sind mit *Boost Test* [45] realisiert. Der continuous integration server *Jenkins* [27] lässt bei jedem Commit alle Tests laufen und lädt den generierten Output. Daraus wird dann zum Beispiel eine Trend-Grafik wie in Abbildung 18 erstellt. Ebenfalls ist das Testergebnis, wie zum Beispiel in Abbildung 19, einsehbar.

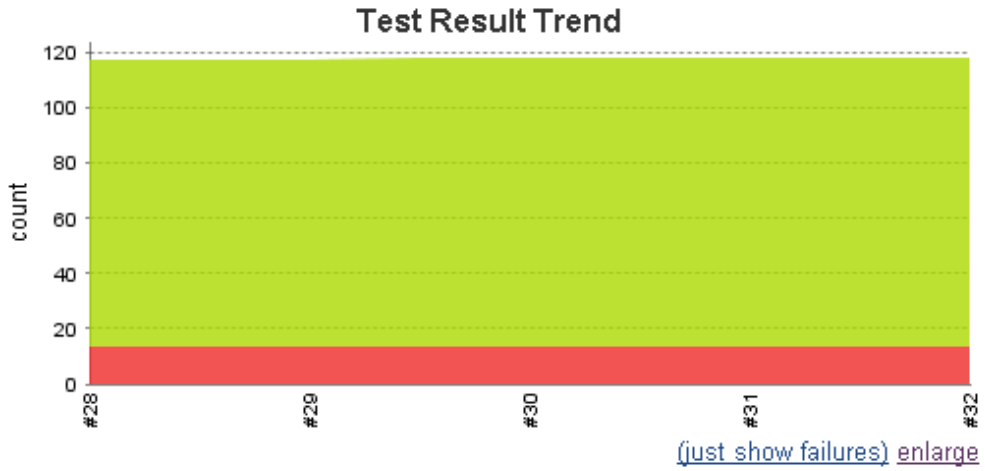
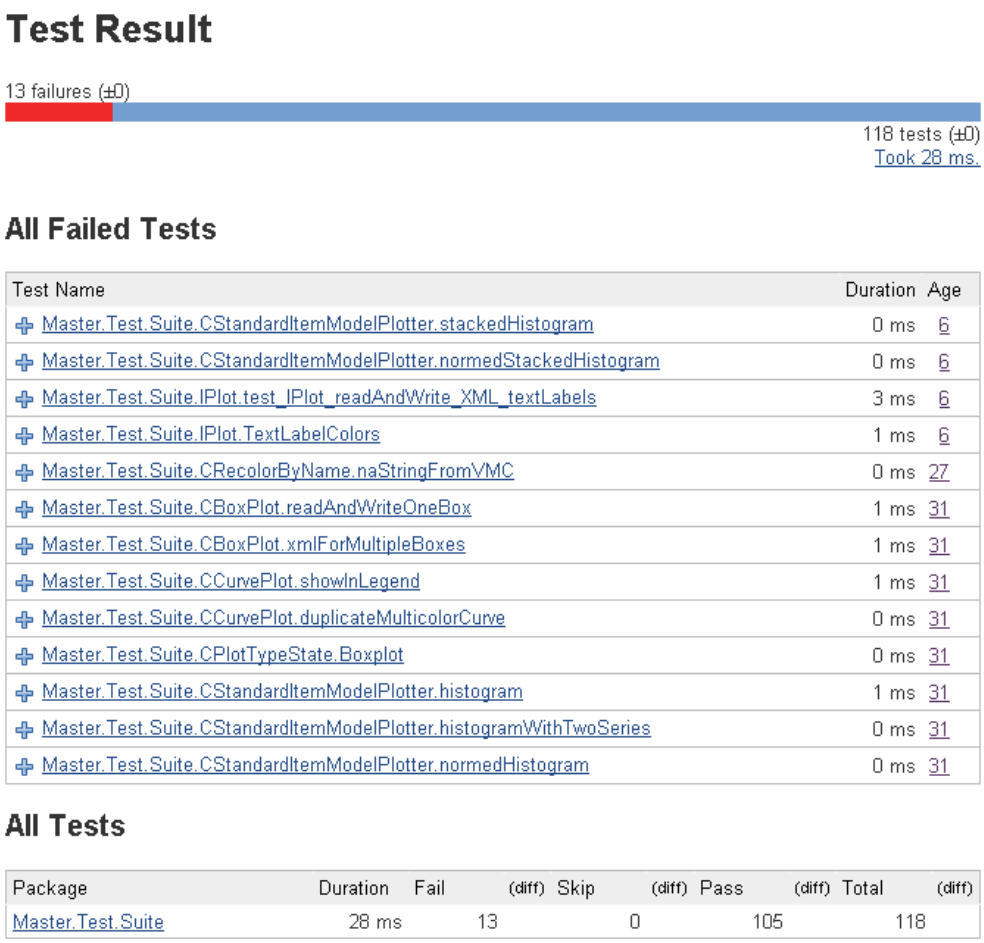


Abbildung 18: Trendgrafik des *Jenkins*-Servers

Es wird jedoch keine Test-Abdeckung berechnet. Visual Studio kann in der benutzten Variante keine Abdeckung berechnen. Auch für ein anderes kommerzielles Produkt sind keine Mittel vorhanden. Über die Qualität der Tests kann daher an dieser Stelle keine Aussage getroffen werden.



Test Result

13 failures (±0)

118 tests (±0)
Took 28 ms.

All Failed Tests

Test Name	Duration	Age
Master.Test.Suite.CStandardItemModelPlotter.stackedHistogram	0 ms	6
Master.Test.Suite.CStandardItemModelPlotter.normedStackedHistogram	0 ms	6
Master.Test.Suite.IPlot.test_IPlot_readAndWrite_XML_textLabels	3 ms	6
Master.Test.Suite.IPlot.TextLabelColors	1 ms	6
Master.Test.Suite.CRecolorByName.naStringFromVMC	0 ms	27
Master.Test.Suite.CBoxPlot.readAndWriteOneBox	1 ms	31
Master.Test.Suite.CBoxPlot.xmlForMultipleBoxes	1 ms	31
Master.Test.Suite.CCurvePlot.showInLegend	1 ms	31
Master.Test.Suite.CCurvePlot.duplicateMulticolorCurve	0 ms	31
Master.Test.Suite.CPlotTypeState.Boxplot	0 ms	31
Master.Test.Suite.CStandardItemModelPlotter.histogram	1 ms	31
Master.Test.Suite.CStandardItemModelPlotter.histogramWithTwoSeries	0 ms	31
Master.Test.Suite.CStandardItemModelPlotter.normedHistogram	0 ms	31

All Tests

Package	Duration	Fail	(diff)	Skip	(diff)	Pass	(diff)	Total	(diff)
Master.Test.Suite	28 ms	13		0		105		118	

Abbildung 19: Testbericht des *Jenkins*-Servers

6.1.2 TestLink

Zum Verwalten von manuellen Tests wird *TestLink* [48] eingesetzt. Es ermöglicht unter anderem das Anlegen von Tests, welche dann zu Testplänen gruppiert werden. Ein Teil eines Tests ist in Abbildung 20 zu sehen. Diese Testpläne werden dann Testern zugewiesen. Während der Ausführung können Kommentare zu den einzelnen Schritten des Tests eingetragen werden und der Erfolg erfasst werden. Aus den gesammelten Daten können dann Testreports wie der in Abbildung 21 erstellt werden.

Vor einer Auslieferung wird ein voller Testplan durchlaufen. In der Zeit dazwischen geschieht dies etwa alle ein bis zwei Wochen.

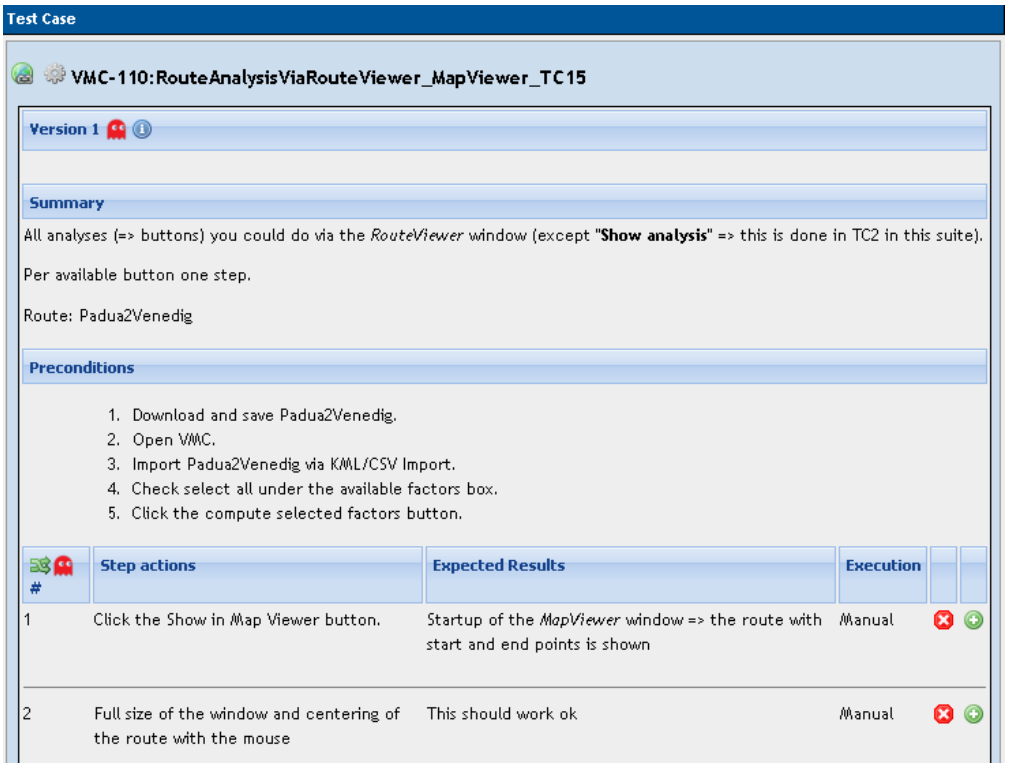


Abbildung 20: Teil eines Tests in *TestLink*

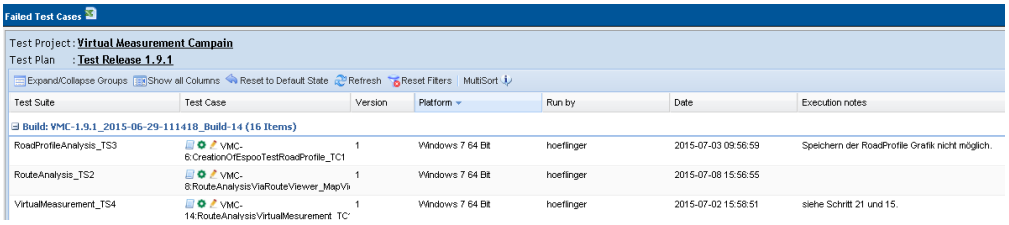


Abbildung 21: Teil des *Failed Test Cases* Berichts von *TestLink*

6.2 Neueingeführte Techniken

6.2.1 Neue connect-Syntax von Qt

VMC befindet sich aktuell in der Migration von *Qt* 4.8 auf *Qt* 5.5. Mit *Qt* 5 wurde eine neue Syntax zum Verbinden von Signals und Slots eingeführt.

Die alte, noch weiterhin verfügbare Syntax erzeugt erst zur Laufzeit einen Fehler, falls zum Beispiel der angegebene Slot nicht existiert. Die neue Syntax benutzt *Pointer-To-Member-Function* (vgl. [19]), sodass bereits beim Kompilieren der Software die Existenz überprüft wird. Auch wird so sofort bemerkt, falls das nötige `Q_OBJECT` Makro fehlt.

Der neue *MapViewer* nutzt ausschließlich die neue Syntax.

6.2.2 TypeScript

Durch die Typisierung können einige Fehler direkt gefunden werden, welche bei reinem JavaScript zu Laufzeitfehlern führen würden.

Unit-Tests wurden bisher noch nicht erstellt. Die Wahl des Frameworks hierfür wird erst nach der Bachelorarbeit geschehen. Zu diesem Zeitpunkt ist dann mehr Erfahrung im Umgang mit TypeScript verfügbar, welche die Auswahl erleichtern wird.

6.2.3 Statische Analyse mit Cppcheck

Das kostenlose Programm *Cppcheck* [5] wurde zur statischen Analyse benutzt. Dabei wurde nur der neue *MapViewer* mit seinen inkludierten Header-Dateien analysiert. Das Überprüfen des restlichen *VMC*-Quelltexts wäre im Rahmen der Bachelorarbeit zu aufwändig gewesen.

Beim Einsatz in diesem kleinen Rahmen hat *Cppcheck* unter anderem darauf hingewiesen, dass eine Funktion ihre Parameter als konstante Objekte, anstatt als konstante Referenzen entgegengenommen hat. Auch sind in *VMC* viele Konstruktoren, die nur einen Parameter haben, nicht als `explicit` markiert. Dies kann zu unerwünschten impliziten Typ-Konvertierungen führen, weswegen *Cppcheck* an diesen Stellen warnt.

In Zukunft wäre es möglich, *Cppcheck* auch mit dem *Jenkins*-Server zu integrieren, um mögliche Probleme frühzeitig zu erkennen.

6.2.4 GUI-Tests mit SikuliX

Da der neue *MapViewer* zu einem großen Teil aus Code für die grafische Oberfläche besteht, wurde noch nach Wegen gesucht, um automatische GUI-Tests durchzuführen.

In diesem Zusammenhang wurde *SikuliX* [23] getestet. Es ist eine Java-Anwendung zum Automatisieren und Testen von grafischen Oberflächen. Diese benutzt die `matchTemplate` Funktion aus *OpenCV*, um Elemente auf dem Bildschirm zu identifizieren. Daneben ermöglicht es die Simulation von Maus- und Tastatureingabe. Neben einer Java-API liefert *SikuliX* auch eine Python-API (Python 2.7) und eine IDE, in der kleinere Scripte schnell erstellt werden können.

Abbildung 22 zeigt ein solches Skript, das *VMC* startet, den neuen *MapViewer* öffnet, Punkte aus einer KML-Datei importiert und daraus eine Route erstellen lässt.

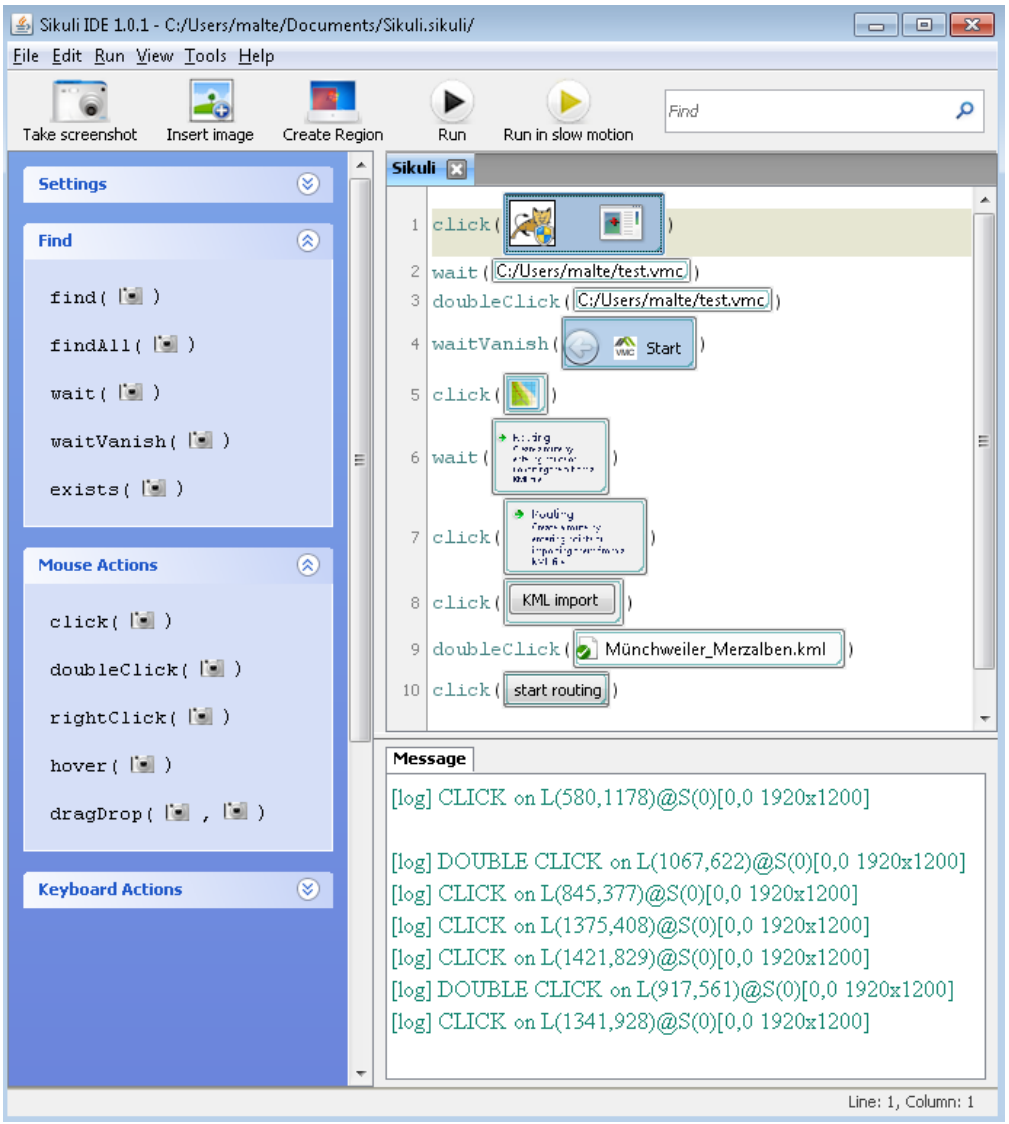


Abbildung 22: *SikuliX* Test

In Zukunft könnte die Python-API zusammen mit dem *Unittest*-Framework von Python kombiniert werden. Dadurch könnten GUI-Tests, ähnlich wie die Unit-Tests, in den *Jenkins*-Server integriert werden und bei jedem Commit oder in bestimmten Intervallen ausgeführt werden. Die Tests aus *TestLink* könnten migriert werden und ab dann automatisch ausgeführt werden. Durch die Automatisierung wäre es möglich, mehr Testfälle abzudecken und diese noch häufiger zu kontrollieren. So könnten beim Erstellen von Grafiken alle unterschiedlichen Kombinationen überprüft werden. Eine solche Abdeckung ist bei der aktuellen Teamgröße nicht realisierbar.

Diese Tests mit *SikuliX* würden dann also Integrations-Tests darstellen, welche bisher nur rein manuell durch das Nutzen der Software durchgeführt werden.

6.3 Weitere Techniken

6.3.1 Formale Methoden

UML State Machines können ohne Implementierung bereits überprüft werden. Auch können aus der Spezifikation Tests abgeleitet werden. Dabei gibt es zum Beispiel Kriterien, wie das Abdecken aller Transitionen. Bei der Verifikation von nebenläufigen Maschinen kann auch temporale Logik benutzt werden. Einen Überblick über diese formalen Methoden gibt „Improving Test Coverage for UML State Machines Using Transition Instrumentation“ ([13]).

Diese Methoden sind im Allgemeinen aber sehr aufwändig und für die kleine Maschine im *MapViewer* nicht angebracht.

6.3.2 Möglichkeiten unter Linux

VMC wird ausschließlich auf Windows als 64bit Anwendung entwickelt. Da jedoch hauptsächlich *Qt* und *Boost* benutzt werden, sollte es auch möglich sein, mit recht geringem Aufwand eine Linux-Variante zu erstellen. Lediglich die benutzten Bibliotheken Dritter könnten Probleme bei der Portierung hervorrufen.

Unter Linux stünden dann kostenlose Werkzeuge, wie zum Beispiel *Valgrind* [51] und die unterschiedlichen *Sanitizer* in *Clang* [3], zur Verfügung. Diese könnten dabei helfen, einige verborgene Fehler aufzufinden und die Testabdeckung zu bestimmen. Beide Werkzeuge instrumentieren die erstellten Programme und führen dabei zu Verlangsamung und erhöhtem Speicherverbrauch. Der Aufwand, diese Werkzeuge einzurichten, wäre hoch. Danach könnte das Entdecken von Problemen, wie zum Beispiel *memory leaks* und *use-after-free*, deutlich bei der Sicherung der Qualität helfen. Zusammen mit der GUI-Automatisierung durch *SikuliX* wäre die schlechte Performance jedoch tolerierbar, da kein Tester dabei warten müsste.

Kapitel 7

Fazit

Ziel dieser Arbeit war es, das neue Konzept des *MapViewers* als State Machine umzusetzen und neue Ideen zur Verifikation zu betrachten. Dabei sollte untersucht werden, ob die Modellierung eine klarere Struktur ermöglicht und inwiefern auf zukünftige Änderungen reagiert werden kann.

Nach bisherigem Stand der Erkenntnis hat sich die State Machine bewährt. Durch das Nutzen der Entry/Exit-Aktionen lassen sich die einzelnen Zustände kapseln. Dadurch können die Zustände wie eine Art Baukastensystem miteinander verbunden werden. Das Ausnutzen der Hierarchie zur Verringerung der Transitionen sorgt für Übersichtlichkeit. Auch ist das Diagramm leichter zu erfassen als Informationen, die über viele Klassen verstreut sind.

Bei der Entwicklung wurde darauf geachtet, Funktionalität und Verhalten zu trennen. So bietet der Routing-Zustand zum Beispiel die Funktionalität, eine Route zu erstellen. Das Verhalten, zum Beispiel in welchen Zustand nach dem Routing gewechselt wird, wird durch die State Machine bestimmt. Diese Trennung erleichtert es, die Struktur der State Machine zu ändern.

Bei der Erweiterung hat sich ein Muster ergeben, bei dem ein Zustand in einen generischen Oberzustand und einen spezialisierten Unterzustand aufgeteilt wird und dann ein weiterer Unterzustand hinzugefügt wird.

Die aktuelle Einschätzung ist insoweit nicht ganz unvoreingenommen, da zunächst die gesamte Maschine entworfen wurde und dann erst mit der Implementierung begonnen wurde. Inwiefern sich tatsächliche Änderungen am implementierten Automaten von den Änderungen während des Entwerfens unterscheiden, kann erst in einiger Zeit beurteilt werden.

Wichtig wird sein, dass das Diagramm der Maschine immer aktuell gehalten wird und nicht von der implementierten Variante abweicht. Auch muss das Diagramm übersichtlich gehalten werden. Sobald es nicht mehr leicht verständlich ist, besteht die Gefahr, dass Änderungen vollzogen werden, die unbeabsichtigte Effekte hervorrufen.

Die Umsetzung der restlichen Maschine wird nach Ende dieser Arbeit fortgesetzt.

Ebenfalls wird die Installation und Konfiguration des *GeoServer* zusammen mit dem weiteren Entwicklerteam in den Installationsvorgang von *VMC* integriert werden.

Die Nutzung von fertiger Software in Form von *GeoServer* und *OpenLayers* 3 hat eine schnelle Entwicklung ermöglicht. Gleichzeitig ist das Vertrauen in deren Korrektheit höher als das für den bisherigen *MapViewer*, unter anderem aufgrund der höheren Anzahl an Nutzern und Entwicklern.

Im Bereich der Verifikation haben sich *Cppcheck* und *SikuliX* bewährt. Eine kostenlose statische Analyse, die zusätzliche Hinweise auf mögliche Probleme liefert, ist äußerst nützlich. Das Untersuchen der restlichen Module wird wahrscheinlich schrittweise während der normalen Weiterentwicklung erfolgen. Auch wird eine Integration mit dem Jenkins-Server angestrebt werden, um mögliche Regressionen frühzeitig zu erkennen.

Die ersten Erfahrungen mit *SikuliX* sind vielversprechend. Es ist daher geplant, im Rahmen eines Studienprojekts, die bereits erwähnte Integration mit dem Python *Unittest*-Framework zu verfolgen. Dabei werden auch die ersten Tests aus TestLink migriert werden. Die somit frei werdenden Ressourcen können dann zur Weiterentwicklung von *VMC* genutzt werden.

Ob sich das neue Konzept tatsächlich bewährt, wird dann zunächst mit den abteilungsinternen Nutzern getestet. Wenn die Rückmeldung positiv ausfällt, werden auch mehrere Entwickler an der Umsetzung arbeiten und die UML State Machine erweitern. Dann wird sich besser einschätzen lassen, wie verständlich diese Modellierung ist. Die abschließende Bewertung des neuen Konzepts kann dann auf Basis von Rückmeldung der externen Kunden geschehen.

Literatur

[1] Abteilung MDF. *VMC® – Virtual Measurement Campaign (Flyer)*. Fraunhofer-Institut für Techno- und Wirtschaftsmathemarik ITWM, 2013. URL: https://www.itwm.fraunhofer.de/fileadmin/ITWM-Media/Abteilungen/MDF/Pdf/MDF_flyer_VMC_DE.pdf (besucht am 02. 08. 2015).

[2] Tim Anderson. „Microsoft snubs Codeplex, moves big projects to GitHub“. In: *The Register UK* (15. Jan. 2015). URL: http://www.theregister.co.uk/2015/01/15/codeplex_repository_out_of_favour_as_microsoft_moves_major_projects_to_github/ (besucht am 30. 06. 2015).

[3] *Clang 3.6 documentation*. 2015. URL: http://llvm.org/releases/3.6.0/tools/clang/docs/index.html (besucht am 26. 07. 2015).

[4] Jim Cowart. *machina.js*. URL: https://github.com/ifandelse/machina.js (besucht am 14. 07. 2015).

[5] *Cppcheck - A tool for static C/C++ code analysis*. URL: http://cppcheck.sourceforge.net/ (besucht am 26. 07. 2015).

[6] *Details of the object model*. Mozilla Developer Network. 5. Juni 2015. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model (besucht am 06. 08. 2015).

[7] Chris DiBona. „Bidding farewell to Google Code“. In: *Google - Open Source Blog* (12. März 2015). URL: http://google-opensource.blogspot.de/2015/03/farewell-to-google-code.html (besucht am 30. 06. 2015).

[8] Andreas Huber Dönni. *The Boost Statechart Library*. Boost. 9. Apr. 2007. URL: http://www.boost.org/doc/libs/1_57_0/libs/statechart/doc/index.html (besucht am 14. 07. 2015).

[9] Andreas Huber Dönni. *The Boost Statechart Library - Frequently Asked Questions (FAQs)*. 5. Jan. 2008. URL: http://www.boost.org/doc/libs/1_58_0/libs/statechart/doc/faq.html (besucht am 24. 06. 2015).

[10] David Durman. *statechart*. 23. Sep. 2014. URL: https://github.com/DavidDurman/statechart (besucht am 14. 07. 2015).

[11] *ECMAScript 6 compatibility table*. 5. Aug. 2015. URL: https://kangax.github.io/compat-table/es6/ (besucht am 06. 08. 2015).

[12] Harald Fecher u. a. „29 New Unclarities in the Semantics of UML 2.0 State Machines“. In: *Formal Methods and Software Engineering*. Springer Science + Business Media, 2005, S. 52–65. DOI: 10.1007/11576280_5. URL: http://dx.doi.org/10.1007/11576280_5.

[13] Mario Friske und Bernd-Holger Schlingloff. „Improving Test Coverage for UML State Machines Using Transition Instrumentation“. In: *Lecture Notes in Computer Science*. Springer Science + Business Media, 2007, S. 301–314. DOI: 10.1007/978-3-540-75101-4_30. URL: http://dx.doi.org/10.1007/978-3-540-75101-4_30.

[14] *Generate the longest error message in C++*. 15. März 2014. URL: https://codegolf.stackexchange.com/questions/1956/generate-the-longest-error-message-in-c (besucht am 26. 07. 2015).

[15] *GeoServer 2.7.0 User Manual*. 23. März 2015. Kap. Installation. URL: http://docs.geoserver.org/2.7.0/user/installation/index.html (besucht am 30. 06. 2015).

[16] *GeoServer : License*. 7. Juni 2012. URL: http://old.geoserver.org/License.html (besucht am 14. 07. 2015).

[17] *Github*. URL: https://github.com/ (besucht am 14. 07. 2015).

[18] *GNU General Public License*. Free Software Foundation, Inc. 29. Juni 2007. URL: http://www.gnu.org/licenses/gpl-3.0.en.html (besucht am 14. 07. 2015).

[19] Olivier Goffart. *Signals and Slots in Qt5*. Woboq GmbH. 12. Apr. 2012. URL: http://woboq.com/blog/new-signals-slots-syntax-in-qt5.html (besucht am 26. 07. 2015).

[20] David Harel. „Statecharts: a visual formalism for complex systems“. In: *Science of Computer Programming* 8.3 (Juni 1987), S. 231–274. ISSN: 0167-6423. DOI: 10.1016/0167-6423(87)90035-9.

[21] Christophe Henry. *Boost Meta State Machine Documentation*. 2014. URL: http://www.boost.org/doc/libs/1_58_0/libs/msm/doc/PDF/msm.pdf (besucht am 24. 06. 2015).

[22] Christophe Henry. *Meta State Machine (MSM)*. Boost. 12. Okt. 2014. URL: http://www.boost.org/doc/libs/1_57_0/libs/msm/doc/HTML/index.html (besucht am 14. 07. 2015).

[23] Raimund Hocke. *SikuliX - Homepage*. URL: http://www.sikulix.com/ (besucht am 26. 07. 2015).

[24] Rafah Hosn u. a. *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. Last Call WD. http://www.w3.org/TR/2014/WD-scxml-20140529/. W3C, Mai 2014.

[25] *Jahresbericht 2013/14*. Fraunhofer-Institut für Techno- und Wirtschaftsmathematik ITWM.

[26] *Jahresbericht 2014/15*. Fraunhofer-Institut für Techno- und Wirtschaftsmathematik ITWM.

[27] *Jenkins - Homepage*. URL: https://jenkins-ci.org/ (besucht am 26. 07. 2015).

[28] Kaspersky Lab. *Java under attack – the evolution of exploits in 2012-2013*. Kaspersky Lab, 30. Okt. 2013. URL: http://www.kaspersky.com/about/news/virus/2013/Kaspersky_Lab_study_reveals_surge_in_Java_exploit_attacks_to_14_1_million_in_a_year.

[29] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008. ISBN: 0132350882.

[30] David Mesquita-Morris. *state.js*. 13. Juli 2015. URL: `https://github.com/steelbreeze/state.js` (besucht am 14. 07. 2015).

[31] Nicola Migliorini. „Comparing UML, SCXML and the Apache implementation of SCXML engine“. In: (11. Apr. 2011). URL: `https://code.google.com/p/acsccg/source/browse/trunk/comodo/doc/gsmecg/mapping.pdf` (besucht am 25. 06. 2015).

[32] *Node.js®*. URL: `https://nodejs.org/`.

[33] OGC. *OpenGIS ® Web Map Server Implementation Specification*. Standard. Version 1.3.0. Open Geospatial Consortium Inc., 15. März 2006. URL: `http://www.opengeospatial.org/standards/wms`.

[34] OGC. *OpenGIS ® Web Map Tile Service Implementation Standard*. Standard. Version 1.0.0. Open Geospatial Consortium Inc., 6. Apr. 2010. URL: `http://www.opengeospatial.org/standards/wmts`.

[35] OGC. *Styled Layer Descriptor profile of the Web Map Service Implementation Specification*. Standard. Version 1.1.0 (revision 4). Open Geospatial Consortium Inc., 29. Juni 2007. URL: `http://www.opengeospatial.org/standards/sld`.

[36] *OMG Unified Modeling Language*. Standard. Version 2.5. Object Management Group, 1. März 2015. URL: `http://www.omg.org/spec/UML/2.5/PDF` (besucht am 24. 07. 2015).

[37] *OpenLayers 3*. URL: `http://openlayers.org/` (besucht am 14. 07. 2015).

[38] OSGeo. *WMS Tiling Client Recommendation*. 13. Okt. 2010. URL: `http://wiki.osgeo.org/wiki/WMS_Tiling_Client_Recommendation`.

[39] Steffen Polanski, Thomas Halfmann und Malte Schledjewski. *Systemspezifikationen: Gesamtsystemspezifikation*. Pflichtenheft. Fraunhofer-Institut für Techno- und Wirtschaftsmathematik ITWM, 8. Mai 2015.

[40] Steffen Polanski, Thomas Halfmann und Malte Schledjewski. *Systemspezifikationen : Softwarespezifikation und Systementwurf*. Fraunhofer-Institut für Techno- und Wirtschaftsmathematik ITWM, 8. Mai 2015.

[41] *Qt Documentation - Model/View Programming*. URL: `https://doc.qt.io/qt-5/model-view-programming.html` (besucht am 12. 07. 2015).

[42] *Qt Documentation - Qt WebChannel*. URL: `http://doc.qt.io/qt-5/qtwebchannel-index.html` (besucht am 14. 07. 2015).

[43] *Qt Documentation - The State Machine Framework*. URL: `https://doc.qt.io/qt-5/statemachine-api.html` (besucht am 12. 07. 2015).

[44] *Rendering*. OpenStreetMap Wiki. 15. Mai 2015. URL: `https://wiki.openstreetmap.org/wiki/Rendering` (besucht am 05. 08. 2015).

[45] Gennadiy Rozental. *Boost Test Library*. 27. Juli 2008. URL: `http://www.boost.org/doc/libs/1_57_0/libs/test/doc/html/index.html` (besucht am 26. 07. 2015).

[46] Bernhard Rumpe. *Modellierung mit UML: Sprache, Konzepte und Methodik (Xpert.press)*. Springer, 2011. ISBN: 3642224121.

[47] Malte Schledjewski. *MapViewer - eine Softwarekomponente zur Visualisierung statistischer und georeferenzierter Daten: Requirements Engineering und Spezifikation*. Bericht zum Praxisprojekt. 22. Mai 2015.

[48] *TestLink Open Source Test Management*. URL: `http://testlink.org/` (besucht am 26. 07. 2015).

[49] *this*. Mozilla Developer Network. 17. Juli 2015. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this` (besucht am 06. 08. 2015).

[50] *TypeScript*. URL: `http://www.typescriptlang.org/` (besucht am 06. 08. 2015).

[51] *Valgrind*. URL: `http://valgrind.org/` (besucht am 26. 07. 2015).

[52] John Vlissides u. a. *Entwurfsmuster*. Aus dem Englischen übers. von Dirk Riehle. Addison Wesley Verlag, 2004. ISBN: 3827321999.

[53] *VMC ® - Virtuelle Messkampagne*. Fraunhofer-Institut für Techno- und Wirtschaftsmathematik ITWM. URL: `https://www.itwm.fraunhofer.de/abteilungen/mdf/leistungen-und-produkte/virtuelle-messkampagne.html` (besucht am 03. 08. 2015).

[54] *Vorlon.JS*. 24. Juni 2015. URL: `http://www.vorlonjs.com/` (besucht am 26. 07. 2015).

[55] Christian Wagenknecht und Michael Hielscher. *Formale Sprachen, abstrakte Automaten und Compiler: Lehr- und Arbeitsbuch für Grundstudium und Fortbildung*. Vieweg+Teubner Verlag, 2009. ISBN: 3834806242.

Anhang

CD mit:

- Quelltext des neuen *MapViewer*
- Bericht zum Praxisprojekt *MapViewer - eine Softwarekomponente zur Visualisierung statistischer und georeferenzierter Daten: Requirements Engineering und Spezifikation* (vgl. [47])
- *Systemspezifikationen: Gesamtsystemspezifikation* (vgl.[39])
- *Systemspezifikationen : Softwarespezifikation und Systementwurf* (vgl.[40])
- Diese Arbeit als PDF und als LaTeX-Quelldatei inklusive aller Grafiken.