Höhere Technische Bundeslehranstalt für Informatik, Perg

Diplomarbeit

SARA
Situationsadaptive Routenfindung und 3D-Visualisierung

Projektteam:	Patrick Weiß David Raffetseder Daniel Lettner
Projektbetreuer:	Dipl.-Ing. Christian Reisinger
Bearbeitungszeitraum:	01.08.2015 - 08.04.2016

In Zusammenarbeit mit der Firma ABF – Industrielle Automation GmbH
Ansprechpartner: Dipl.-Ing. Andreas Sperneder

Eidesstattliche Erklärung

Hiermit versichern wir, die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der von uns angegebenen Quellen angefertigt zu haben. Alle Stellen, die wörtlich oder sinngemäß aus fremden Quellen direkt oder indirekt übernommen wurden, sind als solche gekennzeichnet.

Perg,______________________________ Unterschrift ______________________________
(Patrick Weiß)

Perg,______________________________ Unterschrift ______________________________
(David Raffetseder)

Perg,______________________________ Unterschrift ______________________________
(Daniel Lettner)

Danksagung

Wir bedanken uns bei all jenen, die uns im Laufe der Diplomarbeit unterstützt haben. Besonderen Dank möchten wir unserem Ansprechpartner der Firma ABF, Herrn Dipl.-Ing. Andreas Sperneder aussprechen, der uns bei allen technischen Fragen unterstützte und stets bemüht war, aufgetretene Probleme schnellstmöglich zu beseitigen. Unser Dank gilt ebenfalls unserem Betreuungslehrer, Herrn Dipl.-Ing. Christian Reisinger, der durch seine Unterstützung bei Planungs- und Dokumentationsarbeiten unser Leben erleichterte.

Inhaltsverzeichnis

1 Einführung **11**
1.1 Kurzfassung . . . 11
1.2 Abstract . . . 11
1.3 Motivation . . . 12

2 Projekt **13**
2.1 Projektteam . . . 13
2.2 Betreuungslehrer . . . 13
2.3 Auftraggeber . . . 14
2.3.1 ABF . . . 14
2.3.2 Ansprechpartner . . . 14
2.4 Geschäftsziel . . . 14
2.5 Projektziel . . . 15

3 Ausgangssituation **17**
3.1 Begriffserklärung . . . 18
3.1.1 Fahraufträge . . . 18
3.1.2 Transport-Objekte . . . 18
3.1.3 OneBase . . . 18
3.1.4 OneBase-Tag . . . 18
3.2 System OneBase-MFT . . . 19
3.2.1 OBSocketServer . . . 19
3.2.2 Synchronizer . . . 19
3.2.3 Warehouse Manager . . . 19
3.2.4 Supervisor . . . 19
3.2.5 MFTAdministrator . . . 20
3.2.6 PositionSimulator . . . 20
3.3 Fahrzeugortung und -verfolgung . . . 21
3.3.1 DGPS . . . 21
3.3.2 Zeno Track . . . 22

4 Grundlagen **25**
4.1 Theoretische Grundlagen . . . 25
4.1.1 Aufbau eines Lagers . . . 25
4.1.2 Graphentheorie . . . 27
4.2 Technische Grundlagen . . . 29
4.2.1 Technologien . . . 29
4.2.2 Entwicklungskonfiguration . . . 32

5 Benutzung **35**
5.1 Use Case Diagramm . . . 35
5.2 Ablaufdiagramme . . . 37
5.2.1 Navigation . . . 37
5.2.2 Mehrere Fahraufträge . . . 38
5.2.3 Routenfindung . . . 39

6 Programmbeschreibung **41**
6.1 Datenmodell . . . 41
6.1.1 Datenlexikon . . . 42
6.1.2 Properties . . . 44
6.2 Implementierung . . . 46
6.2.1 SARA-Manager . . . 46
6.2.2 Model . . . 49
6.2.3 Graph . . . 58
6.2.4 Navigation . . . 59
6.2.5 3D-Visualisierung . . . 68

7 Technische Daten **73**
7.1 Initialisierung von SARA . . . 74
7.2 Reinitialisierung des Weggraphen . . . 74
7.3 Suchen des nächsten Wegpunktes . . . 75
7.4 Abweichung von der Route . . . 76
7.5 Neuberechnung der optimalen Route . . . 78
7.6 Dijkstra-Algorithmus . . . 78

8 Projektvorgehen **79**
8.1 Planung . . . 81
8.1.1 Meilensteine . . . 81
8.1.2 Zeitplanung . . . 82

8.2 Verwendete Ressourcen 85
8.2.1 Hardware 85
8.2.2 Software 85
8.2.3 Orgware 85
8.3 Projektstrukturplan 87
8.4 Verantwortungsmatrix 88
8.5 Qualitätsmanagement 89
8.5.1 Qualitätsmerkmale 89
8.5.2 Maßnahmen zur Qualitätssicherung 89

9 Resümee und Ausblick **91**
9.1 Projektvorgehen 91
9.2 Kommunikation 91
9.3 Einarbeitung in neue Technologien 92
9.3.1 OneBase-MFT 92
9.3.2 Graphenbibliothek 92
9.4 Aufgetretene Probleme 92
9.4.1 Serverinstallation 92
9.4.2 JMonkeyEngine 93
9.5 Zukünftige Erweiterungen 93

1 Einführung

1.1 Kurzfassung

Die vorliegende Diplomarbeit befasst sich mit dem Thema der situationsadaptiven Routenfindung und 3D-Visualisierung für Transportfahrzeuge der industriellen Intralogistik und wurde im Auftrag der Firma ABF Industrielle Automation GmbH durchgeführt. Die Firma ABF erstellt Software-Lösungen zur Unterstützung von Logistikprozessen in unterschiedlichen Branchen der Industrie. Ziel dieser Arbeit ist die Erweiterung des Systems OneBase-MFT aus dem Bereich der Materialflusssteuerung.
Aufbauend auf dieses System wurde ein unabhängiges Modul (SARA) entwickelt, welches die optimale Route zu Lagerplätzen berechnet und diese in die bestehende 3D-Visualisierung integriert, wodurch eine Kosten- und Zeitersparnis für Transportfahrzeuge erreicht wird. Dabei wird die Navigation auf den jeweiligen Fahrzeugtyp angepasst und reagiert adaptiv auf mögliche Änderungen, wie zum Beispiel das Sperren von Wegen oder einer Abweichung von der optimalen Route durch den Lenker des Transportfahrzeuges.

1.2 Abstract

This thesis deals with the issue of route planning and 3D visualization for transportation vehicles of the industrial intralogistics on behalf of the ABF Industrielle Automation GmbH. The company ABF develops software solutions to support logistics processes in various branches of the industry sector. The aim of the project is to expand a system in the field of material flow control. This system will be extended to include a module for navigation (SARA), which calculates the optimal route to storage areas and integrates it into the existing 3D visualization, in order to save time and money for transport vehicles. The navigation will be adjusted to the type of the vehicle and respond adaptively to possible changes, such as the blocking of roads or a deviation from the optimal route through the driver of the transportation vehicle.

1.3 Motivation

Motivation für die Diplomarbeit ist der schonende Einsatz von Ressourcen in Lager- und Transporteinrichtungen der industriellen Intralogistik. Dieser schonende Einsatz ergibt sich aus der Kosten- und Zeitersparnis durch die optimale Routenfindung.
Die Diplomarbeit setzt auf ein bestehendes System der Firma ABF auf, welches unter anderem bereits den optimalen Lagerplatz für bestimmte Produkte berechnet. Ziel der Diplomarbeit ist die Erweiterung dieses Systems um die Wegoptimierung für Transportfahrzeuge.

2 Projekt

2.1 Projektteam

Name	Patrick Weiß
Aufgabengebiet	3D-Visualisierung
E-Mail	patrickweiss0@gmail.com

Name	David Raffetseder
Aufgabengebiet	Routenfindung
E-Mail	raffetseder.david@gmail.com

Name	Daniel Lettner
Aufgabengebiet	Business Logik, Datenbank
E-Mail	daniellettner1996@gmail.com

2.2 Betreuungslehrer

Name	Dipl.-Ing. Christian Reisinger
Funktion	Diplomarbeitsbetreuer
E-Mail	c.reisinger@htl-perg.ac.at

2.3 Auftraggeber

2.3.1 ABF

ABF Industrielle Automation GmbH
Deggendorfstraße 6
A-4030 Linz
+43 (0) 732 304030 0
office@abf.co.at
www.abf.at

2.3.2 Ansprechpartner

Name	Ing. Gustav Buchberger
Aufgabengebiet	Geschäftsführer ABF
Telefon	+43 732 304030 23
E-Mail	gustav.buchberger@abf.co.at

Name	Dipl.-Ing. Andreas Sperneder
Aufgabengebiet	Ansprechpartner
Telefon	+43 732 304030 14
E-Mail	andreas.sperneder@abf.co.at

2.4 Geschäftsziel

SARA (Situationsadaptive Routenfindung und 3D-Visualisierung) unterstützt Lenker von Transportfahrzeugen der Intralogistik, die optimale Route zu Lagerplätzen zu finden. Durch den sich daraus ergebenden schonenden Einsatz von Ressourcen wird eine Kosten- und Zeitersparnis erzielt.

2.5 Projektziel

Ziel des Projektes ist die Erweiterung des vorhandenen Systems der Firma ABF um die Berechnung der optimale Routen für Transportfahrzeuge, welche in die bestehende 3D-Visualisierung integriert wird. Dabei werden die jeweiligen Fahrzeugeigenschaften berücksichtigt und die Route auf den Fahrzeugtyp angepasst. Der Algorithmus reagiert adaptiv auf plötzliche Wegänderungen (beispielsweise beim Verlassen der vorgesehene Route durch den Lenker des Fahrzeuges, wenn abgestellte Produkte diese blockieren) und berechnet umgehend einen neuen Weg.

3 Ausgangssituation

Als Grundlage dieser Arbeit dient die bestehende Logistiklösung OneBase-MFT (= Material Flow Tracker) der Firma ABF [abf]. Dieses System aus dem Bereich der Lagerverwaltung unterstützt Lagerprozesse wie Wareneingang, Bestandsführung, Verladung und Versand. Darüber hinaus bietet es Materialverfolgung auf Lagerplatzebene inklusive Fahrzeugortung und 3D-Visualisierung des Lagers, sowie eine vollautomatische Materialflusssteuerung an.

SARA wurde als unabhängiges Modul für das System der Firma ABF zur Berechnung der optimalen Route für Transportfahrzeuge in der Intralogistik entwickelt und integriert diese in die bestehende 3D-Visualisierung. Aufgrund der vorhandenen Lösung OneBase-MFT wurde die Programmiersprache Java verwendet.

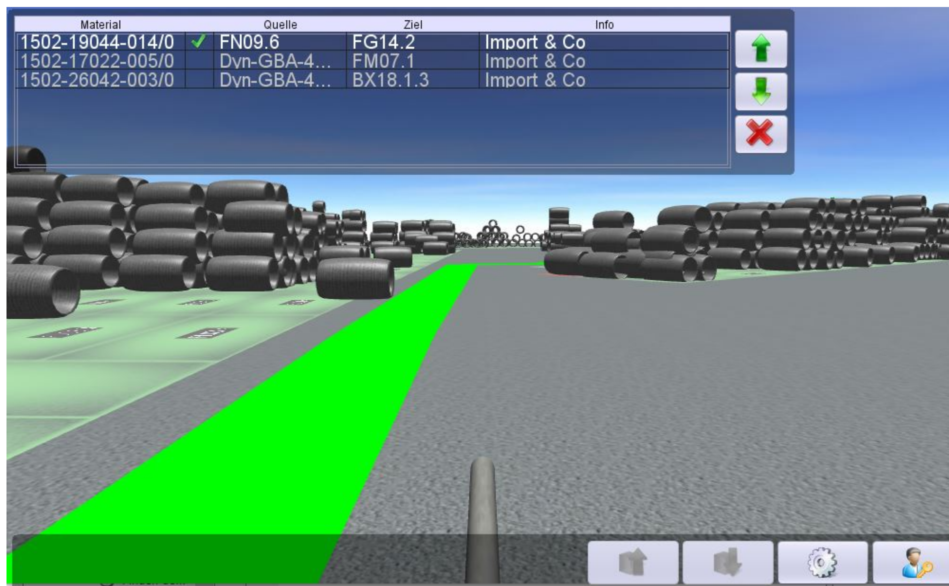

Abbildung 3.1: 3D-Visualisierung der optimalen Route

3.1 Begriffserklärung

3.1.1 Fahraufträge

Zur Ein- oder Umlagerung von Produkten im Lager werden Fahraufträge erstellt. Fahraufträge haben jeweils ein Produkt, welches transportiert werden soll, eine Quelle, wo sich das entsprechende Produkt befindet und ein Ziel. Fahrer von Transportfahrzeugen sehen die Fahraufträge auf ihrem Bildschirm (siehe Abbildung 3.1) und können diese aktivieren beziehungsweise deaktivieren. Die Anzahl, wie viele Fahraufträge gleichzeitig aktiviert werden können, ist nicht begrenzt.

3.1.2 Transport-Objekte

Im System der Firma ABF bilden Transport-Objekte die Basis für alle sich im Lager befindlichen Objekte. Sie haben eine genaue Position im Lager. Die Navigation basiert auf eben diesen Transport-Objekten. Dabei handelt es sich im Normalfall um Transportfahrzeuge beziehungsweise Produkte der Fahraufträge.

3.1.3 OneBase

Als Kommunikationsschicht zwischen Serverprozessen und Visualisierungen kommt OneBase zum Einsatz. Dabei handelt es sich um eine Plattform, die es ermöglicht, dass zwei oder mehrere Prozesse über Datenpunkte kommunizieren (1:n-Kommunikation). Zusätzlich können Datenpunkte in einer Datenbank persistiert werden, so dass der Zustand nach einem Neustart wiederhergestellt werden kann.

3.1.4 OneBase-Tag

Ein OneBase-Tag ist ein Datenpunkt, der von beliebigen Clients abonniert werden kann. Beschreibt ein Client diesen Tag, so erhalten alle anderen Clients eine Notifizierung über die Änderung.

3.2 System OneBase-MFT

OneBase-MFT umfasst die Lagerverwaltung, sowie die kontinuierliche Materialverfolgung und situationsadaptive Materialflusssteuerung für die innerbetrieblichen Logistikprozesse. Aufgebaut ist dieses System serverseitig durch die Dienste *OBSocketServer*, *Synchronizer*, *WarehouseManager* und *Supervisor*. Diese Services stellen das Basissystem dar und werden je nach Bedarf um weitere Services ergänzt.

3.2.1 OBSocketServer

Der SocketServer bildet das gesamte Lagerbild in Form von Tags im Speicher ab. Jeder Client, der sich zu diesem Server verbindet, kann ohne direkten Datenbankzugriff auf die Daten zugreifen und wird mithilfe des OneBase-Tags über Änderungen informiert.

3.2.2 Synchronizer

Dieser Prozess sorgt nach einem Neustart des Systems dafür, dass das Lagerabbild aus der Datenbank in den SocketServer geladen wird.

3.2.3 Warehouse Manager

Der Warehouse Manager ist der zentrale Prozess, der die Daten der Clients verarbeitet (Materialumbuchungen, Verwaltung der Transportaufträge, etc.), in der Datenbank persistiert und die Änderungen über den OBSocketServer an die verbundenen Clients meldet.

3.2.4 Supervisor

Der Supervisor überwacht alle projektspezifischen Prozesse und erlaubt eine gezielte Verwaltung. Beendet sich ein Prozess unerwartet, so wird er automatisch neu gestartet.

3.2.5 MFTAdministrator

Für Verwaltungsaufgaben in OneBase-MFT gibt es den *MFTAdministrator*. Hier können Fahraufträge erstellt und Transportfahrzeugen zugewiesen werden. In Abbildung 3.2 ist zu sehen, wie ein Fahrauftrag angelegt werden kann.

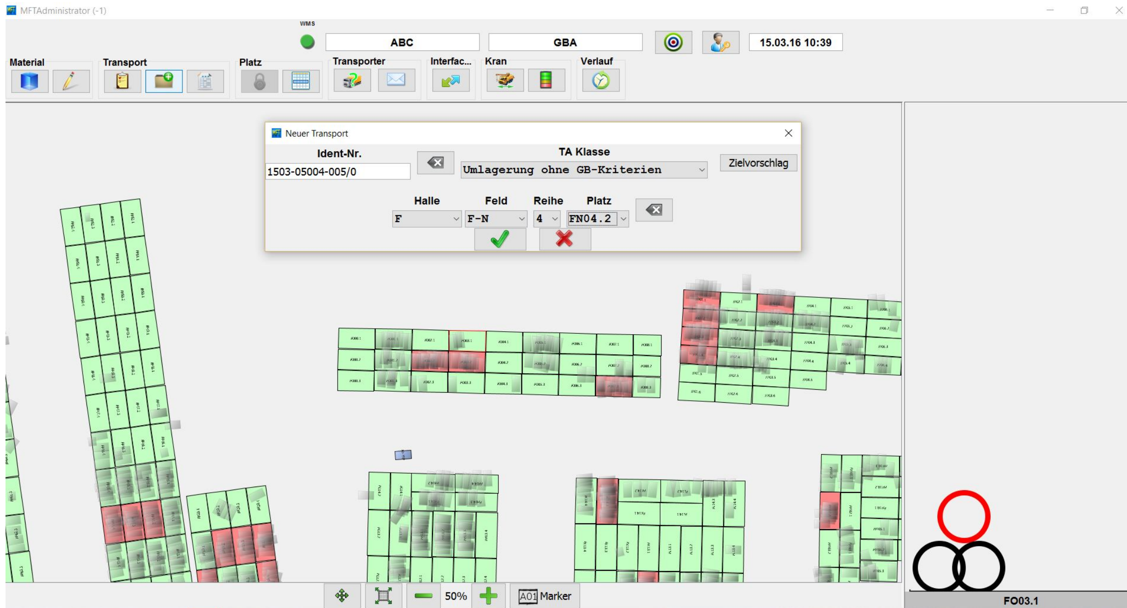

Abbildung 3.2: MFTAdministrator

3.2.6 PositionSimulator

Die aktuelle Position des Transportfahrzeuges wird mithilfe der am Fahrzeug angebrachten Sensoren und Kameras berechnet. (siehe Kapitel 3.3 Fahrzeugortung und -verfolgung). Während der Entwicklungsphase werden diese Daten vom eigens dafür entwickelten *PositionSimulator* simuliert, welcher in Abbildung 3.3 zu sehen ist.

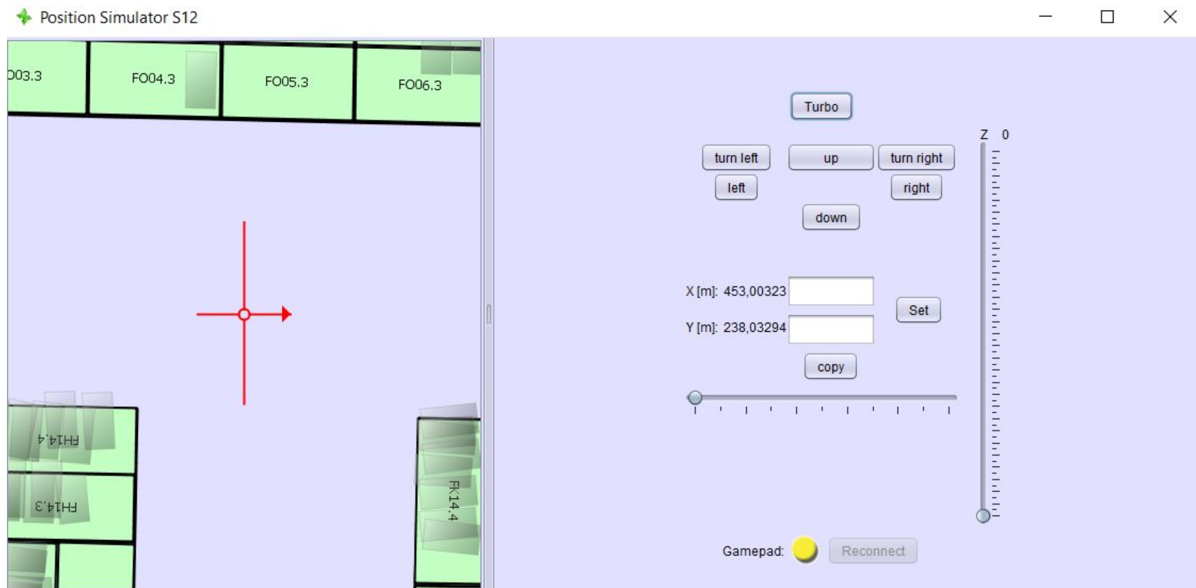

Abbildung 3.3: PositionSimulator

3.3 Fahrzeugortung und -verfolgung

3.3.1 DGPS

Für die Fahrzeugortung und -verfolgung in Freilager-Bereichen verwendet die Firma ABF differential GPS.
Durch die höhere Genauigkeit des DGPS-Systems (Differential Global Positioning System), kann die Position von Fahrzeugen bis auf wenige Zentimeter genau bestimmt werden.

Erreicht wird dies durch die Verwendung von stationären Referenzstationen, deren Koordinaten exakt bekannt sind. Diese berechnet die Differenz zwischen den Sollwerten und den empfangenen Positionen, welche dann über das Netzwerk an die GPS-Empfänger in den einzelnen Fahrzeugen übertragen wird, um sie in der Positionsberechnung zu berücksichtigen. Die Referenzstation erlaubt damit in einem gewissen Umkreis, momentane Störungen und Ungenauigkeiten herauszurechnen. Damit kann die aktuelle Position genauer bestimmt werden.

3.3.2 Zeno Track

Die Fahrzeugverfolgung im Innenbereich basiert auf dem patentierten Zeno Track-Verfahren der Firma Zeno Track GmbH [Gmb]. Bei diesem Verfahren werden zwei sich ergänzende Methoden eingesetzt, um die über eine Kamera (Zeno Cam) aufgenommenen Bilder auszuwerten:

- **Marker-Tracking**
  Markierungen an Boden oder Wänden werden vom System identifiziert und stellen einen Referenzpunkt für die Positionsbestimmung dar.

Abbildung 3.4: Marker-Tracking [Gmb]

- **Feature-Tracking**
  Kontrastreiche Strukturen (z.B. Verschmutzungen, Reifenspuren, Bodenstrukturen) werden genutzt, um die Bewegung des Fahrzeuges zu bestimmen. Dies ist vergleichbar mit dem Prinzip der optischen Maus.

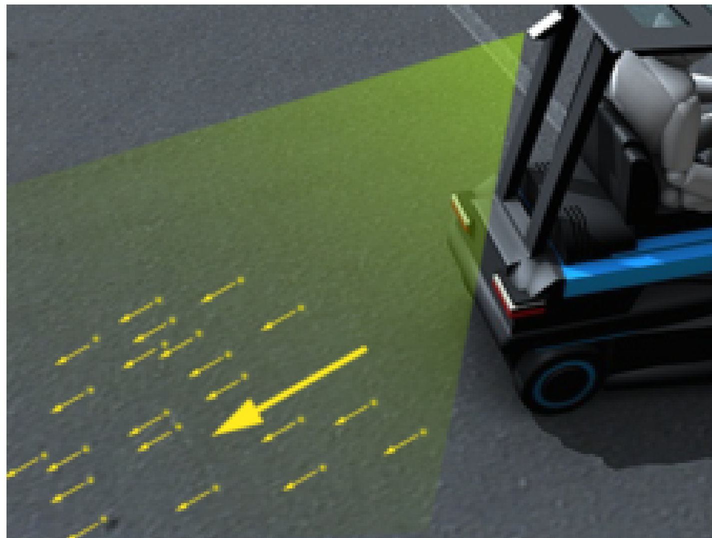

Abbildung 3.5: Feature-Tracking [Gmb]

4 Grundlagen

4.1 Theoretische Grundlagen

4.1.1 Aufbau eines Lagers

Laut der Definition der Firma ABF besteht ein Lager aus Feldern, welche in Reihen unterteilt werden. Diese wiederum bestehen aus einzelnen Lagerplätzen.
Auf den befahrbaren Wegen werden für die Navigation erforderliche Kreuzungspunkte definiert, welche als Wegpunkte bezeichnet werden. Die Wegpunkte sind durch Routen miteinander verbunden, für die bestimmte Attribute wie zum Beispiel die Breite definiert sind.

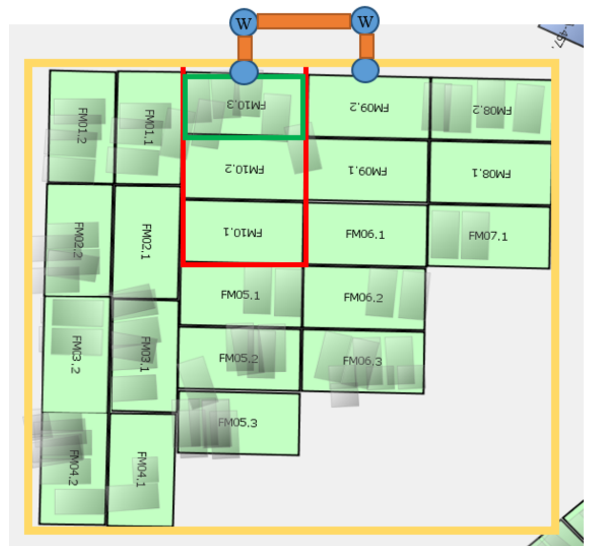

Abbildung 4.1: Aufbau eines Lagers

In Abbildung 4.1 ist ein gelb umrandetes Feld zu sehen. Dieses Feld ist in Reihen unterteilt (rot), welche wiederum in Lagerplätze (grün) unterteilt werden. Der Name eines

Lagerplatzes besteht aus dem Namen des Feldes, der Reihe und des Platzes.
Im Beispiel von Feld „FM10.3“ bezeichnet „FM“ das Feld, „10“ die Reihe und „.3“ die Nummer des Lagerplatzes.
SARA navigiert zu der Reihe des Zielplatzes. Für die Reihen sind jeweils Wegpunkte definiert, welche blau eingezeichnet sind. Ist für eine Reihe kein Wegpunkt definiert, wird der räumlich nächstgelegene Wegpunkt berechnet.

In Abbildung 4.2 ist eine Skizze des für die Entwicklung von SARA eingesetzte Testnetz zu sehen. Es umfasst 157 Wegpunkte und 175 Wege. Daneben befindet sich in Abbildung 4.3 das Wegenetz, welches mithilfe der Visualisierung von JGraphT als Graph dargestellt wurde.

Abbildung 4.2: Aufbau des Wegenetzes

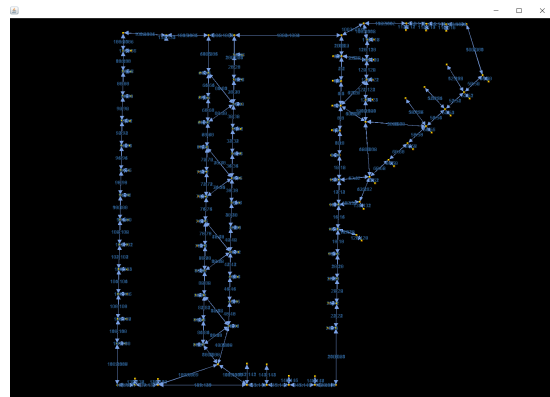

Abbildung 4.3: Wegenetz in JGraphT

4.1.1.1 Wegpunkte

Wegpunkte bestimmen den Start und das Ende einer Route. Sie werden unterteilt in Reihenwegpunkte, welche für eine bestimmte Reihe definiert sind, sowie allgemeine Wegpunkte, welche Kreuzungspunkte zwischen den Feldern markieren.

In der Abbildung 4.1 Aufbau eines Lagers sind die allgemeinen Wegpunkte mit W markiert. Die Reihenwegpunkte liegen in der Abbildung direkt an der Reihe 9 und 10 an.

4.1.2 Graphentheorie

Routenoptimierungssysteme sind ein typisches Anwendungsgebiet der Graphentheorie. Die Grundgesamtheit für die Berechnung der optimalen Route zwischen zwei Orten stellt ein Wegenetz dar, welches aus Routen und Kreuzungspunkten besteht. Diese vernetzten Strukturen können mithilfe von Graphen beschrieben werden.

4.1.2.1 Der Graph

Ein Graph G = (V, E) ist ein Paar von Mengen. Die Elemente der Menge V bezeichnet man als Knoten und repräsentieren die Objekte des Anwendungsbereiches. Bei der Routenoptimierung sind dies die Wegpunkte. Die Elemente der Menge E sind die Kanten des Graphen und stellen die Beziehungen zwischen den Objekten, also die Routen zwischen den Wegpunkten dar.
Dabei gilt:

$$\mathrm{E} \subseteq \binom{V}{2} = \{\{\mathrm{u,v}\} \subseteq \mathrm{V} \mid \mathrm{u} \neq \mathrm{v}\}$$

Die Elemente von E sind also 2-elementige Teilmengen von V. Die beiden Knoten, deren Beziehung durch die Kante dargestellt ist, müssen verschiedene sein, um Zirkulationen zu vermeiden. [uK13]

4.1.2.2 Gewichtung der Kanten

Die Kanten eines Graphen können mit einer Gewichtung versehen werden. Dies sind Zusatzinformationen zu den Beziehungen der Knoten. Bei SARA sind diese Gewichtungen die Priorität der Routen.

Da für die Priorisierung der Routen nicht nur die Länge relevant ist, wird einer Route zusätzlich die durchschnittliche Geschwindigkeit als Attribut zugewiesen. So wird zum Beispiel ermöglicht, dass größere, längere Verbindungswege zwischen den Feldern kleineren Wegen vorgezogen werden. Dadurch wird versichert, dass die Auslastung bei den kleineren Wegen zwischen den Lagerplätzen gering gehalten wird.

Die Priorität der Route in SARA entspricht also dem Zeitaufwand, welcher sich aus der Länge und der durchschnittlichen Geschwindigkeit auf der Route errechnet.

4.1.2.3 Gerichtete Graphen

Bei gerichteten Graphen ist jeder Kante ein Anfangsknoten (A) und ein Endknoten (B) zugeordnet. Diese Kante bezeichnet man als "von A nach B gerichtet." [Esp10] [Die00]

Um die Darstellung von Einbahnstraßen im Lager zu ermöglichen, wird ein gerichteter Graph verwendet. Für Routen ist das Attribut *BIDIRECTIONAL* definiert. Ist dieses Attribut in der Datenbank gesetzt, so wird die Route je einmal als Route von A nach B und als Route von B nach A im Graph dargestellt.

4.1.2.4 Dijkstra-Algorithmus

Für die Berechnung der optimalen Route im Graphen wird der Dijkstra-Algorithmus verwendet. Dieser findet den kürzesten Weg von dem gegebenen Startknoten zum Zielknoten. Er führt dabei eine modifizierte Breitensuche aus.

Voraussetzung für die Verwendung des Algorithmus von Dijkstra ist, dass es keine Kanten mit negativer Gewichtung gibt. Da weder die für die Route definierte Durchschnittsgeschwindigkeit, noch die Länge der Route, aus denen sich die Gewichtung der Kante berechnet, negative Werte annehmen können, ist dies im Weggraphen sichergestellt.

Der Dijkstra-Algorithmus beginnt mit dem gegebenen Startknoten und trägt alle mit diesem Knoten über Kanten in Verbindung stehenden Knoten, mitsamt der Gewichtung im Speicher ein. Wurden alle Kanten durchgegangen, so wird der aktuelle Knoten als "besucht" markiert und mit jenem Knoten, dessen Kante die geringste Gewichtung aufweist, fortgefahren. Der Weg zu den besuchten Knoten ist bereits optimal, da kein kürzerer Weg vom Startknoten zu diesem Knoten existiert.

Im nächsten Schritt werden wiederum alle Kanten durchlaufen, und die berechneten Werte mit den bereits gespeicherten Daten verglichen. Wurde ein kürzerer Weg gefunden, wird der Wert im Speicher durch diesen ersetzt. Dieser Vorgang wird so lange wiederholt, bis der Endknoten besucht wurde. Diese Strategie, dass jeweils mit dem Knoten der geringsten Gewichtung fortgefahren wird, wird als Greedystrategie bezeichnet. Für weitere Informationen über den Dijkstra-Algorithmus, siehe "Vorlesungsskript Graphalgorithmen" der Humboldt-Universität zu Berlin. [uK13]

4.2 Technische Grundlagen

4.2.1 Technologien

4.2.1.1 JGraphT

JGraphT ist eine Open Source Java-Bibliothek und basiert auf der Grundlage der Graphentheorie. Es werden Datenstrukturen und Algorithmen bereitgestellt, um Graphen aufzubauen und Berechnungen durchzuführen. Die Bibliothek ist sehr umfangreich und durch die Verwendung von generischen Datentypen einfach anpassbar.

Der Source Code von JGraphT ist auf GitHub frei zugänglich. [jgrb]
Die aktuelle Version ist auf der Homepage erhältlich. [jgra]

4.2.1.2 JMonkeyEngine

JMonkeyEngine, oder auch kurz JME genannt, ist ein frei zugängliches Java-Framework zur Grafikprogrammierung. Die leistungsstarke, 3D-Szenen-basierte Grafik API (Programmierschnittstelle) wird hauptsächlich zur Programmierung von 3D-Spielen verwendet.

Die Installationsdatei [jmeb] enthält eine erweiterte Version der Entwicklungsumgebung Eclipse, welche als JMonkeyPlatform bezeichnet wird.
Die Firma ABF verwendet die Version JMonkeyEngine 3.0, welche in die bestehende Entwicklungsumgebung Eclipse integriert wird. In der Installationsdatei befinden sich ebenfalls die JAR-Dateien, die im Java Classpath des Projektes platziert werden. Das Framework ist in Java geschrieben und nutzt LWJGL (Lightweight Java Game Library) für den Zugriff auf OpenGL, eine offene Grafikbibliothek. [Bac12]

Für die Entwicklung mit JMonkeyEngine sind folgende Informationen relevant: [jmea]

- **Scene Graph**
  Der *Scene Graph* ist ein virtueller Raum, in dem sich die Objekte der Visualisierung befinden. In diesem Raum werden die Objekte erzeugt, bewegt oder gelöscht. Der Raum wird mithilfe eines dreidimensionalen Koordinatensystems dargestellt und hat X-, Y- und Z-Koordinaten. Dabei ist zu beachten, dass die Y-Koordinate bei JMonkeyEngine die Höhe darstellt.

- **Spatial**
  Die Objekte im *Scene Graph* werden als *Spatial* bezeichnet. *Spatial* bestimmt die Position und Ausrichtung eines Objektes.

- **Geometry**
  Ein *Geometry* ist ein sichtbares Objekte im virtuellen Raum, also ein am Bildschirm angezeigtes *Spatial*. Ein *Geometry* hat eine genau definierte Form.

- **Material**
  Ein *Geometry* beschreibt nur die Form eines Objektes. Um es anzeigen zu können, muss ihm ein *Material* zugewiesen werden. Dieses beschreibt das Aussehen beziehungsweise die Oberfläche der Form.

- **Node**
  Eine *Node*, auch Knoten genannt, ist ein *Spatial*, an dem mehrere *Spatial*-Objekte angehängt werden können. Die Bewegung und Rotation der *Spatial*-Objekte ist relativ zu deren Knoten.

- **RootNode**
  Die *RootNode* ist der Wurzelknoten der Visualisierung. An ihm werden alle anderen *Node*-Objekte angehängt. Am Wurzelknoten ist auch der dreidimensionale *Scene Graph* befestigt.

- **GUINode**
  JMonkeyEngine bietet neben dem dreidimensionalen Raum auch eine zweidimensionale Darstellungsebene an. Damit können Bilder und Texte direkt am Bildschirm angezeigt werden.

- **AssetManager**
  Der *AssetManager* hat die Aufgabe, Multimedia-Dateien wie 3D-Modelle, *Materials* oder Schriftarten zu verwalten. Bei der Initialisierung der Visualisierung werden *RootNode*, *GuiNode*, *Scene Graph* und *AssetManager* erzeugt.

- **SimpleApplication**
  Jedes JME-Programm überschreibt die Methoden *simpleInitApp()*, *simpleUpdate()* und *simpleRender()* der Basisklasse *SimpleApplication*.

  - **simpleInitApp()**
    Die *simpleInitApp()* Methode wird beim Start der Visualisierung aufgerufen. Darin werden Objekte, wie zum Beispiel *Geometry* beziehungsweise *Nodes*, erzeugt.
  - **simpleUpdate()**
    Die *simpleUpdate()* Methode wird zyklisch aufgerufen und hat einen Parameter "tpf" (=time per frame). Die "time per frame", also die Zeit pro Bild, ist abhängig von der Geschwindigkeit des ausführenden Rechners. Bei einem langsamen Rechnern ist der Wert dieses Parameters höher als bei einem schnelleren. Die Variable wird verwendet um beispielsweise die Geschwindigkeit von bewegenden Objekten bei schnelleren oder langsameren Rechnern konstant zu halten.

4.2.2 Entwicklungskonfiguration

4.2.2.1 Server

Von Seiten der HTL Perg wurde ein virtueller Projektserver mit dem Betriebssystem Windows Server 2012 R2 bereitgestellt. Auf diesem Server wurde Oracle XE 11 installiert und mit den erhaltenen Testdaten befüllt. Auf dem Server wurden zudem folgende Services der Firma ABF installiert:

- OBSocketServer
- Synchronizer
- Supervisor
- WarehouseManager

Dazu mussten die von der Firma ABF erhaltenen Dateien in *C:/ABF* kopiert werden und die in Ausschnitt 4.1 zu sehenden Batch-Dateien ausgeführt werden. Für Informationen über die Services siehe Kapitel 3.2 System OneBase-MFT.

```
1 /OBSocketServer/wrapper/bin/InstallOBSocketServerMFT.bat
2 /Services/InstallSynchronizer.bat
3 /Services/WarehouseManager.bat
4 /Supervisor/Supervisor/installService.bat
```

Ausschnitt 4.1: Installation der Services für OneBase-MFT

4.2.2.2 Client

Zur Erweiterung des Systems wurde von der Firma ABF der Source Code von OneBase-MFT (Entwicklungsstand 17.7.2015) übergeben. Der gesamte Eclipse Workspace wurde auf die Entwicklungsnotebooks kopiert.
Um die Staplervisualisierung ausführen zu können, muss am lokalen Rechner der OB-SocketServer über die Batch-Datei **startSocketServer XE LOCAL.bat** gestartet werden. Der SocketServer wird über die Datei **OBSocketServerTransportationXELocal.conf** konfiguriert. Hier müssen der Datenbank-User, das Passwort und der Connection-String der Datenbank angepasst werden:

```
####################################################
# Connect Strings
####################################################
DBUser=mft_sara
DBPassword=sara
DBConnectString=jdbc:oracle:thin:@10.114.57.44:1521:XE

LoadPropertiesFromDB=OB,OBLocal
ServerTopic=TRANS_LOCAL

####################################################
# OB SocketServer
####################################################
ServerSystimWriter=true
...
```

Ausschnitt 4.2: Konfiguration des OBSocketServers

In der jeweiligen Property-Datei (**application.properties**) der einzelnen Projekte müssen ebenfalls der Datenbank-User, das Passwort und der Connection-String (wie in Ausschnitt 4.3 zu sehen) angepasst werden.

```
#*******************************
# DB Connection
#*******************************
DBUser=mft_sara
DBPassword=sara
DBConnectString=jdbc:oracle:thin:@10.114.57.44:1521:XE

transportation.centeredObject=S17
transportation.visu.alwaysOnTop=false
transportation.mode.ignoreMissingPLC=true
ShowInstallButton=0
...
```

Ausschnitt 4.3: Application Properties

5 Benutzung

5.1 Use Case Diagramm

Abbildung 5.1: Use Case Diagramm

User

- **Route einsehen**
  Der Benutzer hat die Möglichkeit, nach Aktivierung des Systems und Zuweisung eines Auftrages, die Navigation zum Zielpunkt einzusehen.

- **Navigation starten/stoppen**
  Falls sich der Benutzer entschließt, die Navigation beenden zu wollen, hat er die Möglichkeit, dies über den Menüpunkt „Navigation deaktivieren“ zu tätigen. Des Weiteren ist es möglich, durch einen erneuten Klick auf den Menüpunkt „Navigation aktivieren“ die Navigation wieder zu starten.

System

- **Navigation starten**
  Die Navigation wird mit dem Aktivieren eines Fahrauftrages gestartet. Die optimale Route zur Abarbeitung der Fahraufträge wird in der 3D-Visualisierung angezeigt. Dadurch ist es für den Fahrer des Transportfahrzeuges möglich, den optimalen Weg zum Produkt bzw. zum Zielplatz zu finden.

- **Navigation stoppen**
  Wenn ein Auftrag mit dem Erreichen des endgültigen Zielplatzes abgeschlossen wurde oder der Auftrag durch das System abgebrochen wurde, wird die Navigation beendet.

- **Route berechnen**
  Nachdem der Graph vor Beginn des Navigationsstartes aufgebaut wurde, wird der schnellste Weg durch den Dijkstra Algorithmus berechnet. Bei einer Abweichung von der Route wird die Navigation neu gestartet und die optimale Route dadurch neu berechnet.

- **SARA-Reinitialisieren**
  Bei Änderungen in der Datenbank (z.B. Sperren von Routen, Bearbeiten der Attribute einer Route) wird die Navigation gestoppt. Anschließend werden alle Daten aus der Datenbank neu geladen und eine Neuberechnung der Route erzwungen.

5.2 Ablaufdiagramme

5.2.1 Navigation

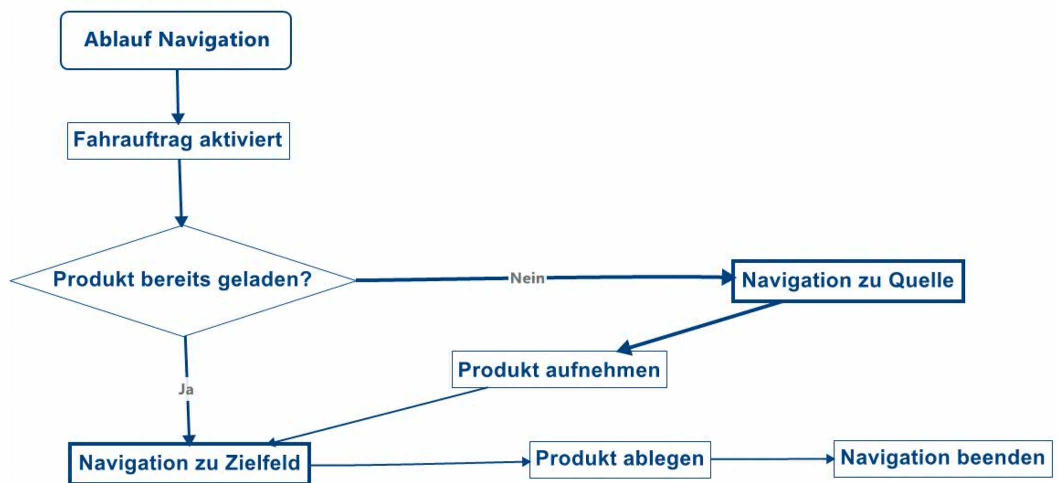

Abbildung 5.2: Ablauf Navigation

Sobald ein Fahrauftrag aktiviert wird, wird die Navigation gestartet. Dies geschieht entweder durch die Aufnahme eines Produktes, für welches ein Fahrauftrag besteht, oder durch Aktivierung über das graphische Interface.
Nach dem Start wird überprüft, ob das entsprechende Produkt des Fahrauftrages bereits auf dem Fahrzeug geladen ist. Ist dies der Fall, so wird zum Zielplatz des Fahrauftrages navigiert. Ist das Produkt noch nicht geladen, wird die optimale Route zur Quelle (= Lagerplatz, auf dem sich das Produkt befindet) berechnet.
Die Navigation wird beendet, sobald das Produkt wieder abgelegt wurde. Befindet sich das Produkt auf dem richtigen Zielplatz, wird der Status des Fahrauftrages vom System der Firma ABF auf "abgeschlossen" gesetzt. Wird das Produkt an einer anderen Stelle abgelegt, wird der Fahrauftrag pausiert und nach der erneuten Aufnahme wieder aktiviert.

5.2.2 Mehrere Fahraufträge

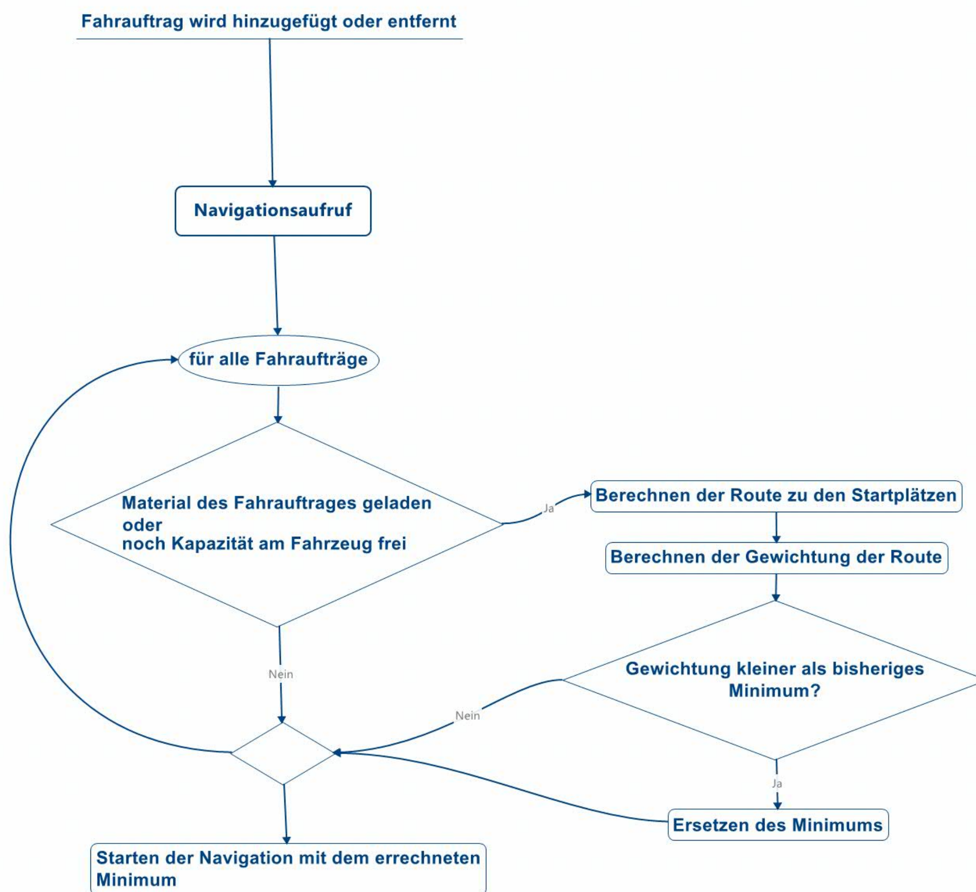

Abbildung 5.3: Ablauf Navigation - mehrere Fahraufträge

Es können mehrere Fahraufträge gleichzeitig aktiv sein. Um in diesem Fall die optimale Route zur Abarbeitung der Fahraufträge berechnen zu können, wurde der Algorithmus erweitert:
Es werden alle aktivierten Fahraufträge linear durchlaufen. Wenn sich das Produkt eines Auftrages bereits auf dem Transportfahrzeug befindet, wird die Gewichtung der Route zum Zielplatz berechnet. Ist das Produkt des Fahrauftrages nicht geladen, wird überprüft, ob auf dem Transportfahrzeug noch genügend Kapazität zur Aufnahme des Produktes vorhanden ist. Ist dies der Fall, wird die Gewichtung der Route zum Produkt

berechnet. (Details zur Umsetzung siehe Kapitel 6.2.4.1 Kapazität von Transportfahrzeugen)
Anschließend wird die Navigation mit dem jeweiligen Start- bzw. Zielpunkt gestartet, für dessen Route die niedrigste Gewichtung errechnet wurde.
Wird ein Fahrauftrag aktiviert beziehungsweise deaktiviert, wird dieser in die Liste der Fahraufträge hinzugefügt oder entfernt und die Navigation neu gestartet.

5.2.3 Routenfindung

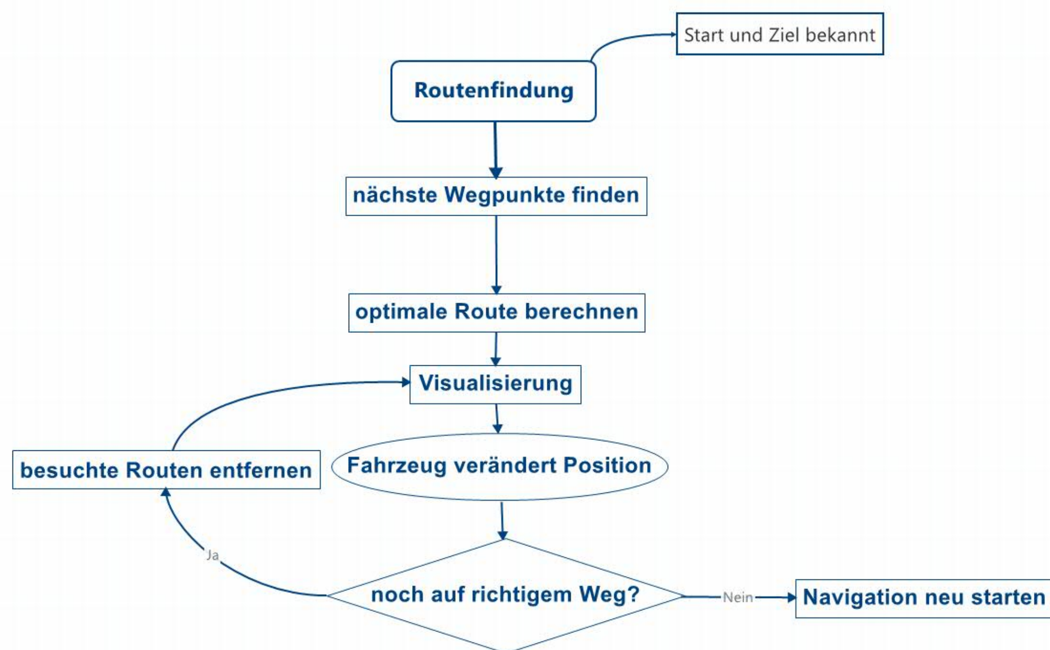

Abbildung 5.4: Ablauf Routenfindung

Beim Aufruf der Routenfindung sind Start und Ziel der Navigation bekannt. Der Startpunkt entspricht zumeist dem Transportfahrzeug, allerdings kann für die Berechnung ein beliebiges Transport-Objekt übergeben werden.
Zu Beginn werden die Wegpunkte berechnet. Handelt es sich beim Start beziehungsweise Ziel um einen Lagerplatz, so wird nach dem für diesen Platz definierten Wegpunkt gesucht. Wird dieser nicht gefunden, oder handelt es sich bei dem Punkt um ein Transportfahrzeug, so wird der von der distanzmäßig nächstgelegene Wegpunkt für die

Berechnung herangezogen. Gleiches geschieht auch, wenn für einen Lagerplatz mehrere Wegpunkte definiert sind.
Die Wegpunkte stellen den Start- beziehungsweise Endknoten für die Suche im Graphen dar. Mithilfe des Graphenalgorithmus von Dijkstra wird die Route mit der geringsten Gewichtung berechnet und das Ergebnis anschließend in der 3D-Visualisierung integriert. Bei einer Positionsveränderung wird überprüft, ob sich das Transportfahrzeug noch auf der berechneten, optimalen Route befindet. Ist dies der Fall, werden die bereits besuchten Teilstücke aus der Route entfernt und ausgeblendet. Verlässt das Transportfahrzeug die Route, wird die Navigation neu gestartet, um so die aktuelle optimale Route neu zu berechnen.

6 Programmbeschreibung

6.1 Datenmodell

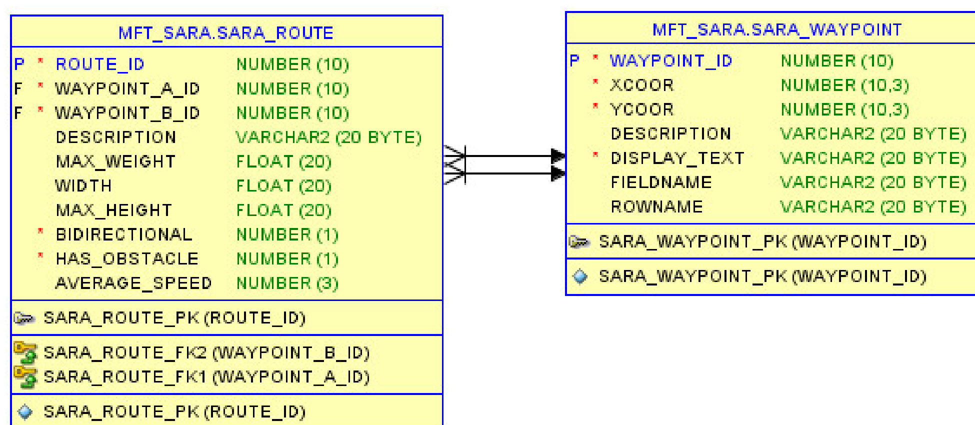

Abbildung 6.1: ERD der SARA Tabellen

Zur Realisierung von SARA wurde die Datenbank der Firma ABF um 2 Tabellen erweitert, welche durch den Präfix "SARA_" zu erkennen sind. Wegpunkte, welche über die Fremdschlüssel *Waypoint_A_ID* sowie *Waypoint_B_ID* referenziert werden, bestimmen die Länge und die Lage der Route in einem Lager. Die einzelnen Attribute der Tabelle *SARA_ROUTE* und *SARA_WAYPOINT* werden im folgendem Datenlexikon näher erklärt.

6.1.1 Datenlexikon

6.1.1.1 Wegpunkt

Nr.	Parametername	Parameterbeschreibung	Datentyp	Länge	Einheit	Default	Constraints
Tabelle Waypoint							
	WAYPOINT_ID	Primary Key	number	10			PK
	XCOOR	X-Koordinate	number	10,3	m		
	YCOOR	Y-Koordinate	number	10,3	m		
	DESCRIPTION	Für zusätzliche Informationen	varchar	20			
	DISPLAY_TEXT	Name des Wegpunktes	varchar	20			
	FIELDNAME	Identifikator für das Feld	varchar	20		Null	
	ROWNAME	Identifikator für die Reihe	varchar	20		Null	

Abbildung 6.2: Datenlexikon Wegpunkt

- **XCOOR**
  X-Koordinaten des Wegpunktes
- **YCOOR**
  Y-Koordinaten des Wegpunktes
- **DESCRIPTION**
  Optionale Informationen und Anmerkungen.
- **DISPLAY_TEXT**
  In diesem Attribut wird der vollständige Name gespeichert. Wegpunkte, die direkt an Lagerplätzen anliegen (=Reihenwegpunkte), werden nach dem Namen der Reihe und des Feldes benannt.
  Beispiel: Wegpunkt für Lagerplatz "FL" in Reihe "14": "FL14"
- **FIELDNAME**
  Bei Reihenwegpunkten wird in dieser Spalte der Name des Feldes eingetragen. Handelt es sich bei diesem Wegpunkt nicht um einen Reihenpunkt, ist dieses Feld nicht auszufüllen.
- **ROWNAME**
  Bei Reihenwegpunkten wird in dieser Spalte der Reihenname eingetragen. Bei Wegpunkten ohne dazugehörigen Lagerplatz bleibt diese Spalte leer.

6.1.1.2 Route

Nr.	Parametername	Parameterbeschreibung	Datentyp	Länge	Einheit	Default	Constraints
Tabelle Route							
	ROUTE_ID	Primary Key	number	10			PK
	WAYPOINT_A_ID	Wegpunkt A (Start)	number	10			FK
	WAYPOINT_B_ID	Wegpunkt B	number	10			FK
	HAS_OBSTACLE	ist die Route frei oder gesperrt? (has obstacle = Route blockiert)	number	1		1	
	DESCRIPTION	Für zusätzliche Informationen	varchar	20			
	MAX_WEIGHT	Maximal zulässiges Gewicht	float	20	kg	Null	
	WIDTH	Breite der Fahrbahn	float	20	m	Null	
	MAX_HEIGHT	Maximal zulässige Höhe	float	20	m	Null	
	ROUTE_AVERAGE_SPEED	Durchschnittliche Geschwindigkeit mit der die Route befahren werden darf	number	3	km/h		
	BIDIRECTIONAL	Route ist in beide Richtungen befahrbar	number	1		0	

Abbildung 6.3: Datenlexikon Route

- **WAYPOINT_A_ID**
  Definiert den Startpunkt einer Route
- **WAYPOINT_B_ID**
  Definiert den Endpunkt einer Route
- **HAS_OBSTACLE**
  Ist dieses Attribut auf „1“ gesetzt, wird die Route gesperrt. Das heißt, dass die Route momentan nicht befahren werden kann.
- **DESCRIPTION**
  Optionale Informationen und Anmerkungen.
- **MAX_WEIGHT**
  Gibt das maximal zulässige Gesamtgewicht eines Transportfahrzeuges an, mit welchem die Route befahren werden darf.
- **WIDTH**
  Dieses Attribut bezeichnet die maximale Breite eines Transportfahrzeuges, mit welchem die Route befahren werden darf.
- **MAX_HEIGHT**
  Dieses Attribut bezeichnet die maximale Höhe eines Transportfahrzeuges, mit welchem die Route befahren werden darf.

- **ROUTE_AVERAGE_SPEED**
  Dieses Attribut zeigt an, welche durchschnittliche Geschwindigkeit auf der Route gefahren wird. Diese Geschwindigkeit dient zur Berechnung der Gewichtung der Routen.

  Wird in „km/h“ angegeben.

- **BIDIRECTIONAL**
  Ist dieses Attribut auf „1“ gesetzt, kann die Route in beide Richtungen befahren werden. Bei dem Wert „0“ ist die Route nur in Richtung *Waypoint_A* nach *Waypoint_B* befahrbar.

Derzeit gibt es kein Programm zur Erstellung von Routen und Wegpunkten. Daher werden die Daten vorerst direkt in die Datenbank eingetragen.

6.1.2 Properties

Für die Verwaltung bestimmter Einstellungen bietet OneBase einen Lösungsansatz zur zentralen Speicherung und standardisiertem Zugriff. Dieser Lösungsansatz wird auch in SARA verwendet.
Die Daten werden dazu in der Tabelle OB_Properties gespeichert. Diese Properties bestehen jeweils aus einem Namen und einem Wert. Für die Zuordnung zu den einzelnen Teilen des Systems verfügt die Tabelle über eine Spalte "Client". Die Properties von SARA werden dem Client "SARA"zugeordnet.

	PROPERTY	CLIENT	VALUE
1	carrier.default.capacity	SARA	2
2	db.tagname	SARA	RELOAD_ROUTES
3	navigation.max.route.width	SARA	5
4	visu.route.color	SARA	0,255,0
5	visu.route.width	SARA	1

Abbildung 6.4: SARA-Properties

Von SARA werden folgende Properties in der Tabelle OB_Properties gespeichert:

- **Standardbreite von Routen - navigation.max.route.width**
  Die Standardbreite der einzelnen Routen in Meter. Dieser Wert wird für die Toleranz bei der Abweichung von Routen verwendet, falls in der Datenbank keine Angaben zur Breite vorhanden sind.

- **Kapazität der Transportfahrzeuge - carrier.default.capacity**
  Dieser Wert gibt an, welche Anzahl an Produkten ein Transportfahrzeug standardmäßig gleichzeitig aufnehmen kann.

- **Farbe der Route - visu.route.color**
  Farbe der Route in der Visualisierung im RGB-Format getrennt durch Beistriche. Beispiel: (0,255,0) $\rightarrow$ Grün

- **Breite der Route - visu.route.width**
  Breite der Route in der Visualisierung angegeben in Meter. Die Breite entspricht laut JMonkeyEngine dem Abstand vom Mittelpunkt der Route, daher ist die sichtbare Route in der Visualisierung doppelt so breit wie der angegebene Wert.

- **Navigation aktiviert - navigation.enabled**
  Dieser Wert ist entweder "true "oder "false "und gibt an, ob die Navigation derzeit aktiviert oder deaktiviert ist. Da der Wert für jeden Client einzeln festgelegt werden muss, wird die Einstellung über die Spalte "Client "zugeordnet.

- **Tagname - db.tagname**
  In "Tagname "wird der Name des OBTags für die Daten von SARA gespeichert. Für nähere Informationen siehe 6.2.2.4 Erkennung von Veränderungen in der Datenbank.

Für Informationen über den Zugriff auf die Properties siehe Kapitel 6.2.1.1.

6.2 Implementierung

6.2.1 SARA-Manager

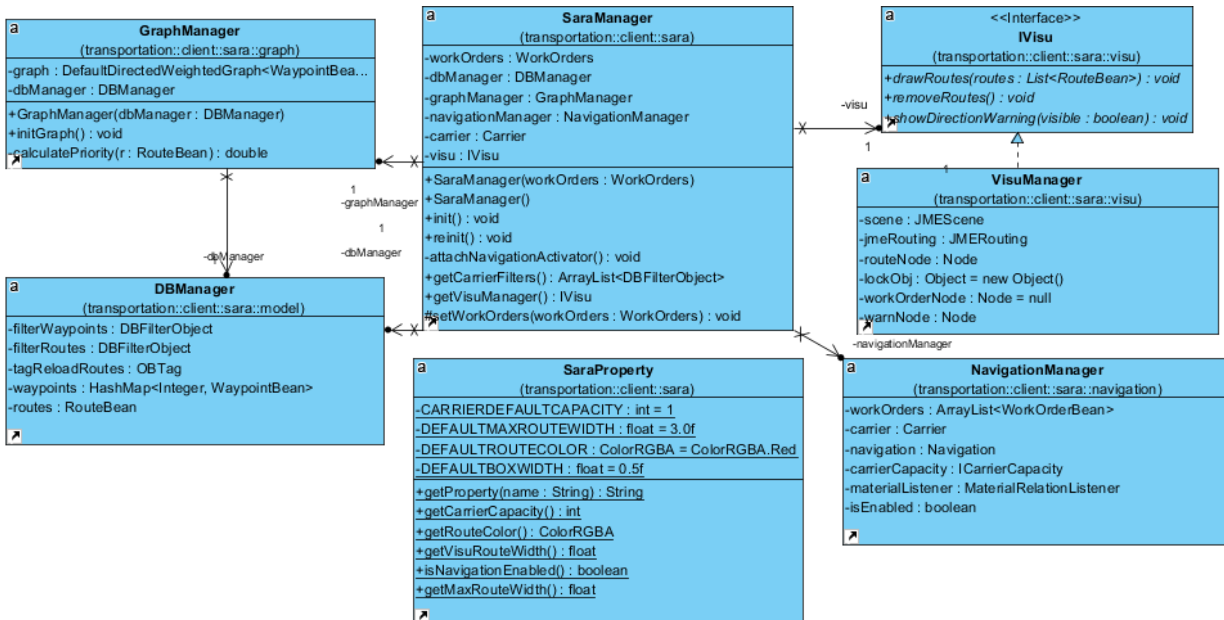

Abbildung 6.5: UML SARA Hauptklassen

Der *SaraManager* bildet die zentrale Klasse des Moduls SARA. Bei der Initialisierung des *SaraManagers* werden mithilfe des *DBManagers* die Daten aus der Datenbank geladen. Der *GraphManager* wird aufgerufen, um den Graph zu initialisieren. Danach wird der *NavigationManager* initialisiert und der *NavigationActivationListener* an das bei der Initialisierung des *SaraManagers* übergebene *WorkOrders*-Objekt angehängt.

Bei Änderung der Wegpunkte oder Routen in der Datenbank müssen die Daten neu geladen werden und der Graph neu aufgebaut werden. (siehe Kapitel 6.2.2.4 Erkennung von Veränderungen in der Datenbank). Laufende Navigationen müssen dabei während dieses Vorganges pausiert werden.

Der Zugriff auf den SaraManager wird mithilfe des Singleton Design Patterns realisiert, um sicher zu stellen, dass es im Programm genau ein *SaraManager*-Objekt gibt. Die Implementierung des Design Patterns ist in Ausschnitt 6.1 zu sehen.

```
public class Sara {
  private static SaraManager INSTANCE = null;

  public static SaraManager getInstance() {
    if(INSTANCE == null){
      INSTANCE = new SaraManager();
      INSTANCE.init();
    }
    return INSTANCE;
  }
}
```

Ausschnitt 6.1: Singleton Design Pattern

Dabei ist zu beachten, dass das *SaraManager*-Objekt der Variable *INSTANCE* zugeordnet wird, bevor die Initialisierungsmethode aufgerufen wird. Der Grund dafür ist, dass in der Methode *init()* bei der Initialisierung der einzelnen Komponenten bereits auf das *SaraManager*-Objekt über *getInstance()* zugegriffen wird.

6.2.1.1 SARA-Properties

Für den Zugriff auf die Properties von SARA (siehe Kapitel 6.1.2 Properties) wurde die Enumeration *SaraPropertyType* entwickelt. Sie umfasst folgende Properties:

```
  VISU_ROUTE_COLOR("visu.route.color"),
  VISU_ROUTE_WIDTH("visu.route.width"),
  CARRIER_DEFAULT_CAPACITY("carrier.default.capacity"),
  MAX_ROUTE_WIDTH("navigation.max.route.width"),
  TAG_RELOAD_ROUTES("db.tagname"),
  NAVIGATION_ENABLED("navigation.enabled");
```

Ausschnitt 6.2: SaraPropertyType

Der Zugriff wird über die Klasse *SaraProperty* abgewickelt. Diese bietet die statische Methode *getProperty()*, welche über die Klasse *ABFProperties* auf die Properties zugreift.

```
1 public static String getProperty(SaraPropertyType property){
2   return ABFProperties.getLastProperties()
3             .getProperty(property.toString());
4 }
```

Ausschnitt 6.3: Zugriff auf SARA Properteis

6.2.2 Model

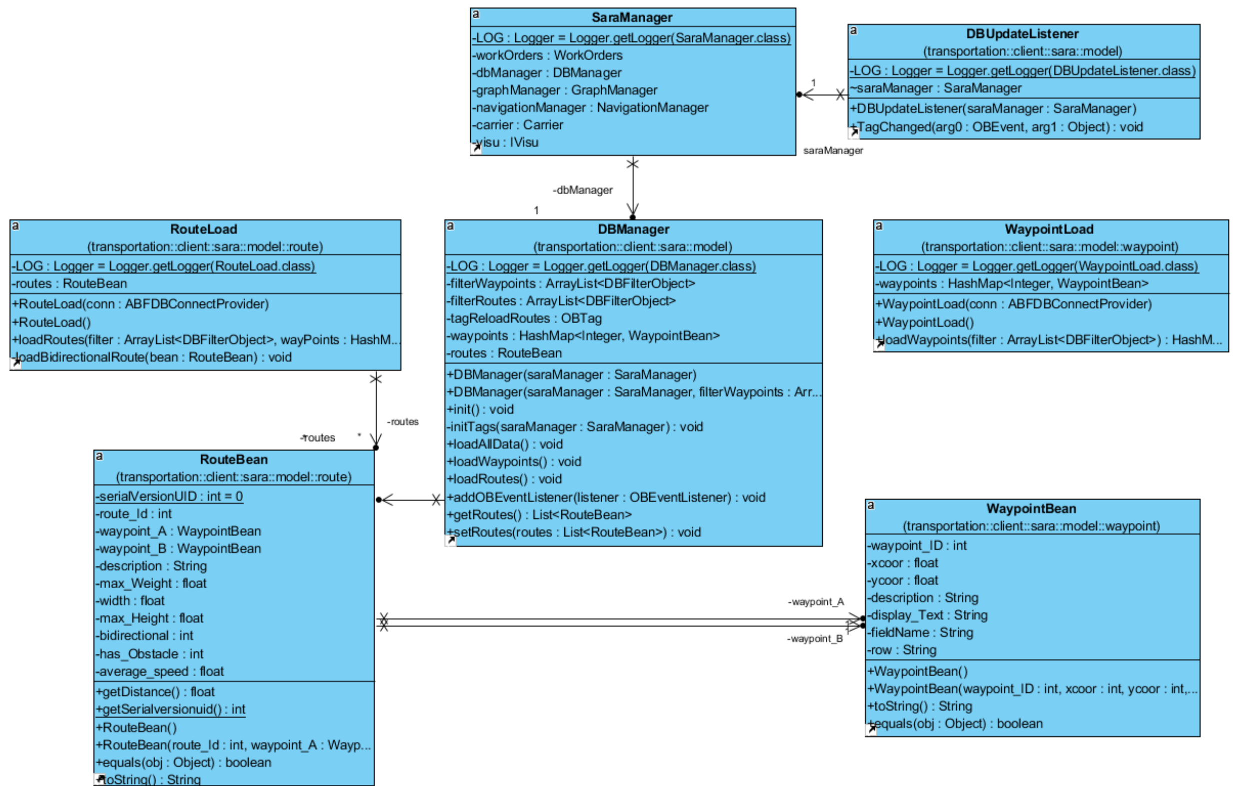

Abbildung 6.6: UML Model

Der *DBManager* verwaltet die Daten, welche für die Navigation benötigt werden. Im Konstruktor der Klasse wird die Methode *initTags()* aufgerufen, welche ein OneBase-Tag (siehe Kapitel 3.1.4 OneBase-Tag) erzeugt. Ändert sich der Wert des Tags, wird der *DBUpdateListener* aufgerufen, die Daten neu zu laden. (6.2.2.4 Erkennung von Veränderungen in der Datenbank).
Mithilfe der Methoden *loadWaypoints()* und *loadRoutes()* können die Routen beziehungsweise Wegpunkte geladen werden.

```java
private void initTags(SaraManager saraManager) {
  if(tagReloadRoutes == null){
    tagReloadRoutes = OBTag.createTag(ABFProperties
      .getLastProperties().getProperty("ServerTopic"),
        WMTagName.TAG_RELOAD_ROUTES);

    tagReloadRoutes.initTag();
    tagReloadRoutes.addOBEventListener(new
      DBUpdateListener(saraManager));
  }
}
```

Ausschnitt 6.4: Erzeugung des OneBase-Tags

6.2.2.1 Laden von Datensätzen

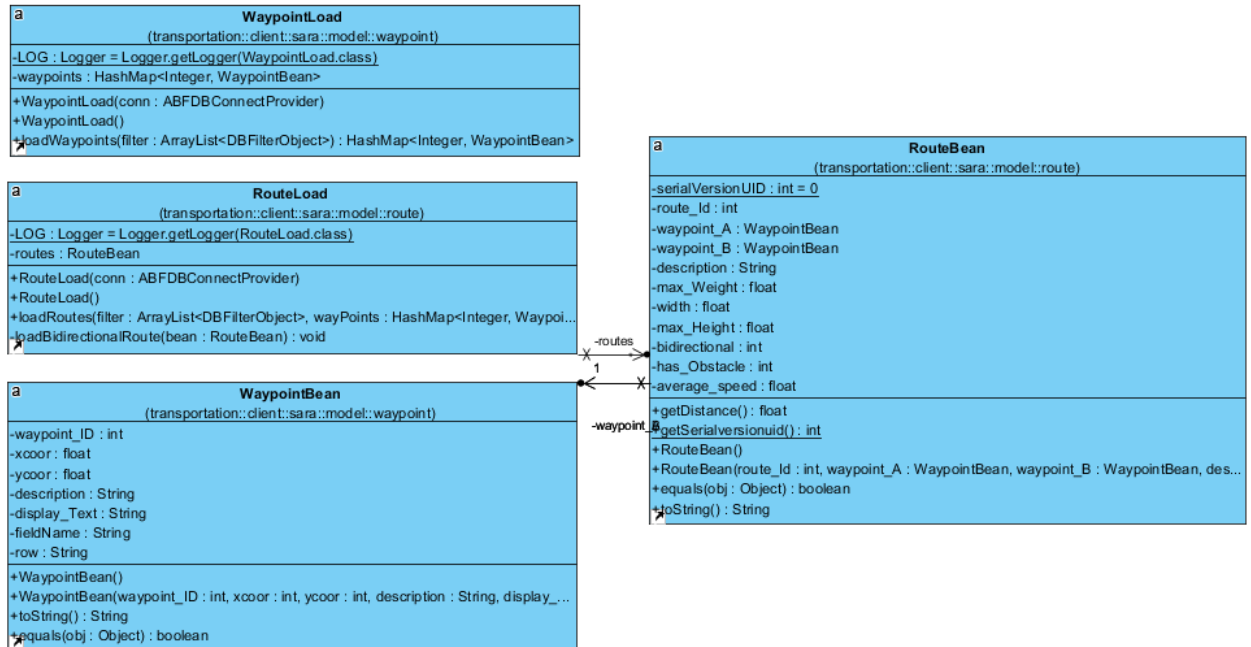

Abbildung 6.7: UML Datenbankschnittstelle

Die Klassen *LoadWayoints* und *LoadRoutes* sind für das Laden der Datensätze zuständig. Vom SQL-Statement wird ein *ResultSet* zurückgegeben. Die Werte aus dem *ResultSet* werden in *WaypointBean*- und *RouteBean*-Objekte gespeichert.

6.2.2.2 Filterung der Datensätze

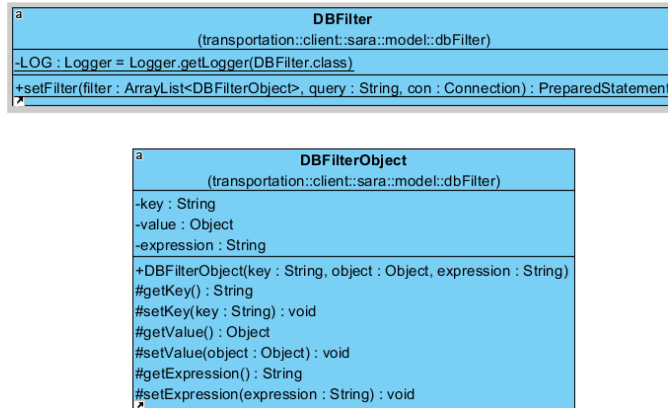

Abbildung 6.8: UML Datenbankfilter

DBFilter verwendet Filterobjekte, um aufgrund der jeweiligen Attribute des Transportfahrzeuges die entsprechenden Where-Klauseln zu erzeugen. Die Klasse *DBFilterObject* ist in Ausschnitt 6.5 zu sehen.
Key repräsentiert dabei den Spaltenname in der Tabelle, wie beispielsweise *WIDTH* für die Breite der Route. Die Variable *expression* enthält den jeweiligen Vergleichsoperator (<, >, <=, ...), und die Variable *value* repräsentiert den Operanden der Vergleichsoperation.

```
1  public class DBFilterObject {
2    private String key;
3    private Object value;
4    private String expression;
5    public DBFilterObject(String key, Object value,
6        String expression) {
7      this.key = key;
8      this.value = value;
9      this.expression = expression;
10   }
```

Ausschnitt 6.5: Konstruktor der Klasse DBFilterObject

Um einem SQL-Statement eine Where-Klausel (= Filterung) anzuhängen, wird die *setFilter( )*-Methode des *DBFilters* aufgerufen, welche in Ausschnitt 6.6 zu sehen ist. Diese hat als Parameter eine Liste von Filterobjekten, sowie das jeweilige SQL-Statement als String und die Datenbankverbindung.

```
1  public class DBFilter {
2
3  private static final Logger LOG = Logger
4    .getLogger(DBFilter.class);
5
6  public static PreparedStatement setFilter(
7    ArrayList<DBFilterObject> filter,
8      String query, Connection con){
```

Ausschnitt 6.6: setFilter( )-Methode in DBFilter

Die Liste wird durchgelaufen und für jedes Filterobjekt wird eine Where-Klausel angefügt.
Beispiel: **Breite < ?**

```
16 for(int i=0;i<filter.size();i++){
17   clause.append("(e."+filter.get(i).getKey()+
18     filter.get(i).getExpression()+"? or e."+
19       filter.get(i).getKey() +" is NULL)");
20
21   if(i<filter.size()-1){
22     clause.append(" and ");
23   }
24 }
25   query += clause;
26   stmt = con.prepareStatement(query);
```

Ausschnitt 6.7: Erzeugen eines PreparedStatements

Anschließend wird aus dem SQL-Statement, welches der Methode übergeben wurde, und der erstellten Where-Klausel, ein Prepared-Statement erzeugt.
Prepared-Statements haben den Vorteil, dass sie bei mehrmaligem Aufrufen die Ausführungszeit reduzieren. Diese Eigenschaften haben jene Art von SQL-Statements dadurch, weil sie nicht wie normale Statements zur Datenbank gesendet werden und dort kompiliert werden, sondern schon vorkompiliert und von der Datenbank gespeichert werden. Bei erneuten Senden des SQL-Statements muss bei einem Prepared-Statement die Anfrage nicht mehr kompiliert werden, sondern kann sofort ausgeführt werden. [pre]

Im letzten Schritt ersetzt die Methode die ? durch die *value*-Werte, welche in den Filterobjekten gespeichert sind. Da zur Laufzeit noch nicht bekannt ist, welchen Datentypen die *value*-Variable hat, muss dieser mittels der Abfrage des dynamischen Datentyps bestimmt werden.

```
24 if(filter.get(i).getValue().getClass()
25     .equals(Integer.class)){
26   stmt.setInt(i+1,(int) filter.get(i).getValue());
27 }
28 if(filter.get(i).getValue().getClass()
29     .equals(String.class)){
30   stmt.setString(i+1,(String) filter.get(i).getValue());
31 }
32 if(filter.get(i).getValue().getClass()
33     .equals(Double.class)){
34   stmt.setDouble(i+1,(double) filter.get(i).getValue());
35 }
36 if(filter.get(i).getValue().getClass()
37     .equals(java.sql.Date.class)){
38   stmt.setDate(i+1,(Date) filter.get(i).getValue());
39 }
40 if(filter.get(i).getValue().getClass()
41     .equals(Float.class)){
42   stmt.setFloat(i+1,(float) filter.get(i).getValue());
43 }
```

Ausschnitt 6.8: Abfrage des Objektdatentyps

6.2.2.3 Wegpunktsuche

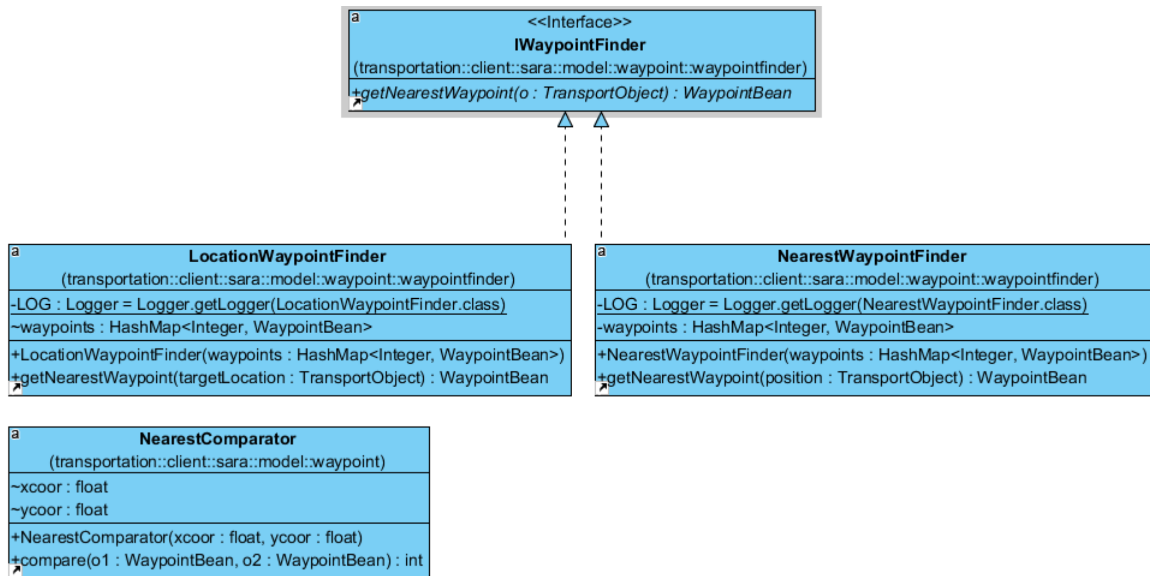

Abbildung 6.9: UML Wegpunktsuche

Zum Finden eines Wegpunktes definiert SARA ein Interface IWaypointFinder:

```
1 public interface IWaypointFinder {
2
3   public WaypointBean getNearestWaypoint(TransportObject o);
4 }
```

Ausschnitt 6.9: IWaypointFinder

Die Klassen *NearestWaypointFinder* und *LocationWaypointFinder* implementieren zwei verschiedene Arten der Suche nach Wegpunkten.

- **NearestWaypointFinder**
  Bei der Instanzierung eines NearesWaypointFinders verlangt der Konstruktor eine Liste von Wegpunkten. Beim Aufruf von getNearesWaypoint(...) holt sich die Methode im ersten Schritt die X- und Y-Koordinaten des Transport-Objektes.

  Im zweiten Schritt erzeugt die Methode einen *NearestWaypointComperator* und übergibt die Koordinaten.

  Da *NearestWaypointComperator* von der generischen Klasse *Comparator* erbt, muss dieser auch die Methode *compare(...)* implementieren.

  Mithilfe des Satzes von Pythagoras errechnet sich *compare(...)* die räumliche Distanz der beiden Wegpunkte zum Transport-Objekt und vergleicht diese.

  Durch die Implementierung der Vergleichsfunktion *compare(...)* ist es nun möglich, die Methode *Collection.min(...)* aufzurufen, welche die errechneten Distanzen miteinander vergleicht und das Minimum zurückgibt.

- **LocationWaypointFinder**
  Anders als beim *NearestWaypointFinder* versucht *LocationWaypointFinder* den für die jeweilige Reihe definierten Wegpunkt zu finden. *LocationWaypointFinder* sucht dabei den Reihenwegpunkt über die Attribute Feldname und Reihenname.

  Ist für die Reihe kein Wegpunkt definiert worden, greift SARA auf die Funktionalität des *NearestWaypointFinder* zurück.

6.2.2.4 Erkennung von Veränderungen in der Datenbank

Dem OneBase-Tag *tagReloadRoutes*, welches bei der Initialisierung des *DBManager* erzeugt wird, wird der *DBUpadeteListener* hinzugefügt. Dieser Implementiert das Interface *OBEventListener*. Dadurch wird bei Veränderung des Tags die Methode *TagChanged* aufgerufen.
Falls der Wert des Tag auf 1 gesetzt wurde, werden die Wegpunkte und Routen aus der Datenbank geladen und der Graph wird neu aufgebaut.

```
6  @Override
7  public void TagChanged(OBEvent arg0, Object arg1) {
8    OBTag source = (OBTag) arg0.getSource();
9
10   try{
11     if(source.getTagInt() == 1){
12       saraManager.reinit();
13     }
14
15   }catch(Exception e){
16     LOG.error(e.getMessage());
17   }
18 }
```

Ausschnitt 6.10: Datenbank-Listener

6.2.3 Graph

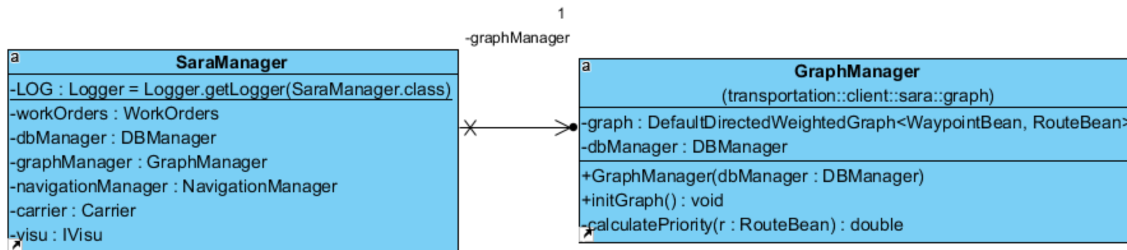

Abbildung 6.10: UML Graph

Der *GraphManager* hat die Aufgabe, den Graph mithilfe der vom *DBManager* geladenen Daten zu initialisieren. Es handelt sich dabei um einen gerichteten, gewichteten Graphen (*DirectedWeightedGraph*). Die Wegpunkte werden als Knoten (= *Vertexes*) hinzugefügt, die Routen dienen als Kanten (= *Edges*). Beim Erzeugen des Graphen muss die Klasse der Kanten (*RouteBean*) übergeben werden.
Für die Kanten wird mithilfe der Länge und der Durchschnittsgeschwindigkeit (in km/h) die Gewichtung berechnet, wie in Ausschnitt 6.12 zu sehen ist.

```
1  public void initGraph() {
2    graph = new DefaultDirectedWeightedGraph(RouteBean.class);
3
4    //add all Waypoints as Vetexes
5    for(Object i : dbManager.getWaypoints().keySet()){
6      graph.addVertex(dbManager.getWaypoints().get(i));
7    }
8    // add Routes as Edges
9    for(RouteBean r : dbManager.getRoutes()){
10     graph.addEdge(r.getWaypoint_A(),r.getWaypoint_B(),r);
11     graph.setEdgeWeight(r, calculatePriority(r));
12   }
13 }
```

Ausschnitt 6.11: Initialisierung des Graphen

```
private double calculatePriority(RouteBean r) {
  double speed = r.getAverage_speed() / 3.6;
  float distance = r.getDistance();
  return distance/speed;
}
```

Ausschnitt 6.12: Berechnung der Gewichtung

6.2.4 Navigation

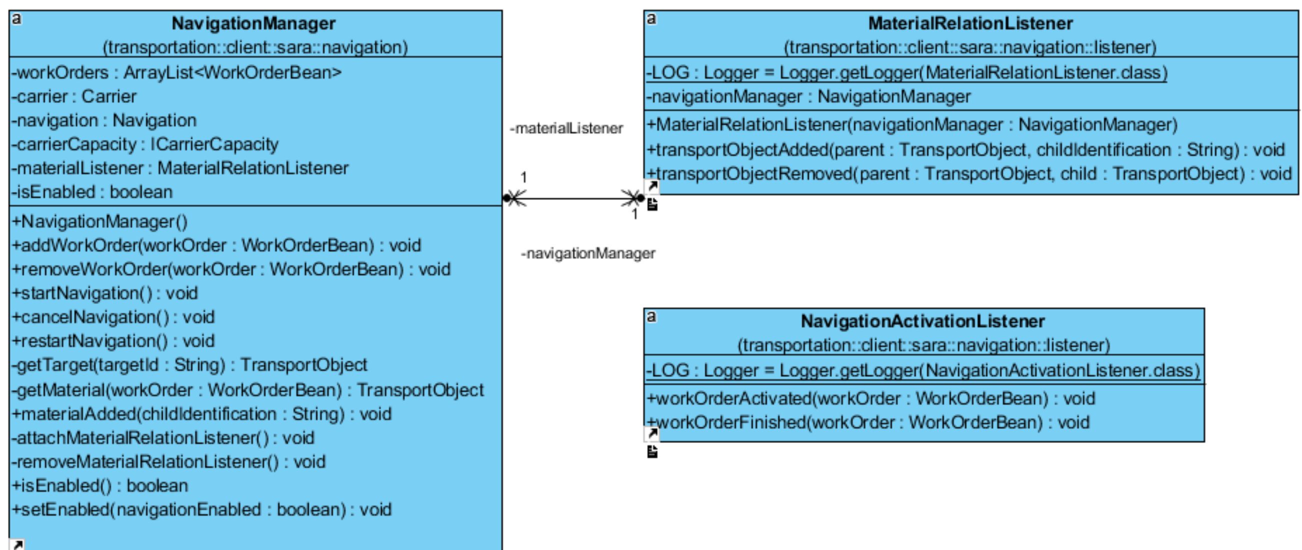

Abbildung 6.11: UML Navigation

Der *NavigationManager* ist für die Verwaltung der Navigation zuständig. Die Navigation wird vom *NavigationActivationListener* gestartet. Dieser implementiert die Methoden *workOrderActivated()* und *workOrderFinished()* aus dem Interface *WorkOrderListener* und wird ausgelöst, wenn sich der Status eines Fahrauftrages ändert. Wird ein Fahr-

auftrag aktiviert, wird er in die vom *NavigationManager* verwaltete Collection an Fahraufträgen (=*workOrders*) hinzugefügt. Wird der Fahrauftrag wieder deaktiviert oder pausiert, wird er wieder aus der Collection entfernt.

Der Listener *MaterialRelationListener* implementiert das Interface *TransportRelationActionListener* der Firma ABF und wird aufgerufen, wenn das Transportfahrzeug ein Produkt aufnimmt. In diesem Fall wird überprüft, ob für das aufgenommene Produkt ein aktiver Fahrauftrag besteht, ist die Navigation zur Quelle des Fahrauftrages beendet und die Navigation zum Ziel beginnt. Das Aufnehmen von Produkten, deren Fahrauftrag nicht aktiv ist beziehungsweise das Ablegen von Produkten muss nicht berücksichtigt werden, da in diesem Fall der Status des Fahrauftrages vom System der Firma ABF auf "aktiviert" oder "pausiert" gesetzt wird und dadurch der *WorkOrderListener* benachrichtigt wird.

Mehrere Fahraufträge
Die Navigationslogik ist so implementiert, dass bei mehreren, gleichzeitig aktiven Fahraufträgen die optimale Route für die Abarbeitung aller aktiven Fahraufträge berechnet wird. Der genaue Ablauf wird im Kapitel 5.2.2 Mehrere Fahraufträge näher beschrieben.

6.2.4.1 Kapazität von Transportfahrzeugen

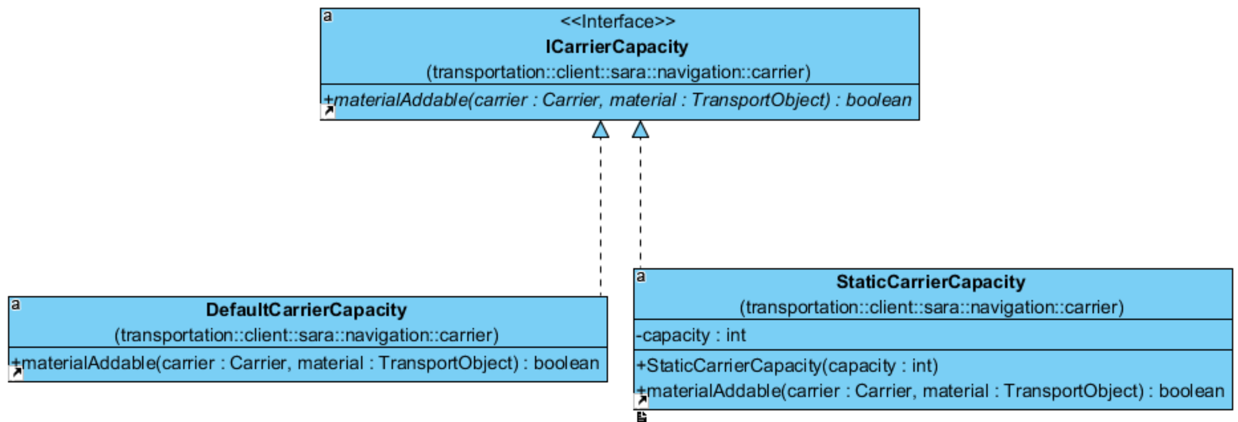

Abbildung 6.12: UML Kapazität

Um feststellen zu können, ob ein Transportfahrzeug zu einem gewissen Zeitpunkt über genug freie Kapazitäten verfügt, um ein Produkt aufnehmen zu können, müssen verschieden Kriterien berücksichtigt werden. Diese sind abhängig von der Art des Fahrzeuges und die Beschaffenheit der Produkte. Das Interface *ICarrierCapacity* definiert eine einheitliche Schnittstelle, welche für die jeweiligen, auftragsspezifischen Kriterien implementiert werden kann.

Wie in Abbildung 6.12 zu sehen ist, wurden zwei konkrete Ausprägungen dieser Schnittstelle implementiert. Bei *DefaultCarrierCapacity* wird angenommen, dass ein Transportfahrzeug jeweils nur ein Produkt aufnehmen kann. Bei der Klasse *StaticCarrierCapacity* kann ein Fahrzeug eine gewisse Anzahl von Produkten aufnehmen. Diese Anzahl wird für die jeweiligen Fahrzeugtypen in der Tabelle *OB_Properties* definiert.

6.2.4.2 Routenfindung

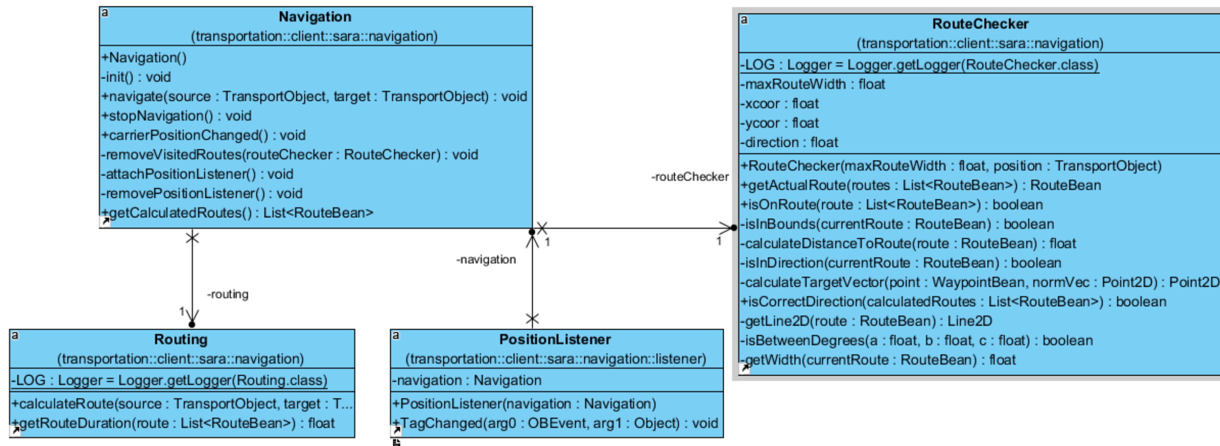

Abbildung 6.13: UML Routenfindung

Nachdem berechnet wurde, zu welchem Ziel navigiert werden soll, wird die Methode *navigate()* in der Klasse *Navigation* aufgerufen. Diese hat als Parameter zwei von *Transport-Object* abgeleitete Objekte für den Start und das Ziel der Navigation.

Um die aktuelle Position des Fahrzeuges zu überwachen, gibt es im System der Firma ABF die Schnittstelle *OBEventListener*, welche in der Klasse *PositionListener* implementiert wurde. Verändert sich die Position, wird die Methode *carrierPositionChanged()* aufgerufen. Dort wird überprüft, ob das Transportfahrzeug die optimale Route verlassen hat. Mehr über die Berechnung der Abweichung in Kapitel 6.2.4.4 Abweichung von der Route.

Befindet sich das Fahrzeug noch auf der optimalen Route, werden bereits besuchte Teilstücke aus der Visualisierung entfernt und die Fahrtrichtung des Fahrzeuges überprüft. Bewegt sich das Fahrzeug in die falsche Richtung, wird auf dem Bildschirm ein Warnhinweis ausgegeben. Hat das Fahrzeug hingegen die Route verlassen, wird die Navigationslogik neu gestartet und die Berechnung der optimalen Route neu durchgeführt.

6.2.4.3 Berechnung der optimalen Route

Für die Berechnung der optimalen Route ist die Klasse *Routing* verantwortlich. Mithilfe von *LocationWaypointFinder* werden die Wegpunkte für Start und Ziel, welche als Parameter in Form von Transport-Objekten übergeben werden, berechnet. Die Berechnung erfolgt durch den Dijkstra-Algorithmus, welcher durch das Framework JGraphT bereitgestellt wird.

```
1 IWaypointFinder finder =
2       new LocationWaypointFinder(waypoints);
3 //Waypoint search
4 WaypointBean start = finder.getNearestWaypoint(source);
5 WaypointBean end = finder.getNearestWaypoint(target);
6 // calculate shortest Path
7 List<RouteBean> path =
8   DijkstraShortestPath.findPathBetween(graph, start, end);
```

Ausschnitt 6.13: Berechnung der optimalen Route

6.2.4.4 Abweichung von der Route

Bei der Berechnung, ob ein Transportfahrzeug die optimale Route verlassen hat, wird eine Toleranz berücksichtigt. Die Methode *isOnRoute()* berechnet, ob sich das Transportfahrzeug auf der als Collection von *RouteBeans* übergebenen Route befindet.

Um diese Berechnung durchführen zu können, muss zuvor bestimmt werden, auf welcher Teilstrecke der optimalen Route sich das Fahrzeug derzeit befindet. Hierfür ist die Methode *getActualRoute()* zuständig. Diese liefert die gesuchte Teilstrecke in Form eines *RouteBean* zurück. Dazu werden alle übergebenen *RouteBeans* linear durchlaufen und deren Abstand zur aktuellen Position des Fahrzeuges zur Route berechnet, wie in Ausschnitt 6.14 Berechnung nächstgelegenen Route zu sehen ist.

```java
public RouteBean getActualRoute(List<RouteBean> routes){
  RouteBean actualRoute = null;

  double minDist = -1;
  double currentDistance = 0.0;
  for(RouteBean route : routes){
    currentDistance = this.calculateDistanceToRoute(route);
    if(minDist == -1 || minDist > currentDistance){
      minDist = currentDistance;
      actualRoute = route;
    }
  }
  return actualRoute;
}
```

Ausschnitt 6.14: Berechnung nächstgelegenen Route

Die Berechnung der Abweichung wird in 3 Kriterien eingeteilt. Diese werden schrittweise abgearbeitet:

- **Breite des Weges**
  Befindet sich das Transportfahrzeug innerhalb der in den Attribute der Route definierten Breite, sind keine zusätzlichen Berechnungen notwendig. Ist für die aktuelle Route keine Breite angegeben, wird die in den Properties definierte Standardbreite für die Berechnung herangezogen.

- **Ausrichtung des Fahrzeuges**
  Befindet sich das Transportfahrzeug außerhalb der Breite der Route, wird die Ausrichtung des Fahrzeuges als Kriterium herangezogen. Kehrt das Fahrzeug bei Beibehaltung der aktuellen Fahrtrichtung wieder auf die berechnete Route zurück, wird keine Neuberechnung durchgeführt.

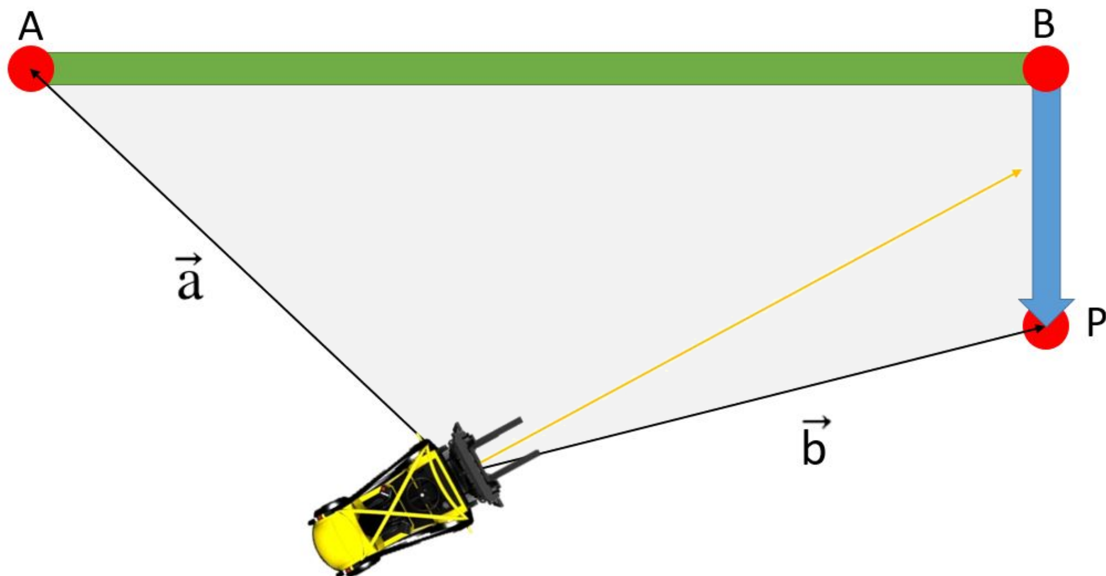

Abbildung 6.14: Ausrichtung des Fahrzeuges

Für die Berechnung wird zunächst der Normalvektor der Route berechnet. Dieser wird durch seine Länge dividiert, um den Einheitsvektor zu erhalten, und anschließend mit der Breite der Route multipliziert. Wird dieser nun mit dem Routenvektor addiert, erhält man den Punkt P, welcher in der Abbildung 6.14 zu sehen ist.

Die Vektoren $\vec{a}$ und $\vec{b}$ führen von der aktuellen Position des Transportfahrzeuges zum Startpunkt der Route A beziehungsweise zum Punkt P. Die Funktion *Math.atan2()* wird verwendet, um das grau eingezeichnete Feld zu erhalten. Die Funktion berechnet mithilfe des Arkussinus die Ausrichtung der beiden Vektoren in Grad. Liegt nun die Ausrichtung des Fahrzeuges innerhalb dieses Feldes, befindet sich das Transportfahrzeug auf der richtigen Route.

Die Berechnung, ob sich die Ausrichtung innerhalb des Feldes befindet, ist in *isBetweenDegrees()* implementiert. Liegt die X-Achse (0°) innerhalb des Feldes, muss zuvor das System so rotiert werden, dass der größere der beiden Winkel (in Ausschnitt 6.15 Vergleich der 3 Winkel ist dies der Winkel a) mit der X-Achse übereinstimmt. Danach muss überprüft werden, ob die Ausrichtung des Transportfahrzeuges (c) kleiner ist als der kleinere der beiden Winkel (b).

```
1  /*
2  * checks if angle c is between angle a and b
3  */
4  boolean isBetweenDegrees(float a, float b, float c) {
5    if(b > a){ /*swap so a is bigger than b*/
6      float x = a;
7      a = b;
8      b = x;
9    }
10
11   if(a - b < 180){
12     // c is between a and b
13     return b <= c && c <= a;
14   }else{ // 0 degrees is between a and b
15     // move bigger angle to x axis
16     float degree = 360.0f - a;
17     // move other angles by the same amount
18     b = (b + degree) % 360;
19     c = (c + degree) % 360;
20     return c <= b;
21   }
22 }
```

Ausschnitt 6.15: Vergleich der 3 Winkel

- **Nächstgelegene Route**
  Befindet sich das Transportfahrzeug laut den beiden oberen Kriterien nicht mehr auf der optimalen Route, wird berechnet, ob die Distanz zu einer anderen Route geringer ist als die Distanz zur optimalen Route. Dies wird wiederum mit dem in Ausschnitt 6.14 beschriebenen Algorithmus gelöst.

  Ist die nächstgelegene Route kein Teilstück der zuvor berechneten, optimalen Route, hat das Transportfahrzeug diese verlassen und die Navigation wird neu gestartet.

6.2.5 3D-Visualisierung

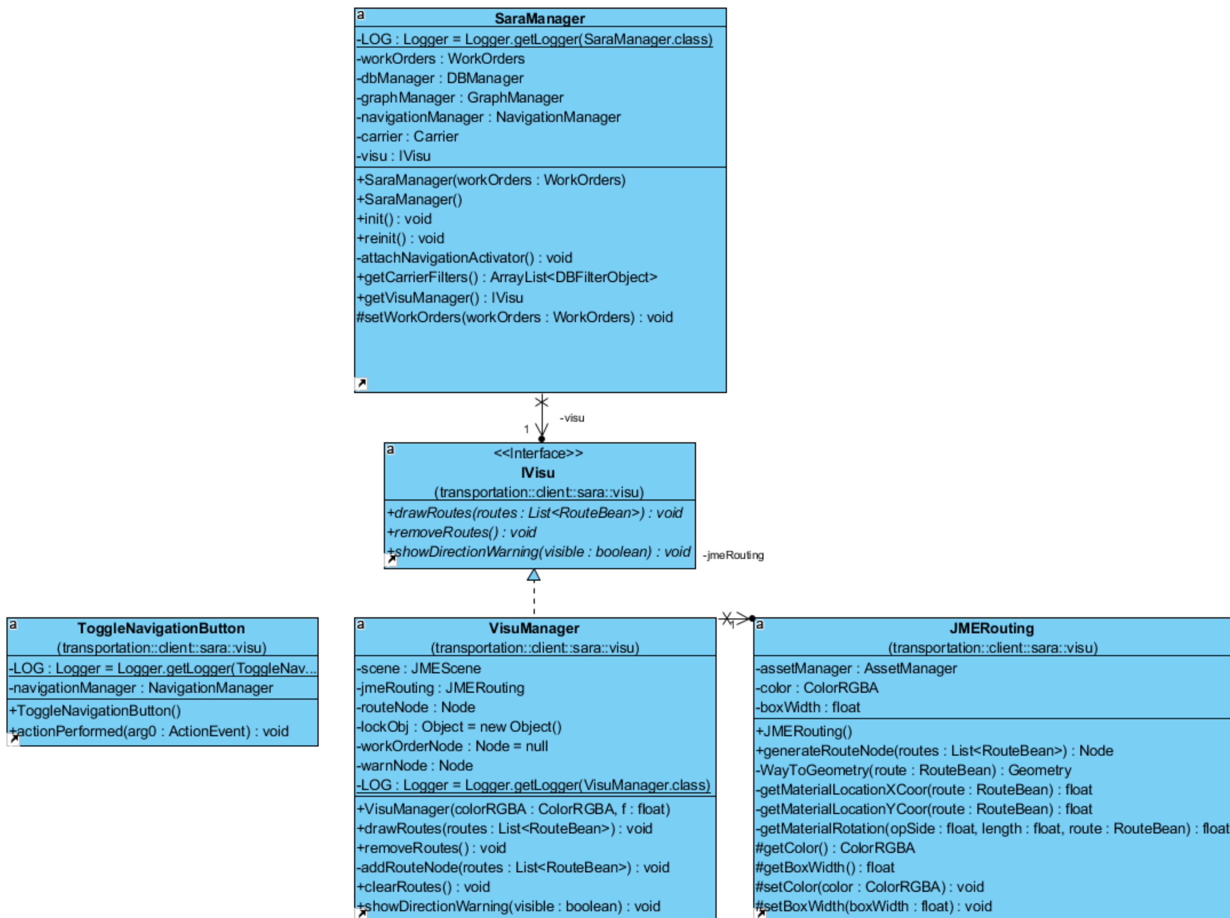

Abbildung 6.15: UML 3D-Visualisierung

Ausgangssituation für die Visualisierung ist eine Liste von *RouteBean*-Objekten, die den optimalen Weg darstellen und von der Navigationslogik an den *VisuManager* übergeben werden. Dies ist im Interface *IVisu* definiert. Die Wegpunkte der *RouteBean*-Objekte werden mit einer Linie verbunden, welche den Weg repräsentieren. Aufgrund der dreidimensionalen Ansicht in JMonkeyEngine wird diese Linie als *Geometry* in Form einer Box dargestellt. Wie in Abbildung 6.16 zu sehen, befindet sich die Fläche am Boden des virtuellen Lagers.

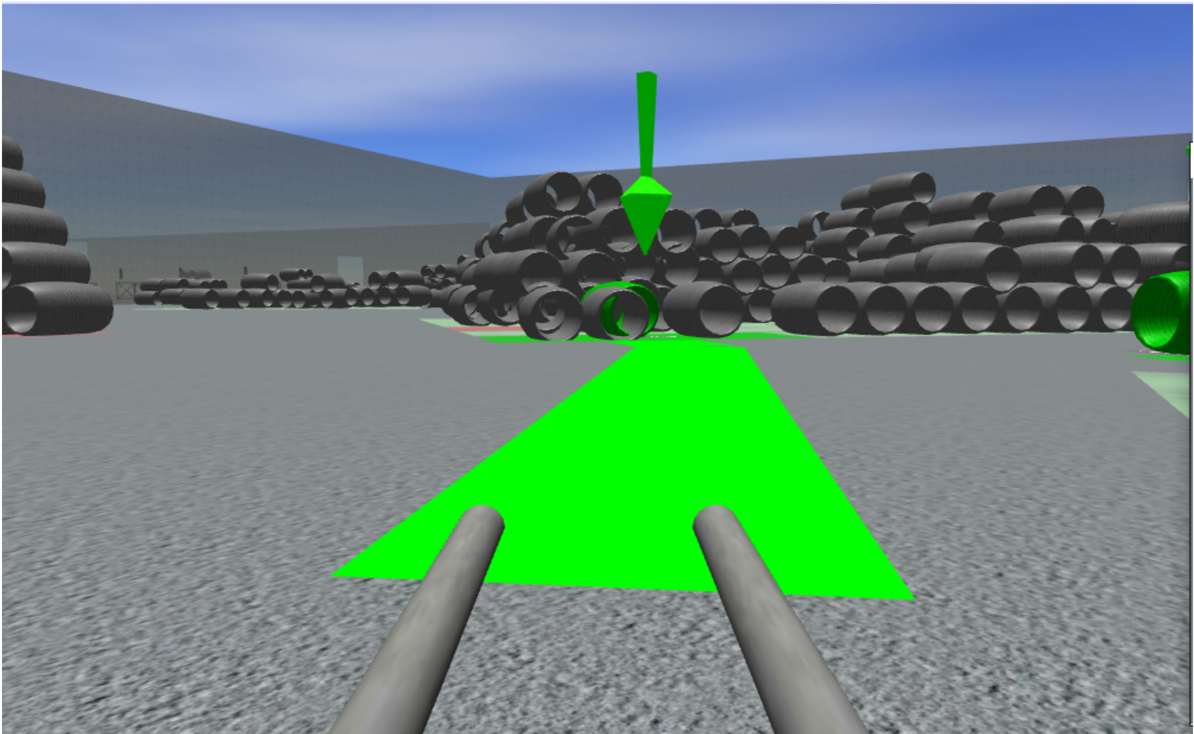

Abbildung 6.16: Darstellung der 3D-Visualisierung

```
Box b = new Box(x,0,z);
Geometry geom = new Geometry("Box", b);
geom.setLocalTranslation(x,0.001,z);
Material mat = new Material(assetManager, "Unshaded.j3md");
mat.setColor("Color", color);
geom.setMaterial(mat);
geom.rotate(x,y,z);
```

Ausschnitt 6.16: Geometry einer Route

Wie in Ausschnitt 6.16 zu sehen ist, wird ein *Geometry* in Form einer Box und mit der in *color* definierten Farbe erzeugt. Die Y-Koordinate, die in JMonkeyEngine die Höhe darstellt, wird auf 0,001 gesetzt, da sich ansonsten die beiden Ebenen überlagern

würden. Danach wird mithilfe des *AssetManager* ein *Material* erzeugt. Dieses verleiht dem *Geometry* Farbe und Aussehen.
Die Position und Ausrichtung des Weges kann über die Methoden *setLocalTanslation()* beziehungsweise *rotate()* zugewiesen werden.

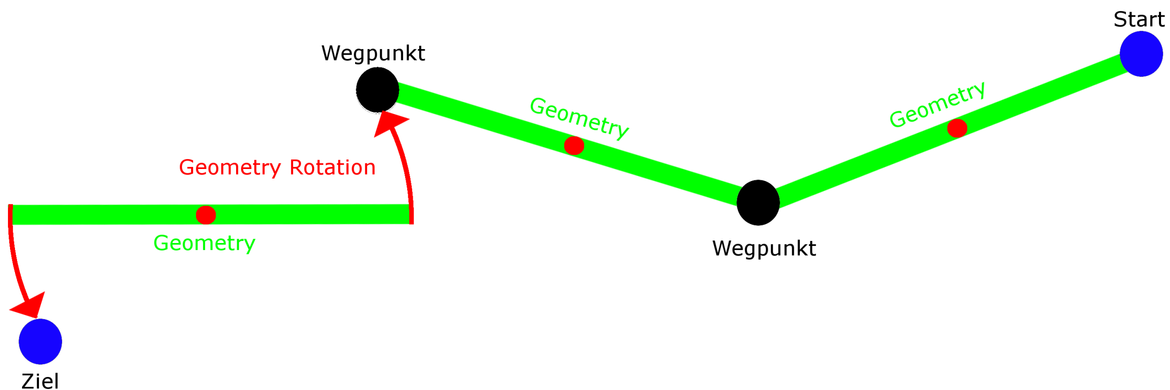

Abbildung 6.17: Aufbau der Visualisierung einer Route

In Abbildung 6.17 ist der Mittelpunkt in jedem *Geometry*-Objekt rot markiert. Das *Geometry*-Objekt, welches den Weg zwischen zwei Punkten repräsentiert, wird in die richtige Position gedreht. Der Rotationswinkel wird in Ausschnitt 6.17 berechnet.

```
private float getGeometryRotation(float opSide, float length,
RouteBean route) {

  float rotation = (float) Math.asin(opSide/length);
  if(route.getWaypoint_B().getXcoor() <
  route.getWaypoint_A().getXcoor()){
    rotation *= -1;
  }
  return(rotation);
}
```

Ausschnitt 6.17: Berechnung der Rotation

Zum Berechnen des Winkels werden Winkelfunktionen im rechtwinkligen Dreieck verwendet. Der Rotationswinkel α wird wie folgt berechnet:

$\alpha = acsin(\frac{Gegenkathete}{Hypotenuse})$

Die Länge des Weges ist dabei die Hypotenuse und die Gegenkathete wird durch die Differenz der Y-Koordinaten der Wegpunkte errechnet. Anschließend wird überprüft, ob die X-Koordinate des Wegpunkts B größer ist als die X-Koordinate des Wegpunkts A. Wenn dies zutrifft, muss das *Geometry*-Objekt negativ um α rotiert werden. Siehe Abbildung 6.18.

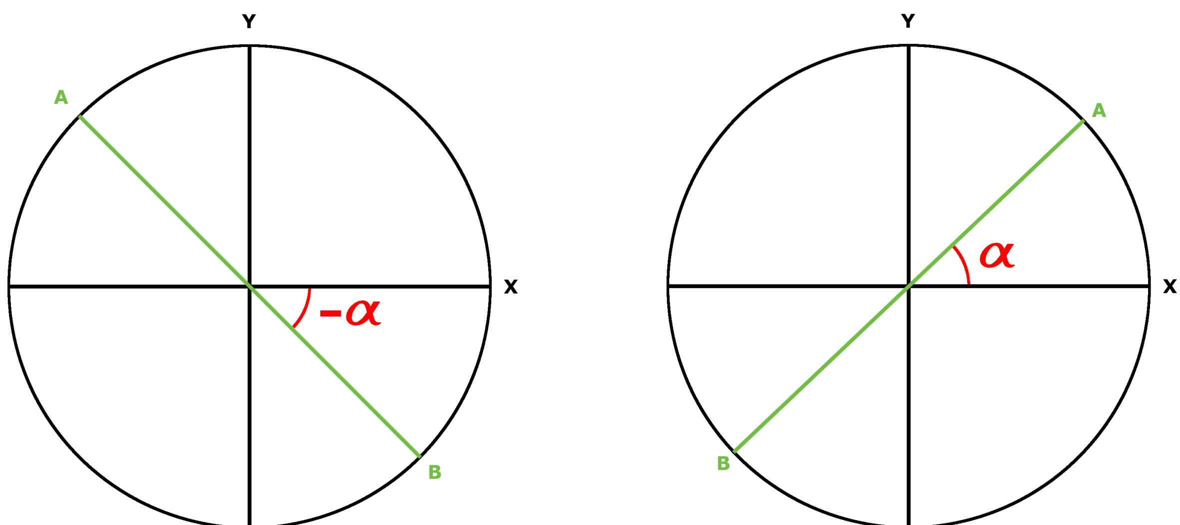

Abbildung 6.18: Rotation der Routen

In Ausschnitt 6.17 wird überprüft, ob die X-Koordinate des Wegpunkts B kleiner ist. Der Grund dafür ist, dass die Y-Koordinaten der Wegpunkte in den *RouteBean*-Objekten positiv sind, in der Visualisierung der Firma ABF jedoch negative Werte haben.
Nach der Positionierung und Rotation der *Geometry*-Objekte werden diese an ein *Node*-Objekt mit dem Namen *routeNode* gehängt. An diesem *routeNode* hängen alle *Geometry*-Objekte und ergeben gemeinsam die zu visualisierende Route.

Die Methoden zur Berechnung sind in der Klasse *JMERouting* implementiert. Im *VisuManager* wird ein Objekt der Klasse *JMERouting* erzeugt. Dadurch kann der *VisuMa-*

nager die Route erzeugen und erhält über die Methode *generateRouteNode()* die *routeNode*. Die Visualisierung der Firma ABF ist in sogenannte *groupingNodes* unterteilt. Die Klasse *JMEScene* verwaltet diese *groupingNodes* und fügt diese an die *rootNode* der Visualisierung von JMonkeyEngine. Der *VisuManager* erzeugt in der Klasse *JMEScene* eine neue Node mit dem Namen *sara_route*. Dadurch wird die Visualisierung der Route in die Visualisierung der Firma ABF integriert.

Analog zum *EventDispatchThread* in Swing gibt es in *JMonkeyEngine* einen Thread, in dem der *Scene Graph* modifiziert werden darf. Dies erreicht man durch das *enqueue()* analog zu *SwingUtilities.invokeLater()*. Somit wird sichergestellt, dass die Veränderung der Visualisierung in dem dafür vorgesehen Thread ablaufen.

```
1  @Override
2  public void drawRoutes(List<RouteBean> routes) {
3    JMECanvas.getApplication().enqueue(
4      new Callable<Spatial>() {
5        @Override
6        public Spatial call() throws Exception {
7          clearRoutes();
8          addRouteNode(routes);
9          return routeNode;
10       }
11   });
12 }
13
14 private void addRouteNode(List<RouteBean> routes){
15   synchronized(lockObj){
16     workOrderNode = jmeRouting.generateRouteNode(routes);
17     routeNode.attachChild(workOrderNode);
18     routeNode.updateGeometricState();
19   }
20 }
```

Ausschnitt 6.18: Enqueue in JMonkeyEngine

7 Technische Daten

Die Performance spielt bei SARA eine wesentliche Rolle. Das Modul wird in das bestehende System der Firma ABF integriert. Um dessen Performance nicht zu beeinträchtigen, müssen die Antwortzeiten möglichst kurz gehalten werden. Verlässt ein Fahrzeug beispielsweise die berechnete Route, so wird bei der Neuberechnung die Wartezeit für den Benutzer minimiert.

Für die Durchführung von Performancetests während der Implementierung von SARA wurde die Klasse PerformanceTimer implementiert. Das Testnetz, welches für diese Tests verwendet wurde, ist in 4.1.1 Aufbau eines Lagers beschrieben.

Vor Beginn der jeweiligen Aktion wird der PerformanceTimer über die Methode *startClockTimer()* gestartet, welche in Ausschnitt 7.1 PerformanceTimer zu sehen ist. Nachdem die gewünschte Aktion ausgeführt wurde, wird der Timer über die Methode *stopClockTimer()* wieder gestoppt und das Ergebnis zurückgegeben.

```
1 public void startClockTimer(){
2   startTimeNano = System.nanoTime();
3 }
4
5 public long stopClockTimer(){
6   return (System.nanoTime() - startTimeNano);
7 }
```

Ausschnitt 7.1: PerformanceTimer

7.1 Initialisierung von SARA

Der *SaraManager* wird beim Start der Visualisierung in Client3D initialisiert. Die Daten der Routen und Wegpunkte werden aus der Datenbank geladen und der Graph wird aufgebaut. Sind beim Start von SARA bereits Fahraufträge aktiv, wird die Navigationslogik gestartet und die optimale Route berechnet. Die Laufzeit für die Initialisierung von SARA bis zur ersten Visualisierung der optimalen Route beträgt im Testnetz weniger als **2 Sekunden**.

7.2 Reinitialisierung des Weggraphen

Bei Änderungen der Attribute von Routen in der Datenbank werden die Daten neu geladen und der Graph reinitialisiert. Dazu wird die aktuelle Navigation gestoppt und der Graph mit den Daten aus der Datenbank neu aufgebaut. Danach wird die Navigation fortgesetzt und die optimale Route neu berechnet.
Dieses Ereignis wird in der Praxis weniger häufig eintreten, da sich die Attribute der Routen nur selten ändern (beispielsweise bei vorübergehender Sperre einer Route).
Die Wartezeit für den Fahrer des Transportfahrzeuges bei einer Reinitialisierung entspricht in etwa der Dauer für die Initialisierung des Moduls, also maximal **2 Sekunden**.
Für den Performancetest wurde die Methode *reinit()* in der Klasse SaraManager, welche für die Reinitialisierung des Graphen verantwortlich ist, um Logausgaben erweitert:

```
1  public synchronized void reinit(){
2    PerformanceTimer reInitTimer = new PerformanceTimer();
3    reInitTimer.startClockTimer();
4    // stop current Navigation
5    if(getNavigationManager() != null){
6      getNavigationManager().cancelNavigation();
7    }
8    getDbManager().loadAllData();
9    this.graphManager.initGraph();
10   getNavigationManager().startNavigation();
11   long result = reInitTimer.stopClockTimer();
12   LOG.debug("Performance: ReInit Graph: "+result+" ns");
13 }
```

Ausschnitt 7.2: Performancetest Reinitialisierung des Weggraphen

7.3 Suchen des nächsten Wegpunktes

Bei der Suche von Start- und Endkonten im Graphen aus der aktuellen Position und dem aktuellen Fahrauftrag wird unterschieden, ob für den Endknoten (Quell- beziehungsweise Zielplatz des Fahrauftrages) ein eigener Wegpunkt definiert ist. (Details siehe Kapitel 6.2.2.3 Wegpunktsuche)
Die geladenen Wegpunkte werden dabei linear nach dem für den Zielplatz definiertem Wegpunkt durchsucht. Ist für das Ziel kein Wegpunkt definiert, wird wiederum die räumliche Distanz des Ziels zu den Wegpunkten berechnet und verglichen. Es wird also derjenige Wegpunkt gesucht, der dem tatsächlichen Ziel des Fahrauftrages räumlich am nächsten liegt.

```
1 //Performance: Nearest Waypoint search
2 PerformanceTimer timer = new PerformanceTimer();
3 timer.startClockTimer();
4
5 WaypointBean start = waypointFinder.getNearestWaypoint(src);
6 WaypointBean end = waypointFinder.getNearestWaypoint(target);
7
8 long result = timer.stopClockTimer();
9 LOG.debug("Performance: Find Waypoints: "+result+" ns");
```

Ausschnitt 7.3: Performancetest Suche der Wegpunkte

Beim Startpunkt für die Navigation handelt es sich um die aktuelle Position des Staplers. Als Wegpunkt hierfür wird der von der distanzmäßig nächstgelegene Wegpunkt verwendet. Ist für das Ziel des Fahrauftrages ein Wegpunkt definiert, liegt die Dauer der Suche der beiden Wegpunkte im Testnetz zwischen 700 und 800 Mikrosekunden.

Ist für das Ziel kein eigener Wegpunkt definiert, muss wiederum nach dem nächstgelegenen Wegpunkt gesucht werden. Dadurch verdoppelt sich die Antwortzeit der Suche auf ungefähr 1,4 Millisekunden

7.4 Abweichung von der Route

Bei der Berechnung der Toleranz in Bezug auf Abweichung von Routen werden 3 Kriterien berücksichtigt: (Details siehe Kapitel 6.2.4.4 Abweichung von der Route)

- Routenbreite
- Blickrichtung
- Nächstgelegene Route

Für den Performancetest wurde die Methode *carrierPositionChanged()* so erweitert, dass bei jeder Positionsänderung die Dauer der Berechnung der Abweichung ausgegeben wird.

```
carrierPositionChanged(){
  ...
  //Performance: Check is on Route:
  PerformanceTimer timer = new PerformanceTimer();
  timer.startClockTimer();

  // carrier on calculated route?
  if(routeChecker.isOnRoute(calculatedRoutes)){
    removeVisitedRoutes(routeChecker);
  }else{
    // calculate the nearest route
    RouteBean actualRoute =
      routeChecker.getActualRoute(allRoutes);
    // if nearest route is not part of the optimal route
    if(!calculatedRoutes.contains(actualRoute)){
      //restart Navigation
      getNavigationManager().restartNavigation();
    }
  }
  long result = timer.stopClockTimer();
  LOG.debug("Performance: Check on Route: "+result+" ns");
}
```

Ausschnitt 7.4: Performancetest Abweichung von der Route

Beim Performancevergleich der 3 Kriterien lassen sich signifikante Unterschiede feststellen. Die Laufzeit für die Berechnung der Toleranz liegt im Mikrosekundenbereich.

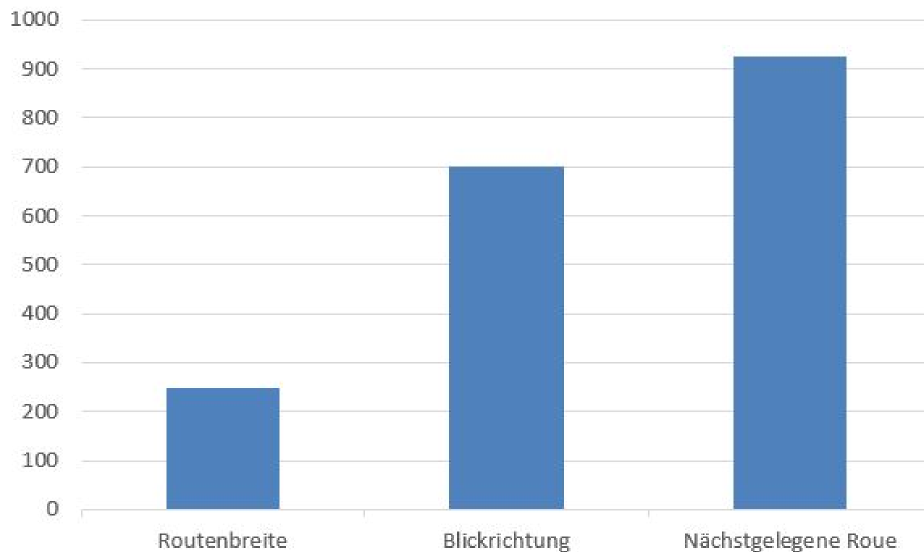

Abbildung 7.1: Performance in Mikrosekunden

- **Routenbreite**
  Befindet sich das Transportfahrzeug innerhalb der für die Route definierte beziehungsweise der in den Properties angegebenen Breite, dauert die Berechnung im Schnitt 200-300 Mikrosekunden.

- **Blickrichtung**
  Befindet sich das Transportfahrzeug außerhalb der Routenbreite, muss die aktuelle Blickrichtung und das Fenster für die Blickrichtung der Route berechnet werden. Hierbei verschlechtert sich die Performance um mehr als das Doppelte. Die Antwortzeit im Testnetz liegt bei circa 700 Mikrosekunden.

- **Nächstgelegene Route**
  Befindet sich das Transportfahrzeug aufgrund der beiden oberen Kriterien nicht mehr auf der optimalen Route, wird die distanzmäßig nächstgelegene Route berechnet. Ist diese ein Teilstück der berechneten optimalen Route, befindet sich das Transportfahrzeug noch auf dem berechneten Weg. Die Laufzeit hierfür liegt im Testnetz knapp unter einer Millisekunde.

7.5 Neuberechnung der optimalen Route

Verlässt das Transportfahrzeug die optimale Route, so wird die Navigationslogik von Neuem aufgerufen.
Der Performancetest, welcher in Ausschnitt 7.5 zu sehen ist, ergibt für den Neustart der Navigation einen Wert von durchschnittlich 3 Millisekunden.

```
1 //Performance: Restarting Navigation and recalculating route:
2 PerformanceTimer timer = new PerformanceTimer();
3 timer.startClockTimer();
4 
5 this.cancelNavigation();
6 this.startNavigation();
7 
8 long result = timer.stopClockTimer();
9 LOG.debug("Performance: Restart Navigation: "+result+" ns");
```

Ausschnitt 7.5: Performancetest Neustart der Navigation

7.6 Dijkstra-Algorithmus

Die Suche des kürzesten Weges im Graphen durch den Dijkstra-Algorithmus, welcher von JGraphT bereitgestellt wird, dauert im Testlager zwischen 600 und 700 Mikrosekunden.

```
1 PerformanceTimer timer = new PerformanceTimer();
2 timer.startClockTimer();
3 
4 List<RouteBean> path =
5   DijkstraShortestPath.findPathBetween(graph, start, end);
6 
7 long result = timer.stopClockTimer();
8 LOG.debug("Performance: Dijkstra: "+result+" ns");
```

Ausschnitt 7.6: Performancetest Dijkstra-Suchalgorithmus

8 Projektvorgehen

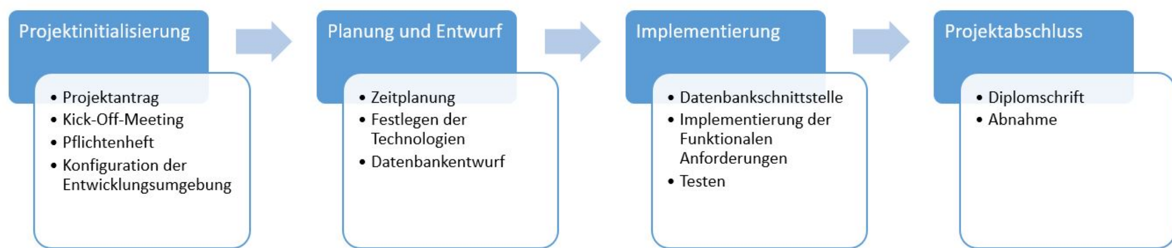

Abbildung 8.1: Projektvorgehen

Projektinitialisierung
Die Projektinitialisierungsphase begann mit dem Kick-Off Meeting. Nach der Übergabe des bereits bestehenden Systemes der Firma ABF wurde mit der Einarbeitungsphase in das System und der Konfiguration der Entwicklungsumgebung begonnen. Ein weiterer Bestandteil dieser Phase war die Einarbeitung in die Graphentheorie, sowie Routenoptimierung. Die genauen Projektziele und Anforderungen an das Modul SARA wurden vereinbart und im Pflichtenheft dokumentiert.

Planung und Entwurf
In der nachfolgenden Planungs- und Entwurfsphase wurde die Zeitplanung ausgearbeitet. In dieser Phase wurden mögliche Technologien zur Umsetzung des Projektes recherchiert und eine Nutzwertanalyse zum Vergleich der verschiedenen Graphenbibliotheken erstellt. Aufgrund dieser wurde die Entscheidung getroffen, *JGraphT* zu verwenden. Der erste Entwurf der Datenbank und der Systemarchitektur wurden ebenfalls in dieser Phase erstellt. Für die Entwicklung des Systems wurde die Datenbank mit Testdaten befüllt.

Implementierung

In der Implementierungsphase wurden die zuvor definierten Anforderungen umgesetzt. Begonnen wurde mit dem Laden der Daten aus der Datenbank und der Initialisierung des Graphen.
Bei der Umsetzung der Navigationslogik wurde zuerst die Routenfindung implementiert, welche später durch die Berechnung der Abweichung und der Unterstützung von mehreren aktiven Fahraufträgen erweitert wurde. Für die 3D-Visualisierung wurde zunächst ein unabhängiger Prototyp erstellt, bevor dieser in die bestehende Visualisierung integriert wurde.

Projektabschluss

In der letzten Phase des Projektes wurde die wissenschaftliche Arbeit erstellt und der entwickelte Source Code dem Auftraggeber übergeben.

8.1 Planung

8.1.1 Meilensteine

31.08.2015	**Einschulung in Technologien**
	Recherche Graphentheorie und Algorithmen Übergabe des Systems der Firma ABF Einarbeitung
20.10.2015	**Projektinitialisierung**
	Kick-off-Meeting Zeitplanung Aufteilung des Aufgabenbereiches Definieren der funktionalen Anforderungen
07.11.2015	**Entwurf**
	Festlegung der Technologien Datenbankentwurf
07.03.2016	**Implementierung**
	Fertigstellen der funktionalen Anforderungen Beginn der Übergabe
08.04.2016	**Diplomarbeit**
	Abgabe der wissenschaftlichen Arbeit

8.1.2 Zeitplanung

Für die Zeitplanung wurde die Microsoft Excel Vorlage für Gantt-Projektpläne verwendet. Die Aktivitäten sind entlang der Zeilen eingetragen, die Spalten repräsentieren die Zeiträume. Die Dauer eines Zeitraumes beträgt zwei Wochen. Für jede Aktivität wurde im Voraus der Soll-Beginn und die Soll-Dauer definiert. Im Laufe des Projektes wurde der tatsächliche Ist-Beginn und die Ist-Dauer eingetragen. Der Fertigstellungsgrad noch offener Aktivitäten wurde durch Prozentwerte festgelegt.

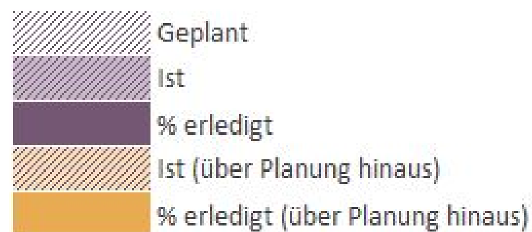

Abbildung 8.2: Gantt-Zeitplan Legende

AKTIVITÄT	GEPLANTE BEGINN	GEPLANTE DAUER	IST-BEGINN	IST-DAUER	PROZENT ERLEDIGT
Vorbereitung					
Grundsätzliche Recherche zu Graphentheorie	1	2	1	2	100%
Einarbeitung in bestehendes System	1	2	1	2	100%
Konfiguration der Entwicklungsumgebung	1	2	1	2	100%
Projektinitialisierung					
Pflichtenheft	4	7	4	6	100%
Zeitplan	4	4	4	3	100%
Risikomanagement	6	2	6	1	100%
Nutzwertanalyse Suchalgorithmen/Technologie	4	3	4	2	100%
Entwurf					
Entwurf des Wegnetzwerkes	6	3	6	2	100%
Datenbankentwurf	5	2	5	2	100%
Systementwurf	6	4	6	3	100%
Datenbankschnittstelle	8	2	6	3	100%
Implementierung					
Aufbau des Weggraphen	8	4	7	3	100%
Weg zum Routennetz	9	5	10	5	100%
Navigation	10	2	9	4	100%
Navigation - mehrere Fahraufträge	12	2	12	2	100%
Routenfindung	11	2	9	4	100%
3D - Visualisierung					
Prototyp eigenständig	12	2	9	1	100%
Einbindung in bestehende GUI	14	2	11	3	100%
Warnung bei falscher Richtung	14	1	14	1	100%
Routenabweichung	13	2	13	2	100%
Reinitialisierung Weggraph	13	1	13	2	100%
Auslagerung der Properties	15	1	13	1	100%
Einbindung der Buttons (Start/Stop)	15	1	14	1	100%
Test und Abnahme					
Abschlusstests	15	2	14	2	100%
Abnahme	16	1			

ZEITRÄUME (Beginn): 1 = 17.08.2015, 2 = 31.08.2015, 3 = 14.09.2015, 4 = 28.09.2015, 5 = 12.10.2015, 6 = 26.10.2015, 7 = 09.11.2015, 8 = 23.11.2015, 9 = 07.12.2015, 10 = 21.12.2015, 11 = 04.01.2016, 12 = 18.01.2016, 13 = 01.02.2016, 14 = 15.02.2016, 15 = 29.02.2016, 16 = 14.03.2016, 17 = 28.03

Abbildung 8.3: Gantt-Zeitplan

8.1.2.1 Ist-Zeiterfassung

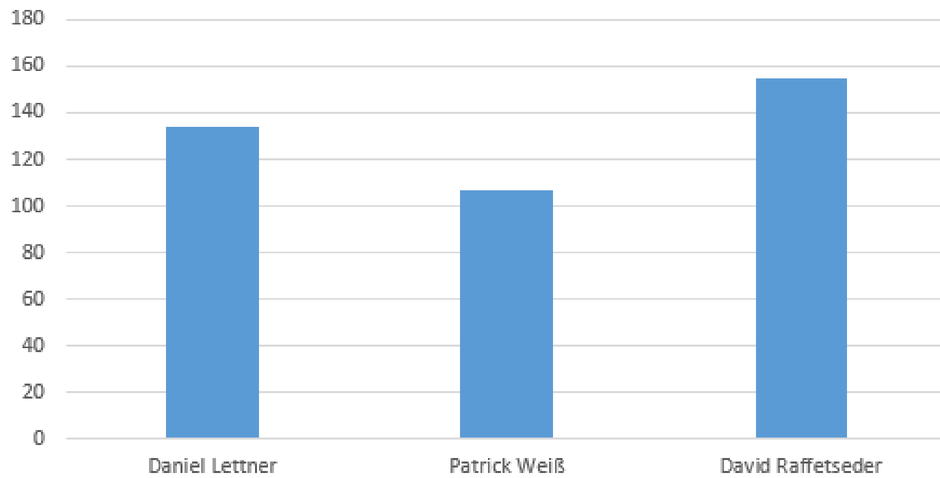

Abbildung 8.4: Ist-Stunden

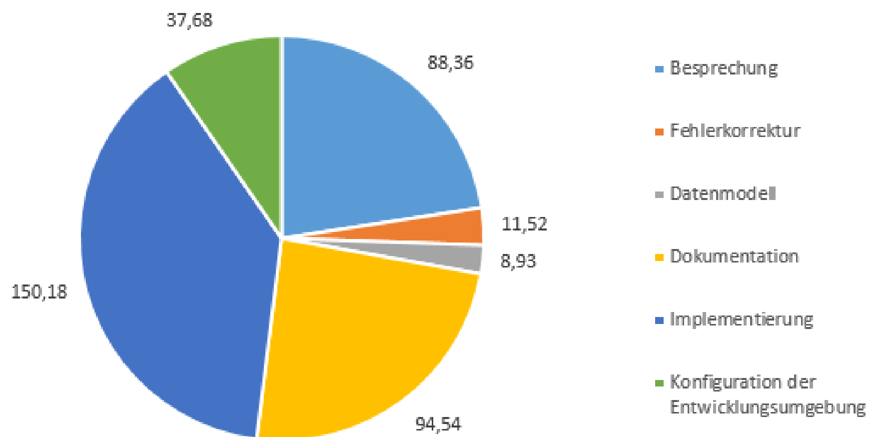

Abbildung 8.5: Stundenverteilung

8.2 Verwendete Ressourcen

8.2.1 Hardware

- **Speedlink Gamepads**
  Für die Entwicklung von SARA wurden von Seiten des Auftraggebers 3 Gamepads bereitgestellt, um die Bedienung der Staplervisualisierung zu vereinfachen.

8.2.2 Software

Zur Entwicklung von SARA wurde folgende Software verwendet:

- **Oracle XE 11**
  Auf dem Projektserver wurde Oracle XE 11 installiert. Dort wurden die für OneBase-MFT notwendigen Testdaten, welche von der Firma ABF bereitgestellt wurden, importiert. Die Testdaten umfassen ein Testlager nach Vorbild des Drahtlagers im Walzwerk der Voestalpine Austria Draht GmbH in Donawitz.
  Die Datenbank der Firma ABF wurde um Tabellen für SARA erweitert. Diese enthalten den Präfix: SARA_

- **Oracle SQL Developer**
  Zur Erstellung der Tabellen für SARA und Import der Daten wurde der SQL Developer von Oracle verwendet.

- **Eclipse**
  Die Entwicklungsumgebung Eclipse wurde von ABF zur Verfügung gestellt.

- **Visual Paradigm**
  Zur Erstellung der Klassendiagramme (siehe Kapitel 6.2 Implementierung) wurde Visual Paradigm verwendet.

8.2.3 Orgware

- **Microsoft Office**
  Zur Erstellung der Dokumente im Projekt wurde Microsoft Office verwendet. Für die Zeitplanung wurde die Microsoft Excel Vorlage für Projektplanung (Gantt) verwendet.

- **Git**
  Für die Versionsverwaltung des Source Codes wurde das Tool SmartGit verwendet. Das Repository dafür wurde von **GitLab** [git] bereitgestellt.

- **XMind**
  Die Ablaufdiagramme (siehe Kapitel 5.2 Ablaufdiagramme) wurden mithilfe des Tools XMind erstellt.

- **Kimai**
  Die Ist-Zeiterfassung der Projektmitglieder wurde über das Kimai Zeiterfassungs-Tool abgewickelt.
  Ist-Zeiterfassung siehe Kapitel 8.1.2.1 Ist-Zeiterfassung

8.3 Projektstrukturplan

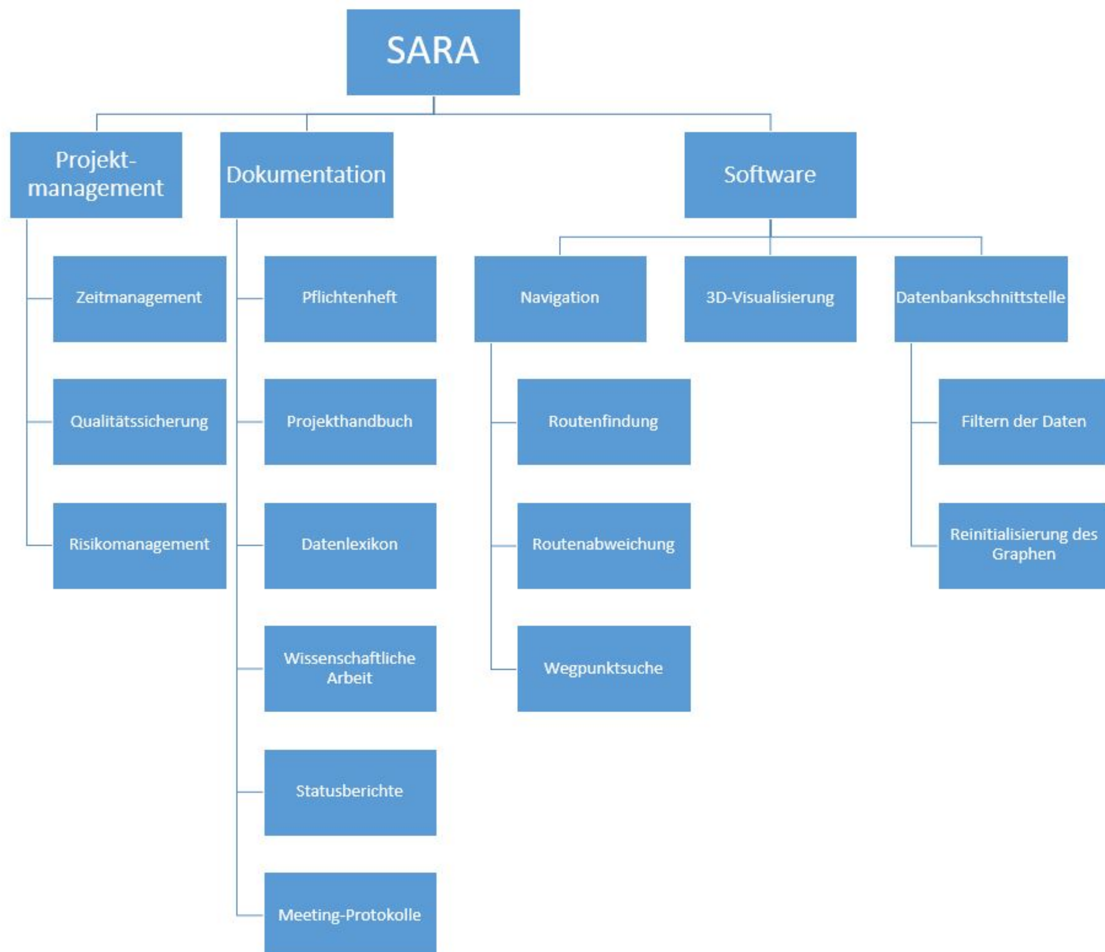

Abbildung 8.6: Projektstrukturplan

8.4 Verantwortungsmatrix

Tätigkeiten	David Raffetseder	Patrick Weiß	Daniel Lettner
Allgemeine Tätigkeiten			
Kontakt mit Auftraggeber (ABF)	V		
Qualitätssicherung	V	M	M
Datenbank Sicherung			V
QS Versionsverwaltung (Git)	M		V
Projektbeginn			
Kick-Off Meeting mit AG abhalten	M	M	V
Verantwortungsmatrix	M	V	M
Projekt- und Geschäftsziel festlegen	M	M	V
Repository anlegen		V	
Pflichtenheft	M	V	M
Benötigte HW und SW beschaffen			V
Beschreibung			
Schnittstellenbeschreibung	V	M	
Architektur festlegen	V	M	M
Dokumente			
Projektstrukturplan	V		M
Meeting-Protokolle	M	V	
Projektstatusbericht	M	V	M
Datenlexikon	M		V
Zeitplan	V		M
Dokumentation auftretender Probleme	M	M	V
Entwurf			
Entwurf des Wegnetzwerkes	M	V	
Datenbankentwurf	M		V
Systementwurf	V		M
Implementierung			
Aufbau des Weggraphen	V		M
Datenbankschnittstelle			V
Navigation - Ablauf allgemein	V		
Navigation - mehrere Fahraufträge	V		
Routenfindung	V		M
Abweichung von Routen	M		V
3D - Visualisierung		V	
Visualisierung - Prototyp eigenständig	M	V	
Visualisierung - Einbindung in System		V	
Reinitialisierung Weggraph	V		M
Auslagerung der Properties	V		
Tests	M	M	V

M = Mitarbeit V = Verantwortlich

8.5 Qualitätsmanagement

8.5.1 Qualitätsmerkmale

Für das Projekt SARA sind folgende Qualitätsmerkmale relevant:

- **Benutzbarkeit**
  Zur Benutzbarkeit zählt sowohl eine einfache und unkomplizierte Bedienung, als auch die Vermeidung von langen Wartezeiten für den Benutzer der Software.

- **Stabilität**
  Das Modul soll stabil laufen und eine gewisse Fehlertoleranz aufweisen, um unerwartete Abstürze des gesamten Systems zu verhindern.

- **Anpassbarkeit**
  Das Modul soll eine gewisse Anpassbarkeit aufweisen, um so kundenspezifische Anforderungen erfüllen zu können.

- **Erweiterbarkeit**
  Von Seiten des Auftraggebers sind Erweiterungen des Systems geplant (siehe Kapitel 9.5 Zukünftige Erweiterungen). Daher bietet SARA Schnittstellen, um diese Erweiterungen realisieren zu können.

8.5.2 Maßnahmen zur Qualitätssicherung

8.5.2.1 Versionsverwaltung

Zur Erfassung von Änderungen an Dokumenten und Source Code wurde das Versionsverwaltungstool *SmartGit* verwendet. Dadurch wurde sichergestellt, dass das gesamte Projektteam jederzeit auf die aktuelle Version der Dokumente zugreifen kann.

8.5.2.2 Meetings

Im Laufe des Projektes wurden monatlich Meetings mit dem Auftraggeber abgehalten, um den Fortschritt bei der Implementierung zu besprechen und diese zu testen. Zudem wurden monatlich Statusberichte angefertigt, in denen die aktuellen Aufgaben dokumentiert wurden. Dadurch wurde das Risiko, dass das System nicht den Vorstellungen des Auftraggebers entspricht, eliminiert.

8.5.2.3 Laufende Tests

Bereits während der Entwicklung wurden laufend Tests durchgeführt. Diese umfassten sowohl Tests der einzelnen Komponenten, als auch Tests des gesamten Systems. Dadurch wurde die Benutzbarkeit und Stabilität von SARA sichergestellt.

8.5.2.4 Definieren von Schnittstellen

Um die Erweiterbarkeit beziehungsweise Anpassbarkeit von SARA zu garantieren, wurden Schnittstellen definiert. Beispielsweise wurde für die Berechnung der Kapazität von Transportfahrzeugen das Interface *ICarrierCapacity* erstellt, welches für die jeweiligen Fahrzeugtypen implementiert werden kann, um so die projektspezifischen Anforderungen erfüllen zu können.

9 Resümee und Ausblick

9.1 Projektvorgehen

Bei der Planung des Projektes sowie der Erstellung der Dokumente konnte das Vorwissen aus dem Projektentwicklungsunterricht angewandt werden. Durch die Erfahrung aus vorangegangenen Projekten war es uns möglich, einen realistischen Zeitplan zu erstellen. Dadurch ist es uns gelungen, die zu Beginn des Projektes festgelegten Meilensteine einzuhalten. Die Implementierung der funktionalen Anforderungen wurde bereits vor Abschluss des Projektes dem Auftraggeber übergeben, sodass SARA noch vor der Präsentation bei der Logistikmesse LogiMat [log16] in Stuttgart in das System der Firma ABF integriert werden konnte.

Das Projekt wurde in die drei Aufgabenbereiche Businesslogik, Routenfindung und 3D-Visualisierung unterteilt. Der verhältnismäßige Umfang und die Komplexität der Routenfindung wurde zu Beginn unterschätzt. Da jedoch die Erstellung des Prototypen der 3D-Visualisierung und die Integration in das bestehende System vor dem geplanten Termin fertiggestellt wurde, konnte diese Fehleinschätzung ausgeglichen werden.

9.2 Kommunikation

Die Kommunikation mit der Firma ABF verlief reibungslos und ohne Komplikationen. Unser Ansprechpartner Dipl.-Ing. Andreas Spernede war immer sehr engagiert und antwortete innerhalb weniger Tage auf Fragen, die im Laufe des Projektes auftraten. Durch umfangreiche monatliche Statusberichte wurde unser Auftraggeber, sowie der Betreuungslehrer über den aktuellen Stand des Projektes informiert.

9.3 Einarbeitung in neue Technologien

9.3.1 OneBase-MFT

Da das System der Firma ABF kaum dokumentiert ist, waren in der Einarbeitungsphase des Projektes zahlreiche Meetings notwendig. In diesen Meetings erklärte uns der Ansprechpartner die einzelnen Teile des Systems. Zu Beginn der Implementierung wurden die Schnittstellen von OneBase-MFT zum Modul SARA, wie zum Beispiel die Listener zur Überwachung der Position des Transportfahrzeuges, gemeinsam mit dem Auftraggeber entwickelt.

9.3.2 Graphenbibliothek

Bei der Entscheidungsfindung zur Auswahl der Graphenbibliothek wurden zuerst mögliche Alternativen im Internet recherchiert. Nach einer Vorauswahl einigten wir uns auf die drei Bibliotheken *JGraphT*, *Jung* und *yworks*. Diese wurden unter uns aufgeteilt, sodass jeder die Details zu den einzelnen Bibliotheken in Hinblick auf die Erfüllung der Anforderungen recherchierte. Die gewonnen Erkenntnisse wurde in einer Nutzwertanalyse gesammelt, sodass sich am Ende *JGraphT* als am besten geeignet herausstellte. Diese Herangehensweise war unseres Erachtens sehr effektiv, da sich jedes Teammitglied mit einer Bibliothek beschäftigte und sein Wissen anschließend weitergab.

9.4 Aufgetretene Probleme

9.4.1 Serverinstallation

Durch die mangelhafte Dokumentation gab es Schwierigkeiten, die Services sowie den Eclipse Workspace von ABF auf den Projektserver beziehungsweise die lokalen Notebooks zu migrieren. Da beim Import der Datenbank die von ABF erstellten Benutzer nicht übergeben wurden, musste in der Konfigurationsdatei des OBSocketServers der neue Benutzer eingetragen werden. Durch intensive Beschäftigung mit den Logausgaben der Services und der Kommunikation mit dem Auftraggeber konnte dieses Problem behoben werden.

9.4.2 JMonkeyEngine

Nach der Integration der Routen in die 3D-Visualisierung ist in unregelmäßigen Abständen ein Fehler aufgetreten, der das gesamte System zum Absturz brachte. Der Grund dafür war, dass der Scene Graph nicht in dem dafür vorgesehene Thread modifiziert wurde. Zur Lösung dieses Problems wurde der bereits bestehende Code der Firma ABF analysiert. Dabei sind wir auf die Methode *JMECanvas.getApplication().enqueue()* gestoßen. Details zur Lösung siehe Kapitel 6.2.5 3D-Visualisierung.

9.5 Zukünftige Erweiterungen

Da sich das System der Firma ABF noch in der Entwicklungsphase befindet, wurde SARA als erweiterbares Modul entwickelt. Bereits während der Entwicklung von SARA sind vonseiten des Auftraggebers Ideen für mögliche Erweiterungen entstanden.

- Die Effizienz der Transportfahrzeuge kann gesteigert werden, indem berechnet wird, welchen Transportfahrzeugen ein Fahrauftrag zugeteilt wird. Für diese Anwendung kann der Algorithmus zur Berechnung der optimalen Route eingesetzt werden.
- Um Benutzern besseren Komfort zu bieten, kann die Navigation um eine Sprachausgabe erweitert werden.
- Die Wegpunkte und Routen müssen derzeit direkt in die Datenbank eingetragen werden. Vonseiten des Auftraggebers ist die Implementierung eines Tools zum Erstellen, Bearbeiten und Löschen von Routen und deren Attributen geplant.
- Da das Suchen des nächstgelegenen Wegpunktes nicht auf räumliche Faktoren wie Wände oder andere Transportfahrzeuge reagieren kann, ist dies als Erweiterung geplant.

Literaturverzeichnis

[abf] Homepage der ABF Industrielle Automation GmbH. URL http://www.abf.at/. [26.2.2016].

[Bac12] Gerald Backmeister. jMonkeyEngine: 3D-Engine für Java-Spiele. ME and my U, October 2012. URL http://mamu.backmeister.name/programmierung-und-skripting/jmonkeyengine-3d-engine-fur-java-spiele/. [18.2.2016].

[Die00] Reinhard Diestel. *Graphentheorie*. Springer-Verlag Heidelberg 1996, 2000, elektronische ausgabe edition, 2000. URL http://www.inf.fu-berlin.de/users/rote/Lere/2001-SS/Graphentheorie/Diestel-GraphentheorieII.pdf.

[Esp10] Prof. Dr. J. Esparza. *Diskrete Strukturen*. Lehrstuhl für Grundlagen der Softwarezuverlässigkeit und theoretische Informatik Fakultät für Informatik Technische Universität München, 2009/10. URL https://www7.in.tum.de/um/courses/ds/ws0910/folien_generated/12-Graphen-Grundlagen.pdf.

[git] GitLab. URL https://gitlab.com/.

[Gmb] Zeno Track GmbH. Homepage Zeno Track GmbH. URL http://www.zenotrack.com/index.php?id=48. [5.3.2016].

[jgra] JGraphT. URL http://jgrapht.org/. [26.2.2016].

[jgrb] JGraphT GitHub. URL https://github.com/jgrapht/. [4.3.2016].

[jmea] Dokumentation von JMonkeyEngine. URL http://wiki.jmonkeyengine.org/doku.php. [17.2.2016].

[jmeb] JMonkeyEngine Download. URL http://jmonkeyengine.org/downloads/. [17.2.2016].

[log16] LogiMat, 2016. URL http://www.logimat-messe.de/. [15.3.2016].

[pre] Using Prepared Statements. Oracle Java Dokumentation. URL `http://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html`. [4.3.2016].

[uK13] Prof. Dr. Johannes Köbler Lehrstuhl Komplexität und Kryptografie. *Vorlesungsskript Graphalgorithmen*. Humboldt-Universität zu Berlin, July 2013. URL `https://www.informatik.hu-berlin.de/de/forschung/gebiete/algorithmenII/Lehre/ss13/graphalgo/skript/ga-skript.pdf`.

Abbildungsverzeichnis

3.1 3D-Visualisierung der optimalen Route 17
3.2 MFTAdministrator 20
3.3 PositionSimulator 21
3.4 Marker-Tracking [Gmb] 22
3.5 Feature-Tracking [Gmb] 23

4.1 Aufbau eines Lagers 25
4.2 Aufbau des Wegenetzes 26
4.3 Wegenetz in JGraphT 26

5.1 Use Case Diagramm 35
5.2 Ablauf Navigation 37
5.3 Ablauf Navigation - mehrere Fahraufträge 38
5.4 Ablauf Routenfindung 39

6.1 ERD der SARA Tabellen 41
6.2 Datenlexikon Wegpunkt 42
6.3 Datenlexikon Route 43
6.4 SARA-Properties 44
6.5 UML SARA Hauptklassen 46
6.6 UML Model 49
6.7 UML Datenbankschnittstelle 51
6.8 UML Datenbankfilter 52
6.9 UML Wegpunktsuche 55
6.10 UML Graph 58
6.11 UML Navigation 59
6.12 UML Kapazität 61
6.13 UML Routenfindung 62
6.14 Ausrichtung des Fahrzeuges 65
6.15 UML 3D-Visualisierung 68
6.16 Darstellung der 3D-Visualisierung 69
6.17 Aufbau der Visualisierung einer Route 70

6.18 Rotation der Routen 71

7.1 Performance in Mikrosekunden 77

8.1 Projektvorgehen 79
8.2 Gantt-Zeitplan Legende 82
8.3 Gantt-Zeitplan 83
8.4 Ist-Stunden 84
8.5 Stundenverteilung 84
8.6 Projektstrukturplan 87

Ausschnitte

4.1 Installation der Services für OneBase-MFT 32
4.2 Konfiguration des OBSocketServers 33
4.3 Application Properties 34

6.1 Singleton Design Pattern 47
6.2 SaraPropertyType 47
6.3 Zugriff auf SARA Properteis 48
6.4 Erzeugung des OneBase-Tags 50
6.5 Konstruktor der Klasse DBFilterObject 52
6.6 setFilter( )-Methode in DBFilter 53
6.7 Erzeugen eines PreparedStatements 53
6.8 Abfrage des Objektdatentyps 54
6.9 IWaypointFinder 55
6.10 Datenbank-Listener 57
6.11 Initialisierung des Graphen 58
6.12 Berechnung der Gewichtung 59
6.13 Berechnung der optimalen Route 63
6.14 Berechnung nächstgelegenen Route 64
6.15 Vergleich der 3 Winkel 66
6.16 Geometry einer Route 69
6.17 Berechnung der Rotation 70
6.18 Enqueue in JMonkeyEngine 72

7.1 PerformanceTimer 73
7.2 Performancetest Reinitialisierung des Weggraphen 74
7.3 Performancetest Suche der Wegpunkte 75
7.4 Performancetest Abweichung von der Route 76
7.5 Performancetest Neustart der Navigation 78
7.6 Performancetest Dijkstra-Suchalgorithmus 78