Fachbereich 4: Informatik

TGraphBrowser

Implementierung eines Webservers zum Browsen von TGraphen
Implementation of a webserver for browsing TGraphs

Bachelorarbeit
im Studiengang Informatik

vorgelegt von:
Daniel Dominik Janke
danijank@uni-koblenz.de

Betreuer: Prof. Dr. Jürgen Ebert, Institut für Softwaretechnik, Fachbereich 4
Dr. Volker Riediger, Institut für Softwaretechnik, Fachbereich 4

Koblenz, im Januar 2010

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.

Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Koblenz, den . . . .

Abstract

Im Rahmen dieser Bachelorarbeit wird ein Webserver implementiert, mit dem man sich Graphen, die mit Hilfe des Java-Graphenlabors (JGraLab) erzeugt wurden, in einem Browser anzeigen lassen kann.

Es werden dem Nutzer eine tabellarische und eine graphische Darstellung des Graphen angeboten. In jeder der beiden Darstellungsarten ist das Navigieren durch den Graphen möglich.

Um Graphen mit mehreren tausend Elementen für den Nutzer überschaubarer zu machen, können die angezeigten Kanten und Knoten nach ihren Schematypen gefiltert werden. Des Weiteren kann die dargestellte Menge an Graphelementen durch eine GReQL-Anfrage oder durch explizite Angabe bestimmt werden.

This bachelor thesis describes the implementation of a web server. It is used to display graphs in a browser, which were created with the Java-GraphLaboratory (JGraLab).

The user has the choice between a tabular view and a graphical presentation. In both views it is possible to navigate through the graph.

Because graphs with several thousand elements may be very confusing for the user, he has the option to filter the displayed vertices and edges by their types. Furthermore, the displayed graph elements can be limited by a GReQL-query or by their ids.

Danksagung

An dieser Stelle will ich meinem Betreuer Dr. Volker Riediger herzlich danken, dass er es mir ermöglicht hat, eine so interessante Arbeit durchzuführen. Er stand mir immer mit Rat und vielen hilfreichen Vorschlägen zur Verbesserung des Designs und der Benutzerfreundlichkeit zur Seite.

Ein weiterer Dank geht an Daniel Bildhauer und Tassilo Horn, die mir so manches Mal bei Fragen zu GReQL weiterhelfen konnten.

Ich will auch Andreas Janke für seine Mühen bei der Suche nach Fehlern in meinen Texten danken. Seine hilfreichen Hinweise auf unverständliche Formulierungen ermöglichten es mir, die Verständlichkeit dieses Dokumentes zu verbessern.

Ein letzter Dank geht an Eckhard Großmann, der mir mit seinem Wissen bei so manchem Problem mit Lyx und Latex weiterhalf.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Vorgeschichte . . . 1

1.2 Aufgabenstellung . . . 1

1.3 Aufbau der Arbeit . . . 2

2 Grundlagen **3**

2.1 TGraphen . . . 4

2.1.1 Definition TGraph . . . 4

2.1.2 GReQL . . . 6

2.2 Webanwendungen . . . 6

2.2.1 Klassisches Modell einer Webanwendung . . . 6

2.2.2 AJAX-Modell einer Webanwendung . . . 8

2.3 Graphlayouter Graphviz . . . 9

3 Anforderungen **11**

3.1 Funktionale Anforderungen . . . 11

3.1.1 Graph laden . . . 11

3.1.2 Visualisierung . . . 12

3.1.2.1 2D-Visualisierung . . . 12

3.1.2.2 Tabellarische Darstellung . . . 12

3.1.3 Navigation . . . 13

3.1.4 Auswahl von Elementen . . . 13

3.1.5 Brotkrumenleiste . . . 13

3.1.6 Verwendung durch mehrere Benutzer . . . 14

3.2 Nicht-funktionale Anforderungen . . . 14

4 Bedienung 15
4.1 Laden eines Graphen . . . 15
4.2 Gesamtübersicht . . . 16
4.3 Tabellarische Darstellung . . . 16
4.4 Grafische Darstellung . . . 18
4.5 Auswahl der dargestellten Elemente . . . 20
4.5.1 Auswahl nach den Schematypen . . . 20
4.5.2 Auswahl durch die Ids der Elemente . . . 21
4.5.3 Auswahl durch eine GReQL-Anfrage . . . 22
4.6 Brotkrumenleiste . . . 22

5 Entwurf 24
5.1 Architektur des Servers . . . 24
5.2 Arbeitsweise der einzelnen Komponenten . . . 25
5.3 Aufbau der Nachrichten zwischen Browser und Server . . . 26
5.3.1 GET-Anfrage . . . 26
5.3.2 POST-Anfrage . . . 26
5.3.3 Aufbau der Antwort . . . 27
5.3.4 Beispiel . . . 27

6 Implementation 29
6.1 Kommunikation zwischen Browser und Server . . . 29
6.1.1 Senden einer Anfrage . . . 29
6.1.2 Verarbeitung beim Server . . . 31
6.1.3 Abarbeiten der Antwort . . . 32
6.2 Zustände der einzelnen Sessions . . . 32
6.2.1 Inhalt eines Zustands . . . 33
6.2.2 Löschen eines Zustands . . . 33
6.3 Laden eines Graphen . . . 34
6.4 Tabellarische Darstellung . . . 34
6.4.1 Bestimmung der Elemente einer Tabellenseite . . . 35
6.4.2 HTML-Darstellung eines Elements . . . 35
6.4.3 Navigation durch den Graph . . . 36
6.5 Grafische Darstellung . . . 37
6.5.1 Erzeugung einer dot-Datei . . . 37
6.5.2 Erzeugung einer svg-Datei . . . 37
6.5.3 Einbinden der svg-Grafik in die HTML-Seite . . . 38

6.5.4 Navigation durch den Graphen . . . 38
6.6 Auswahl von Elementen . . . 38
6.6.1 Auswahl der Schematypen . . . 39
6.6.1.1 Darstellung mittels HTML . . . 39
6.6.1.2 (De)Selektion eines Schematyps . . . 40
6.6.1.3 Verarbeitung beim Server . . . 43
6.6.2 Auswahl durch die Ids der dargestellten Elemente . . . 43
6.6.3 Auswahl mit einer GReQL-Anfrage . . . 43
6.7 Brotkrumenleiste . . . 44

7 Beispielablauf einer Session **45**
7.1 Schema des Beispielgraphen . . . 45
7.2 Beispielablauf . . . 47

8 Ausblick und Zusammenfassung **49**
8.1 Ausblick . . . 49
8.1.1 Sichtbarkeit der Graphen . . . 49
8.1.2 Anpassung der zweidimensionalen Darstellung . . . 49
8.1.3 Farbwähler . . . 50
8.2 Zusammenfassung . . . 50

A Inhalt der CD **51**

B Starten des Servers **52**

Abbildungsverzeichnis **53**

Listingverzeichnis **54**

Kapitel 1

Einleitung

Diese Bachelorarbeit befasst sich mit der Implementierung eines *Webservers* im Java-Graphenlabor (JGraLab [Kah06]), der ein dynamisches Browsen durch *TGraphen* ermöglicht.

In diesem einleitenden Kapitel wird zunächst kurz beleuchtet, aus welchem Kontext heraus die Idee für diese Arbeit entstanden ist. Im Anschluss werden die Aufgabenstellung und die einzelnen Schritte, in denen diese Arbeit entstanden ist, erklärt. Abschließend wird ein Überblick über den Aufbau der Arbeit gegeben.

1.1 Vorgeschichte

Bei *TGraphen* handelt es sich um typisierte, attributierte, gerichtete und angeordnete Graphen. Sie wurden im Institut für Softwaretechnik der Universität Koblenz-Landau entwickelt, um z.B. Quellcode zu repräsentieren. Um *TGraphen* effizient nutzen zu können, wurde im Rahmen der Diplomarbeit von Steffen Kahle [Kah06] das Java-Graphenlabor (JGraLab) erstellt. Es ist eine in Java implementierte Klassenbibliothek, die die erste Version des Java-Graphenlabors sowie das C++-Graphenlabor ersetzt. Diese API bietet verschiedene Möglichkeiten, um *TGraphen* und ihr *Schema* zu traversieren und zu manipulieren.

Da das Betrachten der *TGraphen* für die Benutzer schwierig ist, liegt die Idee nahe, *HTML-Files zu exportieren*. Mit Hilfe eines Browsers kann man nun durch die Graphen browsen. Im C++-Graphenlabor wird der gesamte Graph mittels einer einfachen HTML-Seite dargestellt. In der ersten Version des JGraLab, ist auch das Exportieren zu HTML-Files überarbeitet worden. Sie beinhalten nun mehrere Frames und ein wenig *Schema*-Anzeige.

In beiden Fällen gibt es jedoch ein Problem: Wenn man einen Graphen mit mehreren tausend Knoten und Kanten in ein HTML-File exportiert, so wird dieses so groß, dass die *Browser*, beim Versuch es anzuzeigen, *streiken*. Da die Anzeige des gesamten *TGraphen* deswegen oft nicht möglich ist, ist in dieser Bachelorarbeit ein *Webserver* implementiert worden, der es ermöglicht, *dynamisch* durch den Graphen zu *browsen*. Das heißt, dass immer nur ein Teil der Kanten- und Knotenmenge gleichzeitig dargestellt wird und jedes Mal, wenn man sich einen anderen Teil ansehen möchte, nachgeladen werden muss.

1.2 Aufgabenstellung

Die Idee, *TGraphen* mittels dynamischer HTML-Seiten anzuzeigen, hat gegenüber einer einzigen statischen Seite mehrere Vorteile:

1. Da immer nur ein *Teil der Kanten- und Knotenmenge* dargestellt wird, können die HTML-Seiten so klein gehalten werden, dass die Browser mit der Anzeige derselben nicht überfordert sind.

2. Durch die Beschränkung der Anzahl der gleichzeitig dargestellten Kanten und Knoten ist es *leichter* die *Übersicht* zu *behalten*.

3. Man hat bei dynamischen Seiten *Einfluss auf* die *Anzeige*. Denkbar wäre z.B. die Beschränkung auf bestimmte Typen des *Schemas*.

Um die *Dynamik einer HTML-Seite* zu erreichen, bedarf es eines *Webservers*. Seine Arbeitsweise lässt sich folgendermaßen skizzieren: Der Nutzer löst auf der im Browser angezeigten HTML-Seite eine Aktion aus, die ein Nachladen erfordert. Dies kann z.B. durch das Klicken auf einen Link geschehen. Der Browser schickt daraufhin eine Anfrage an den Webserver. Dieser generiert die neue HTML-Seite, die er im Anschluss zurückschickt und schließlich dem Nutzer angezeigt wird.

Das Ziel dieser Arbeit ist es, einen *Webserver* zu entwickeln, mit dessen Hilfe man die Knoten und Kanten eines *TGraphen* in einem *Browser betrachten* und *durch den Graphen browsen* kann.

Zur Erreichung dieses Ziels werden in einem ersten Schritt die Anforderungen, die das fertige System erfüllen soll, erhoben. Im Anschluss wird das benötigte Hintergrundwissen erworben, mit dessen Hilfe die Systemarchitektur entwickelt und das System implementiert wird.

1.3 Aufbau der Arbeit

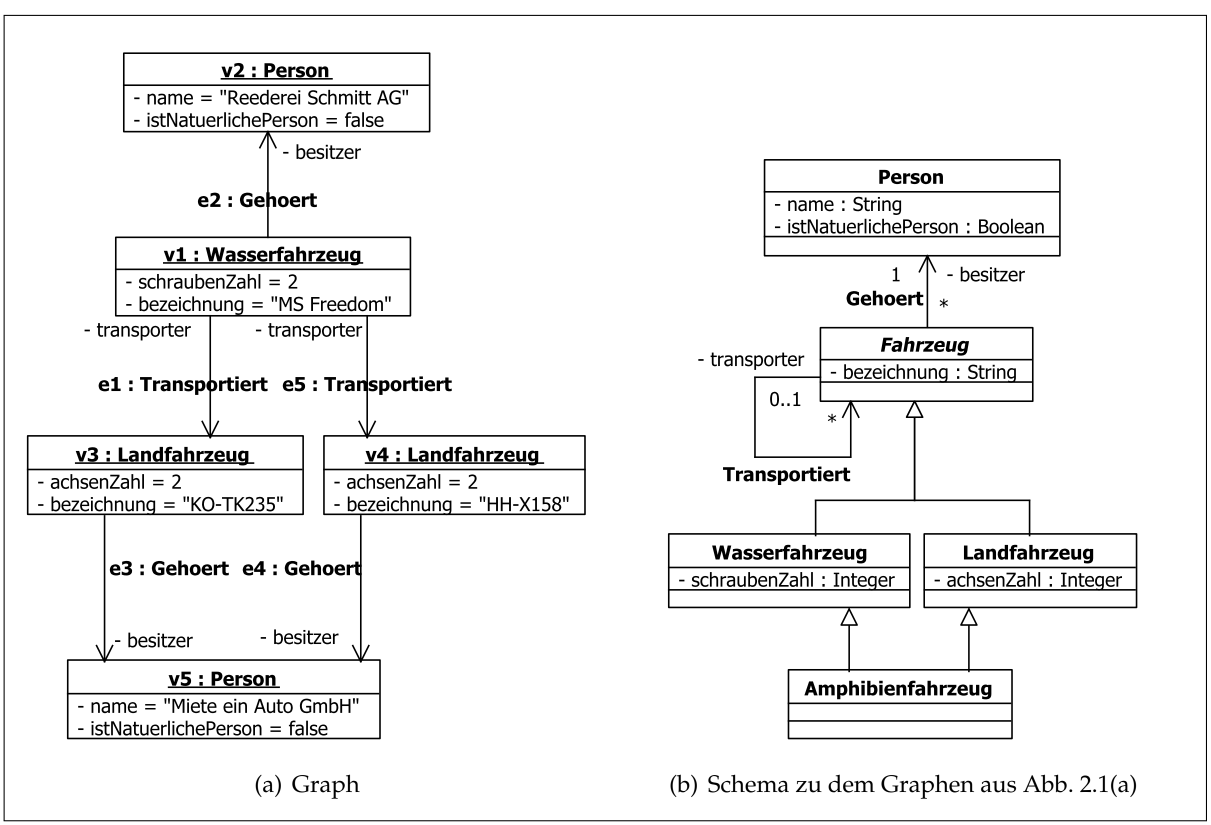
Nachdem in diesem Kapitel das Ziel der Arbeit geklärt wurde, werden im Kapitel 2 die Grundlagen, die für das Verständnis der Bachelorarbeit notwendig sind, vermittelt. Dafür werden *TGraphen*, die Kommunikationsarten einer Webanwendung sowie der Graphlayouter *Graphviz* näher beschrieben. Im Anschluss werden in Kapitel 3 die Anforderungen an das fertige System aufgelistet. Um eine bessere Vorstellung von der Arbeitsweise des *TGraphBrowsers* zu vermitteln, wird in Kapitel 4 auf seine Bedienung eingegangen. Kapitel 5 stellt die Architektur des Systems dar. Darüber hinaus wird beschrieben, wie die Anfragen an den Server und seine Antworten aufgebaut sind. Auf die Implementation des *TGraphBrowsers* wird in Kapitel 6 eingegangen. Im Kapitel 7 wird der Ablauf einer *Session* anhand eines Beispielgraphen erklärt. Dabei wird gezeigt, wie der *TGraphBrowser* dem Nutzer helfen kann, einige exemplarische Fragestellungen bei der Betrachtung eines Graphen zu lösen. Zum Abschluss wird im Kapitel 8 ein Ausblick gegeben, wie der *TGraphBrowser* noch erweitert werden könnte. Des Weiteren wird zusammengefasst, welches die Ziele waren und ob sie erreicht wurden.

Kapitel 2

Grundlagen

In diesem Kapitel sollen die Grundlagen für das Verständnis dieser Bachelorarbeit geschaffen werden. Zunächst werden in Abschnitt 2.1 die *TGraphen* sowie die *Anfragesprache GReQL* erklärt. Da zu ihrer Betrachtung ein *Webserver* implementiert werden soll, werden in Abschnitt 2.2 die unterschiedlichen Kommunikationsarten erklärt, auf die ein Browser und ein *Webserver* miteinander kommunizieren können. Im abschließenden Abschnitt 2.3 wird der Graphlayouter *dot* näher beleuchtet, mit dessen Hilfe man eine grafische Darstellung von Graphen erzeugen kann.

Die Beispiele des ersten und letzten Abschnitts dieses Kapitels beziehen sich auf den Graphen aus Abb. 2.1(a) oder sein *Schema*, welches in Abb. 2.1(b) dargestellt ist.



(a) Graph

(b) Schema zu dem Graphen aus Abb. 2.1(a)

Abbildung 2.1: Ein Graph mit dem dazugehörigen Schema

Durch das Schema wurde modelliert, dass jedes `Fahrzeug` einer `Person` gehört (`Gehoert`). Darüber hinaus kann ein Fahrzeug mehrere andere transportieren (`Transportiert`). Die Fahrzeuge

lassen sich dabei in `Land`- und `Wasserfahrzeuge` kategorisieren. Allerdings gibt es in diesem Modell auch `Amphibienfahrzeuge`, die beiden Fahrzeugkategorien angehören.

Der in Abb. 2.1(a) gezeigte Graph ist eine Instanz des soeben beschriebenen Schemas. Er stellt dar, dass die beiden `Landfahrzeuge` "KO-TK235" und "HH-X158" dem Unternehmen "Miete ein Auto GmbH" gehören. Beide Fahrzeuge werden von einem `Wasserfahrzeug` namens "MS Freedom" transportiert, welches der "Reederei Schmitt AG" gehört.

2.1 TGraphen

TGraphen wurden am Institut für Softwaretechnik (IST) der Universität Koblenz-Landau entwickelt. Sie sind typisierte, attributierte, gerichtete und angeordnete Graphen. Im Einzelnen bedeutet dies:

1. Jeder Kante und jedem Knoten ist genau *ein Typ des Schemas zugeordnet* (*Typisierung*). Diese Typen unterliegen einer Spezialisierungshierarchie, in der Mehrfachvererbung erlaubt ist. Dabei erbt jeder Subtyp alle Attribute seiner Supertypen. Ein Schematyp kann als die Menge aller Instanzen dieses Typs sowie seiner Subtypen aufgefasst werden.

   Beispiel:
   Das Schema aus Abb. 2.1(b) besteht aus den fünf Knotentypen `Person`, `Fahrzeug`, `Wasser-`, `Land`- und `Amphibienfahrzeug` sowie den beiden Kantentypen `Gehoert` und `Transportiert`. `Fahrzeug` ist abstrakt. Das bedeutet, dass sich die Menge aller `Fahrzeug`-Instanzen aus denen der `Wasser-`, `Land`- und `Amphibienfahrzeuge` zusammensetzt. In diesem Schema ist auch eine Mehrfachvererbung zu finden. Der Knotentyp `Amphibienfahrzeug` hat die beiden direkten Supertypen `Wasser`- und `Landfahrzeug`.

2. Jeder Typ besitzt eine *Menge von Attributen* mit einem bestimmten Wertebereich (*Attributierung*). Jeder Kante und jedem Knoten werden Tupel aus dem Attribut und seinem jeweiligen Wert zugeordnet. Der Wert muss dabei aus dem Wertebereich des zugehörigen Attributs stammen.

   Beispiel:
   Im Schema aus Abb. 2.1(b) besitzt der Knotentyp `Person` zwei Attribute: `name` mit dem Wertebereich `String` und `istNatuerlichePerson` mit dem Wertebereich `Boolean`. Betrachtet man die Instanz `v2` aus dem Graphen in Abb. 2.1(a), so hat `name` den Wert "Reederei Schmitt AG" und `istNatuerlichePerson` "false".

3. Jede Kante eines Graphen *ist gerichtet*. Dies bedeutet, dass sie genau einen Anfangs- und genau einen Endknoten besitzt.

4. Es existiert eine *Ordnung über die inzidenten Kanten* eines Knotens. Des Weiteren sind alle Kanten und Knoten eines Graphen in einer Sequenz angeordnet [Kah06, Mar06].

2.1.1 Definition TGraph

Auf Seite 11 seiner Diplomarbeit [Kah06] definiert Steffen Kahle *TGraphen* wie folgt:

Definition (TGraph)
Seien

typeID eine endliche Menge von **Typbezeichnern**,
attrID eine endliche Menge von **Attributbezeichnern**,
Value eine endliche Menge von **Attributwerten**,
V eine endliche Menge von **Knoten** und
E eine endliche Menge von **Kanten**.

Dann ist

G = (Vseq, Eseq, Λseq, type, value)
ein **TGraph**, falls gilt:

- $Vseq \in \text{seq } V$ ist eine Anordnung von V,
- $Eseq \in \text{seq } E$ ist eine Anordnung von E,
- $\Lambda seq : V \to \text{seq}(E \times \{in,out\})$ ist eine **Inzidenzabbildung** für die gilt:
  $\forall\, e \in E\ \exists!\, v, w \in V$:
  $(e,out) \in \text{ran } Iseq(v) \wedge (e,in) \in \text{ran } \Lambda seq(w)$,
- $type : V \cup E \to typeID$ ist eine **Typisierung** und
- $value : V \cup E \to (attrID \twoheadrightarrow\!\!\!\!\!\!\shortmid\ \ Value)$ ist eine **Attributierung**.

Da die Attribute einer Kante oder eines Knotens vom jeweiligen Schematyp abhängen, gilt weiterhin:

- $attributes : typeID \to (attrID \twoheadrightarrow\!\!\!\!\!\!\shortmid\ \ \mathbb{P}(Value))$

Außerdem müssen folgende Bedingungen erfüllt sein:

- $\forall\, e \in V \cup E$, mit $(\kappa,X) = attributes(type(e)) : \exists(\kappa,\lambda) \in value(e), \lambda \in X$
  und
- $\forall\, e \in V \cup E$, mit $(\kappa,\lambda) = value(e) : \exists(\kappa,X) \in attributes(type(e)), \lambda \in X$
  [Mar06]

Zusätzlich wird in dieser Bachelorarbeit die *Umgebung* eines Graphelements und der *Abstand* zwischen zwei Kanten benötigt. Er ist dabei definiert als:

- $distance : V \times V \to \mathbb{N}$ mit
$$distance(v,w) = \begin{cases} \infty, & \text{falls } \text{w von v aus nicht erreichbar ist} \\ 0, & \text{falls } v = w \\ min(W), & \text{falls } v \neq w \wedge W \text{ sei die Menge aller Weglängen von v nach w} \end{cases}$$ [1]

Die *Umgebung* ist wie folgt definiert:

- $environment : (V \cup E) \times \mathbb{N} \to \mathbb{P}(V \cup E)$ mit
$$environment(x,n) = \begin{cases} environment(alpha(x),n) \cup environment(omega(x),n), & \text{falls } x \in E \\ \{v \mid v \in V \wedge 0 \leq distance(x,v) \leq n\} \cup \\ \quad \{e \mid e \in E \wedge 0 \leq distance(x,omega(e)) \leq n \wedge \\ \quad\quad 0 \leq distance(x,alpha(e)) \leq n\}, & \text{falls } x \in V \end{cases}$$

Dabei liefert `alpha(x)` den Startknoten der Kante `x` und `omega(x)` ihren Endknoten.

[1] Die Weglänge ist die Anzahl der Kanten in einem Weg.

2.1.2 GReQL

Die *Graph Repository Query Language* (*GReQL*) ist eine *Anfragesprache auf TGraphen*. Ihre erste Version ist im Rahmen des GUPRO-Projekts entwickelt worden. Im Rahmen der Diplomarbeit "Entwurf und Definition der Graphanfragesprache GReQL 2" von Katrin Marchewka entstand die zweite und aktuelle Version. Diese stellt eine Erweiterung der ursprünglichen *Anfragesprache* dar. So wurde beispielsweise das Konzept der *Pfade* aufgenommen[Mar06].

Als reine *Anfragesprache* ist es mit Hilfe von *GReQL* möglich, inhaltliche, strukturelle oder aggregierte *Informationen aus TGraphen zu extrahieren*. Jedoch sind Manipulationen wie beispielsweise das Löschen bestimmter Knoten nicht erlaubt. Darüber hinaus ist *GReQL*:

1. eine **Ausdruckssprache**, d.h. Anfragen sind geschachtelte Ausdrücke.
2. **dynamisch typisiert**, d.h. Typkonflikte werden erst zur Laufzeit festgestellt.
3. **Schema-sensitiv**, d.h. auf Schemainformationen kann zugegriffen werden.
4. **in Java einbettbar**, d.h. man kann in Java-Programmen Anfragen an *TGraphen* stellen. Als Resultat wird ein *JValue*-Objekt zurückgeliefert, welches als *Wrapper* für das Ergebnis fungiert [Ebe09].

2.2 Webanwendungen

Eine *Webanwendung* ist ein Programm, das auf einem *Webserver* läuft. Der Benutzer interagiert mit ihr ausschließlich über einen Browser [Web10]. Dabei arbeiten *Webanwendungen* in der Regel *nach dem Client-Server-Prinzip*. Dies bedeutet, dass es einen Server gibt, der einen Dienst anbietet, und Clients, die diesen Dienst nutzen. Im Gegensatz zum Client, der einen Dienst aktiv anfordern muss, verhält sich der Server passiv und wartet auf eingehende Anfragen [CSM09]. Da es sich im Falle einer *Webanwendung* bei dem Client um einen Browser handelt, wird die Kommunikation durch das *Hypertext Transfer Protocol (HTTP)* geregelt.

Wie man in Abb. 2.2 sehen kann, werden von Webbrowsern *zwei unterschiedliche Arten der Kommunikation* unterstützt: das *klassische Modell* und das *AJAX-Modell*. Auf diese wird im Folgenden näher eingegangen.

2.2.1 Klassisches Modell einer Webanwendung

Beim *klassischen Modell* einer Webanwendung schickt der *Webbrowser eine HTTP-Anfrage* an den Server. Diese Anfrage wird beispielsweise durch die Eingabe einer URL oder dem Folgen eines Links gesendet. Der *Server empfängt und verarbeitet* diese Anfrage. Als Antwort wird eine gegebenenfalls neu generierte HTML-Seite mit dem Header einer *HTTP-Antwort* versehen und *zurück zum Client* geschickt. Nachdem der Browser die Antwort empfangen hat, zeigt er die neue HTML-Seite an.

Diese Art der Kommunikation hat den Vorteil, dass sie auch in Browsern mit abgeschaltetem JavaScript funktioniert. Jedoch hat sie den Nachteil, dass immer die gesamte Seite neu geladen werden muss, selbst wenn sich nur Kleinigkeiten geändert haben. Dies führt zu einem erhöhten Datenverkehr im Netzwerk.

Da das Neuladen der kompletten Seite nicht immer notwendig ist, wurde das *AJAX-Modell* entwickelt.

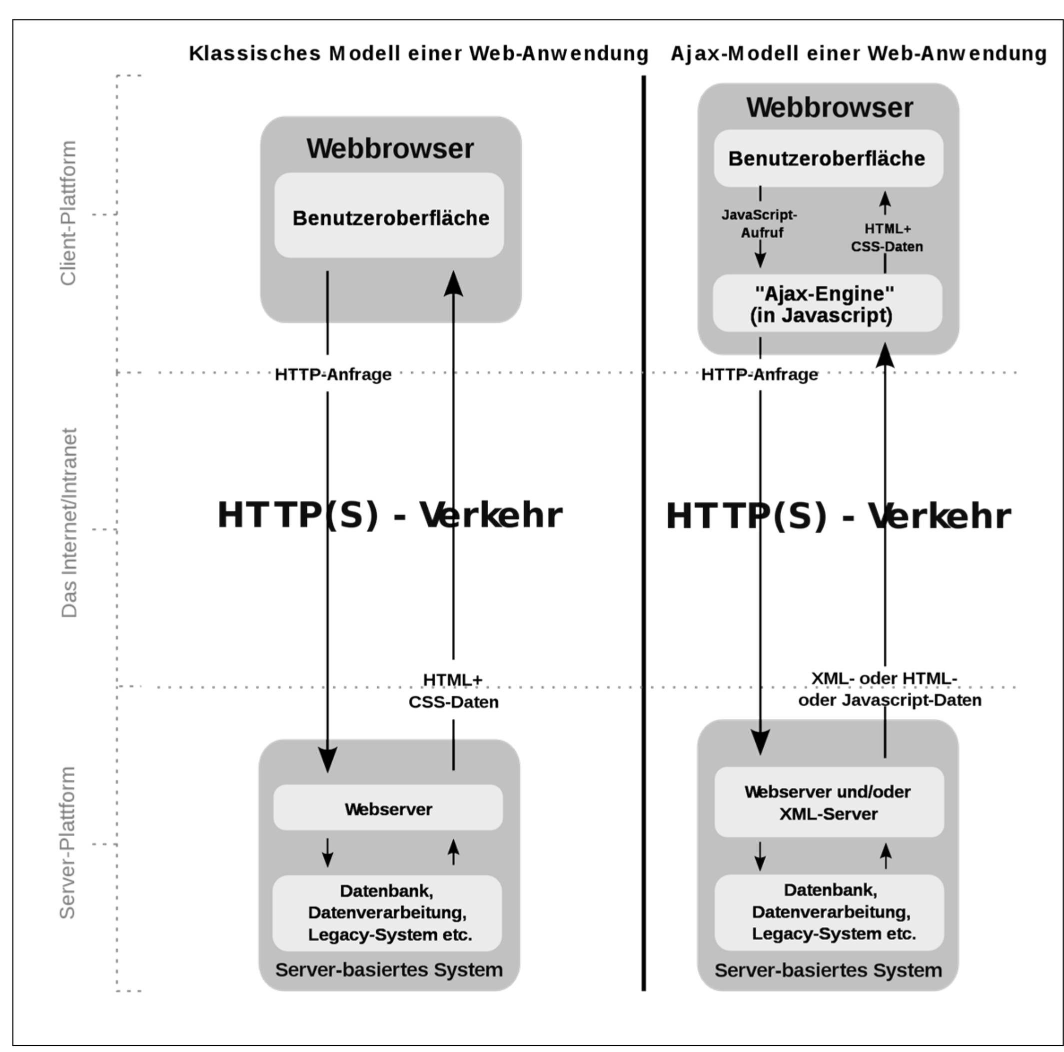


Abbildung 2.2: Die unterschiedlichen Kommunikationsarten einer Web-Anwendung. Quelle: http://commons.wikimedia.org/w/thumb.php?f=Ajax-vergleich.svg&width=2000px

2.2.2 AJAX-Modell einer Webanwendung

Der Begriff *AJAX* steht für "**A**synchronous **J**avaScript **a**nd **X**ML" und wurde von Jesse James Garrett in seinem Artikel „Ajax: A New Approach to Web Applications" vom Februar 2005 eingeführt. Er definiert *AJAX* wie folgt:

"Ajax isn't a technology. It's really several technologies, [...]. Ajax incorporates:

- standards-based presentation using XHTML and CSS;
- dynamic display and interaction using the Document Object Model;
- data interchange and manipulation using XML and XSLT;
- asynchronous data retrieval using XMLHttpRequest;
- and JavaScript binding everything together."[2]

Im Gegensatz zum *klassischen* kommt das *AJAX-Modell* nicht ohne JavaScript aus. Denn zum Senden einer HTTP-Anfrage wird per *JavaScript die AJAX-Engine aufgerufen*. Der Server verarbeitet die Anfrage und schickt eine HTTP-Antwort zurück. Im Gegensatz zum *klassischen Modell* kann sie an Stelle von HTML-Daten auch aus XML- oder JavaScript-Daten bestehen. Die *AJAX-Engine empfängt die Antwort*, die mittels JavaScript verarbeitet wird, was gegebenenfalls zu einer Veränderung der dargestellten HTML-Seite führen kann. Da JavaScript in der Lage ist XML-Daten zu erzeugen und zu verarbeiten, kann an Stelle eines *Webservers* auch ein *XML-Server* eingesetzt werden. Im Gegensatz zu einem *Webserver* speichert und versendet er XML-Daten [XML10].

Um den Webentwicklern eine Möglichkeit zu geben die AJAX-Engine zu nutzen, wurde XMLHttpRequest *entwickelt*, welches die Programmierer-Schnittstelle dieser Engine ist. Ursprünglich wurde es *als ein ActiveX-Control* im Internet Explorer 5 entwickelt. Jedoch konnte es sich zunächst nicht durchsetzen, da es als ActiveX-Control systemabhängig war.

Da die Idee des Nachladens im Hintergrund vielversprechend war, wurde sie von anderen Browserherstellern aufgegriffen. So wurden beispielsweise im Mozilla Firefox und im Safari 1.2 *XMLHttpRequest als natives JavaScript-Objekt* implementiert. Das heißt, dass es verwendet werden konnte, ohne vorher weitere Scripte oder Softwarekomponenten wie beispielsweise ein ActiveX-Control einzubinden. Ab Internet Explorer 7 ist das XMLHttpRequest auch dort als natives JavaScript-Objekt vorhanden.

Am 5. Februar 2004 veröffentlichte das World Wide Web Consortium eine Spezifikation für das XMLHttpRequest-Objekt. Die wichtigsten Felder dieser Spezifikation sind:

onreadystatechange:String In diesem Feld wird die Funktion angegeben, die ausgeführt wird, sobald sich `readystate` verändert.

readystate:int Zum Zeitpunkt, an dem die im Feld `onreadystatechange` gespeicherte Funktion aufgerufen wird, gibt dieses Feld Auskunft über den Zustand der Transaktion. Prinzipiell sind die Werte 0 bis 4 möglich. Allerdings haben die Werte 0 bis 3 in den verschiedenen Browsern unterschiedliche Bedeutungen. Deswegen wird empfohlen, nur den Wert 4 zu benutzen, der darüber Auskunft gibt, dass die Antwort komplett empfangen wurde.

responseText:String Nach Beendigung der Transaktion stehen in diesem Feld die vom Server gesendeten Daten.

[2]Quelle: http://www.adaptivepath.com/ideas/essays/archives/000385.php [Gar05]

Die wichtigsten Methoden sind:

abort():void Mit dieser Methode wird die Transaktion abgebrochen.

open(String,String,boolean):void Mit dem ersten Parameter legt man fest, welchen Anfragetyp man schicken möchte. In dieser Bachelorarbeit werden nur die Typen GET und POST verwendet. Mit dem zweiten Parameter wird die URL angegeben, an die die Anfrage geschickt wird. Mit dem dritten legt man fest, ob die Anfrage synchron (= `true`) oder asynchron (= `false`) erfolgen soll.

send(String):void Diese Methode schickt die Anfrage ab. Als Parameter bekommt sie den Body der Anfrage übergeben. Falls kein Body übergeben werden soll, nutzt man `null`.

setRequestHeader(String,String):void Mittels dieser Methode können Headerinformationen gesetzt werden. Dabei ist der erste Parameter der Name der Headerinformation und der zweite deren Wert [BB09, Wen07, W3C09].

Die Anwendung des *XMLHttpRequest* soll anhand des Beispiels in Listing 2.1verdeutlicht werden.

```
1 var xmlHttp = new XMLHttpRequest();
2 xmlHttp.open('GET', 'beispiel.xml', true);
3 xmlHttp.onreadystatechange = function () {
4             if (xmlHttp.readyState == 4) {
5                 alert(xmlHttp.responseText);
6             }
7       };
8 xmlHttp.send(null);
```

Listing 2.1: Anwendungsbeispiel des *XMLHttpRequest*.

Zunächst wird in der ersten Zeile ein neues *XMLHttpRequest*-Objekt angelegt. Im Anschluss wird es so konfiguriert, dass eine synchrone GET-Anfrage an `beispiel.xml` geschickt werden soll. In diesem Fall handelt es sich um eine URL, die relativ zur Serveradresse ist[3]. In Zeile 3, wird die Funktion gesetzt, die ausgeführt werden soll, sobald sich der Status von `xmlHttp` verändert. Ihre Arbeitsweise lässt sich dabei so zusammenfassen, dass sie, sobald die Antwort des Servers komplett empfangen wurde, deren Inhalt ausgibt. In der letzten Zeile wird die Anfrage schließlich abgeschickt.

2.3 Graphlayouter Graphviz

Der Graphlayouter *Graphviz*[4] (Graph Visualization Software) ist eine Open-Source-Software, mit deren Hilfe man Graphen visuell darstellen kann. Dazu bietet sie verschiedene Layoutverfahren an. In dieser Bachelorarbeit wird *dot* verwendet, mit dem man sich unter anderem SVG-Grafiken erzeugen kann. Zur Erstellung der grafischen Repräsentation wird der Sugiyama-Algorithmus verwendet, welcher aus vier Schritten besteht[5]:

1. Enthält der darzustellende Graph Zyklen, so werden diese aufgebrochen, indem die interne Richtung einzelner Kanten umgekehrt wird.

[3]Es ist möglich, mittels eines *XMLHttpRequest*-Objekts Anfragen auch an andere Server zu schicken. Dazu müsste die komplette URL des Servers angegeben werden.

[4]http://www.graphviz.org/

[5]Diese vier Schritte beschreiben den von *dot* genutzen Algorithmus stark vereinfacht. Für die exakte Arbeitsweise wird auf [GKNV93] verwiesen.

2. Die Knoten werden einzelnen Stufen zugeordnet, aus denen die Y-Koordinaten bestimmt werden können.

3. Innerhalb einer Stufe werden die Knoten so angeordnet, dass sich möglichst wenige Kanten überschneiden.

4. Die X-Koordinaten der Knoten werden so bestimmt, dass die Kanten möglichst kurz sind [GKN06].

Als Eingabe erwartet *dot* einen Graphen, der in der Sprache *DOT* vorliegt. Der in dieser Bachelorarbeit verwendete Teil ihrer Syntax lässt sich wie folgt definieren:

```
graph       ::=  'digraph' id '{' stmt-list '}';
stmt-list   ::=  [ stmt [';'] [stmt-list] ];
stmt        ::=  attr-stmt | node-stmt | edge-stmt | id '=' id;
attr-stmt   ::=  ( 'node' | 'edge' ) attr-list;
attr-list   ::=  '[' [a-list] ']';
a-list      ::=  id '=' id [;] [a-list];
node-stmt   ::=  id [attr-list];
edge-stmt   ::=  id '->' id [attr-list];
```

Eine `id` ist ein alphanumerischer String, der Unterstriche enthalten kann. Allerdings darf er mit keiner Zahl beginnen. Sollte von dieser Regel abgewichen werden, so muss die `id` mit " umschlossen werden [GKN06].

```
1  digraph "BeispielGraph.dot" {
2        node [shape="record"];
3        v1 [label="{{v1|Wasserfahrzeug}}"];
4        v2 [label="{{v2|Person}}"];
5        v3 [label="{{v3|Landfahrzeug}}"];
6        v4 [label="{{v4|Landfahrzeug}}"];
7        v5 [label="{{v5|Person}}"];
8        v1 -> v2 [label="e2:␣Gehoert"];
9        v1 -> v3 [label="e1:␣Transportiert"];
10       v1 -> v4 [label="e5:␣Transportiert"];
11       v3 -> v5 [label="e3:␣Gehoert"];
12       v4 -> v5 [label="e4:␣Gehoert"];
13 }
```

Listing 2.2: Die *DOT*-Repräsentation des Graphen aus Abb. 2.1(a).

Listing 2.2 zeigt die *DOT*-Repräsentation des Graphen aus Abb. 2.1(a). Aus Gründen der Übersichtlichkeit wurden die Attribute ignoriert. Durch Zeile 2 wird *dot* mitgeteilt, dass alle Knoten als Rechtecke dargestellt werden sollen. In den Zeilen 3 bis 7 werden die Knoten und von 8 bis 12 die Kanten angegeben. `v1 -> v2` bedeutet, dass die dargestellte Kante von `v1` nach `v2` verläuft.

Nachdem nun die notwendige Wissensgrundlage geschaffen worden ist, kann auf die Erstellung des *TGraphBrowsers* eingegangen werden. Der erste Schritt besteht dabei aus der Feststellung der von den Stakeholdern gewünschten Anforderungen. Um welche es sich im Einzelnen handelt und in wie weit sie umgesetzt sind, wird im nächsten Kapitel beschrieben.

Kapitel 3

Anforderungen

In diesem Kapitel geht es um die Anforderungen, die an den *TGraphBrowser* gestellt werden. Zu ihrer Erhebung wurde zunächst ein Gespräch mit Herrn Prof. Dr. Jürgen Ebert, Herrn Dr. Volker Riediger, Daniel Bildhauer, Tassilo Horn und Hannes Schwarz geführt. Im Anschluss wurden die Teilnehmer des Gesprächs gebeten, ihre gewünschten Anforderung niederzuschreiben. Die nachfolgende Auflistung entstand durch das Zusammenführen dieser Anforderungen, die zum Teil präzisiert und um weitere ergänzt wurden.

Um festzulegen, welche Anforderungen für die Bachelorarbeit relevant sind, werden die Anforderungen in drei Kategorien eingeordnet:

1. Pflicht: Pflichtanforderungen müssen realisiert werden.
2. Kann: Kannanforderungen gehören nicht zur Grundfunktionalität des Programms, sollten aber dennoch realisiert werden.
3. Wunsch: Anforderungen, die als Wunsch markiert sind, stellen Eigenschaften bzw. Funktionalitäten des fertigen Systems dar, die wünschenswert wären, jedoch nicht in der Bachelorarbeit realisiert werden müssen.

Anforderungen, die umgesetzt wurden, sind mit * am Ende markiert.

3.1 Funktionale Anforderungen

3.1.1 Graph laden

3.1.1.1 [Pflicht] Der Benutzer muss lokale Graphen zur Visualisierung hochladen können.*

3.1.1.2 [Pflicht] Der Benutzer muss aus einer Sammlung von Graphen auf dem Server einen davon zur Visualisierung auswählen können.*

3.1.1.3 [Kann] Der Benutzer soll einen Graphen zur Visualisierung per URI angeben können.*

3.1.1.4 [Kann] Der Benutzer soll einen Graphen im .tg-Format in einem File-Upload-Feld angeben können.*

3.1.1.5 [Pflicht] Bevor ein Graph geladen wird, muss sein ungefährer Speicherverbrauch vom Browser erkannt werden.*

3.1.1.6 [Pflicht] Es muss eine Größenbeschränkung für zu ladende Graphen beim Starten des Servers angegeben werden können oder vom Server aus dem ihm zugewiesenen Speicher bestimmt werden.*

3.1.1.7 [Kann] Der Benutzer muss eine Warnung erhalten, falls der Graph die Größenbeschränkung übersteigt.

3.1.1.8 [Wunsch] Es soll ein Fortschrittsbalken beim Laden eines Graphen angezeigt werden.*

3.1.2 Visualisierung

Der Term "Teilgraph" bezieht sich im Folgenden auf die Teilgraphen, die durch die Navigation (siehe Unterabschnitt 3.1.3) sowie durch die (De)Selektion der Schematypen (siehe Unterabschnitt 3.1.4) ausgewählt wurden.

3.1.2.1 [Pflicht] Der Benutzer muss zwischen einer 2D-Visualisierung und einer tabellarischen Anzeige umschalten können.*

3.1.2.2 [Kann] Es soll ausgewählt werden können, ob Knoten- und Kantenattribute mit angezeigt werden sollen.*

3.1.2.3 [Pflicht] Details der Knoten und Kanten wie Attribute müssen beim Überfahren eines Elementes mit der Maus angezeigt werden, wenn sie nicht bereits in der Visualisierung enthalten sind.*

3.1.2.1 2D-Visualisierung

3.1.2.1.1 [Pflicht] Die Knoten und Kanten eines Teilgraphen müssen als zweidimensionale Grafik angezeigt werden können.*

3.1.2.1.2 [Wunsch] Es kann eine Auswahl mehrerer Layoutverfahren zur Verfügung gestellt werden.

3.1.2.1.3 [Pflicht] Die Anzahl der Elemente des zu visualisierenden Teilgraphen muss beschränkt werden.*

3.1.2.1.4 [Pflicht] Die Größe der Umgebung in der 2D-Visualisierung ist über die Pfadlänge konfigurierbar.*

3.1.2.1.5 [Wunsch] Der Maximalwert der Pfadlänge richtet sich nach dem Layoutverfahren.

3.1.2.1.6 [Wunsch] Es kann eine Warnung erscheinen, wenn der Teilgraph mehr Elemente hat, als die Beschränkung vorgibt.

3.1.2.1.7 [Pflicht] Das selektierte Element muss hervorgehoben werden.*

3.1.2.2 Tabellarische Darstellung

3.1.2.2.1 [Pflicht] Der Benutzer muss die angezeigten Elemente begrenzen können.
Z.B. 10-20-50-100-alle.*

3.1.2.2.2 [Pflicht] Die Knoten und Kanten eines Teilgraphen müssen tabellarisch dargestellt werden können.*

3.1.2.2.3 [Pflicht] Die Darstellung eines Graphelementes muss aus der Id, dem Typ und den inzidenten Elementen bestehen.*

3.1.3 Navigation

3.1.3.1 [Pflicht] Hat der Benutzer kein Element ausgewählt, so wird der erste Knoten des Graphen ausgewählt.*

3.1.3.2 [Pflicht] Wird ein Graphelement in der 2D-Visualisierung per Klick selektiert, so muss er in der Anzeige zentriert und diese um die Umgebung erweitert werden.*

3.1.3.3 [Pflicht] Durch einen Klick auf ein Graphelement in der tabellarischen Darstellung wird dieses mit seinen inzidenten Elementen angezeigt.*

3.1.4 Auswahl von Elementen

3.1.4.1 [Pflicht] Eine globale Filterung der anzuzeigenden Elemente über Typrestriktionen muss möglich sein.*

3.1.4.2 [Pflicht] Die Knoten- und Kantentypen müssen mit ihrem SimpleName in einer Baumdarstellung angezeigt und für die Filterung ausgewählt werden können.*

3.1.4.3 [Pflicht] Der FullyQualifiedName der Typen muss beim Überfahren der Typen mit der Maus angezeigt werden.*

3.1.4.4 [Pflicht] Die Baumdarstellung der Typen muss deren Spezialisierungshierarchie widerspiegeln.*

3.1.4.5 [Kann] Die Äste der Baumdarstellung sollen ein- und ausklappbar sein.*

3.1.4.6 [Wunsch] Das gleichzeitige Ein- und Ausklappen aller Äste soll mit jeweils einem Klick möglich sein.*

3.1.4.7 [Kann] Nach dem Laden eines Graphen soll nur die erste Ebene der Generalisierungshierarchie des Schemas direkt in der Baumdarstellung sichtbar sein.*

3.1.4.8 [Pflicht] Bei Auswahl eines Typs müssen automatisch alle Subtypen mit ausgewählt werden.*

3.1.4.9 [Pflicht] Bei Deselektion eines Typs müssen alle Subtypen deselektiert werden.*

3.1.4.10 [Pflicht] Die separate (De)Selektion von Subtypen eines Typs muss möglich sein.*

3.1.4.11 [Wunsch] Eine Filterung der Typen über ihren QualifiedName kann über reguläre Ausdrücke möglich sein.*

3.1.4.12 [Wunsch] Der Benutzer kann durch einen Klick einen Typ ohne seine Subtypen auswählen.*

3.1.4.13 [Pflicht] Knoten und Kanten müssen anhand ihrer ID ausgewählt werden können. Z.B. durch Eingabe von „v1,e1,v4" in einem Textfeld.*

3.1.4.14 [Pflicht] Knoten und Kanten müssen über eine GReQL-Anfrage mit einem einzelnen Element oder einer Menge von Elementen als Ergebnis ausgewählt werden können.*

3.1.5 Brotkrumenleiste

3.1.5.1 [Pflicht] Die Historie aller besuchten Graphelemente muss für eine "Brotkrumenleiste" gespeichert werden.*

3.1.5.2 [Wunsch] Der in der "Brotkrumenleiste" dargestellte Teil der Historie kann begrenzt sein.*

3.1.5.3 [Pflicht] Beim Klick auf ein Element in der "Brotkrumenleiste" wird dieses wieder selektiert.*

3.1.6 Verwendung durch mehrere Benutzer

3.1.6.1 [Kann] Die gleichzeitige Verwendung des Browsers durch mehrere Benutzer mit jeweils eigenen Graphen muss möglich sein.*

3.1.6.2 [Kann] Vor dem Laden eines Graphen eines neuen Benutzers muss der Gesamtressourcenverbrauch abgeschätzt werden.

3.1.6.3 [Kann] Übersteigt der Gesamtressourcenverbrauch des Ladens eines Graphen die noch freien Ressourcen, muss das Laden verweigert werden.

3.1.6.4 [Kann] Wird die Sitzung eines Benutzers geschlossen, so müssen die durch diesen Benutzer verwendeten Ressourcen freigegeben werden.*

3.1.6.5 [Kann] Wird ein Graph eine gewisse Zeit nicht mehr verwendet, so müssen die durch diesen Graphen beanspruchten Ressourcen freigegeben werden.*

3.2 Nicht-funktionale Anforderungen

3.2.1 [Kann] Zur Generierung der 2D-Darstellung aus Anforderung 1.4.1.1 soll ein externer Graphlayouter verwendet werden (z.B. GraphViz-Tools).*

3.2.2 [Kann] Die Berechnung von Knoten- und Kantenmengen soll mit GReQL erfolgen.*

3.2.3 [Kann] Eine Blockade des Browsers durch das Layouten zu großer Teilgraphen soll verhindert werden.*

3.2.4 [Kann] Es sollen moderne Webtechniken (AJAX, etc.) zum Einsatz kommen.*

3.2.5 [Kann] Auf die Verwendung von Frame und Table sollte verzichtet werden.*

3.2.6 [Pflicht] Alle generierten Seiten müssen mit einem XHTML/CSS-Validator validiert werden.*

3.2.7 [Pflicht] Der TGraph-Browser muss mindestens auf den folgenden Webbrowsern verwendbar sein:
Microsoft Internet Explorer 7.0 und 8.0, Mozilla Firefox 3.0 und 3.5, Apple Safari*

3.2.8 [Pflicht] Bzgl. Programmierung, Dokumentation etc. gelten die üblichen Regeln der Softwaretechnik, die hier nicht weiter aufgeführt werden.*

Aufgrund der beschriebenen Anforderungen, ist ein System entwickelt worden, welches im nächsten Kapitel vorab beschrieben wird.

Kapitel 4

Bedienung

Nachdem die Anforderungen aufgelistet wurden, soll nun die Bedienung des *TGraphBrowsers* erläutert werden, um dem Leser zu verdeutlichen, wie das fertige System arbeitet.

Daher wird zunächst in Abschnitt 4.1 erklärt, wie ein Graph geladen werden kann. In Abschnitt 4.2 wird der Aufbau der gesamten HTML-Seite erläutert. In den folgenden beiden Abschnitten 4.3 und 4.4 wird auf die unterschiedlichen Darstellungsformen des Graphen eingegangen. Im Anschluss wird beschrieben, wie der Nutzer die dargestellte Menge an Kanten und Knoten verändern kann (Abschnitt 4.5). Abschließend wird auf die Brotkrumenleiste eingegangen (Abschnitt 4.6).

4.1 Laden eines Graphen

Um einen Graphen laden zu können, muss zunächst der Server gestartet werden. Sobald dieser läuft, kann man mittels eines Browsers Kontakt aufnehmen. Daraufhin erscheint die in Abb. 4.1 dargestellte HTML-Seite.

Sie stellt dem Nutzer drei unterschiedliche Alternativen zur Verfügung, wie der zu *ladende Graph bestimmt* werden kann:

1. Durch "Upload a graph:" hat der Nutzer die Möglichkeit, einen Graphen, der *lokal auf dem Rechner* des Clients vorliegt, auszuwählen. Dieser wird durch "Daten absenden" auf den Server hochgeladen.

2. Bei "Load from URL:" kann man einen Graphen durch Eingabe *seiner URL* auswählen.

3. Unter "Choose a graph from the server:" werden alle Graphen, die *beim Server vorliegen*, alphabetisch sortiert aufgelistet. Will der Nutzer einen Graphen löschen, so wird dies durch das rote "X" neber seinem Namen ermöglicht. Um einen Graph zu laden, klickt man auf dessen Namen.

Sollte der ausgewählte Graph die Größenbeschränkung des Servers übersteigen, so bekommt der Nutzer dies mitgeteilt und er kann einen anderen auswählen. Ansonsten erscheint ein Fortschrittsbalken. Er zeigt an, zu wie viel Prozent der Graph geladen wurde. Wurden 100 % erreicht, so erscheint die *tabellarische Darstellung*.

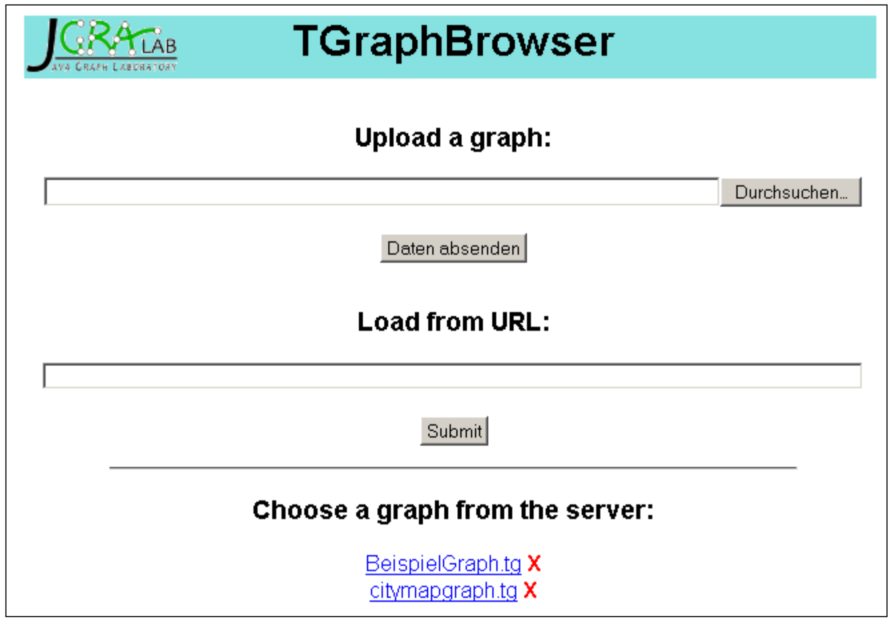


Abbildung 4.1: Die HTML-Seite, bei der man den zu ladenden Graphen auswählen kann.

4.2 Gesamtübersicht

Sobald ein Graph geladen wurde, erscheint im Browser eine HTML-Seite, wie sie in Abb. 4.2 zu sehen ist. Am oberen Rand befindet sich die Optionsleiste. Sie bietet die Möglichkeit einen anderen Graph zu laden und die Graphdarstellung anzupassen. Am rechten Rand befinden sich die Optionen, mit denen man Knoten und Kanten über ihre Ids auswählen sowie *GReQL*-Anfragen stellen kann. Unter der Optionsleiste befindet sich die Brotkrumenleiste. Mit ihrer Hilfe kann man zu Elementen, die man zuvor besucht hat, zurückkehren.

Der untere Bereich der Abbildung setzt sich aus der *Schema-* und der *Graphdarstellung* zusammen. Mit ersteren kann man die Typen deselektieren, deren Instanzen man nicht angezeigt haben möchte. Die Graphdarstellung zeigt den Graphen entweder in der *tabellarischen* oder der *grafischen Darstellung*.

4.3 Tabellarische Darstellung

Die *tabellarische Darstellung* des Graphen, wie man sie in Abb. 4.3 sehen kann, besteht aus zwei Teilen. Der obere besteht aus den Optionen "Show attributes" und "Show ... elements". Der untere Teil ist die Tabelle mit den beiden Schaltflächen "Vertices" und "Edges".

Aus Gründen der Übersichtlichkeit werden die Knoten und Kanten in *zwei verschiedenen Tabellen* dargestellt. Durch die beiden Schaltflächen "Vertices" und "Edges" kann man zwischen beiden Tabellen wechseln.

Jede Tabelle zeigt alle *relevanten Informationen* zu den Elementen eines Graphen an. Diese bestehen aus der Id, dem Schematyp, den Attributen und ihren Werten sowie den inzidenten Elementen. In der Spalte "Vertex" bzw. "Edge" steht der Schematyp des jeweiligen Knoten gefolgt von seiner eindeutigen Nummer. Sie ist tiefgestellt, um sie vom Namen des Typs unterscheiden zu können, der auch auf eine Ziffer enden kann. Die inzidenten Elemente werden in der Spalte "Incident edges" bzw. "Incident vertices" angezeigt. Falls die Option "Show attributes" gewählt wurde, existiert eine weiter Spalte "Attributes", in der die Attribute und ihre Werte zu sehen sind. Ansonsten werden diese Informationen beim Überfahren einer Tabellenzeile mit dem Cursor angezeigt.

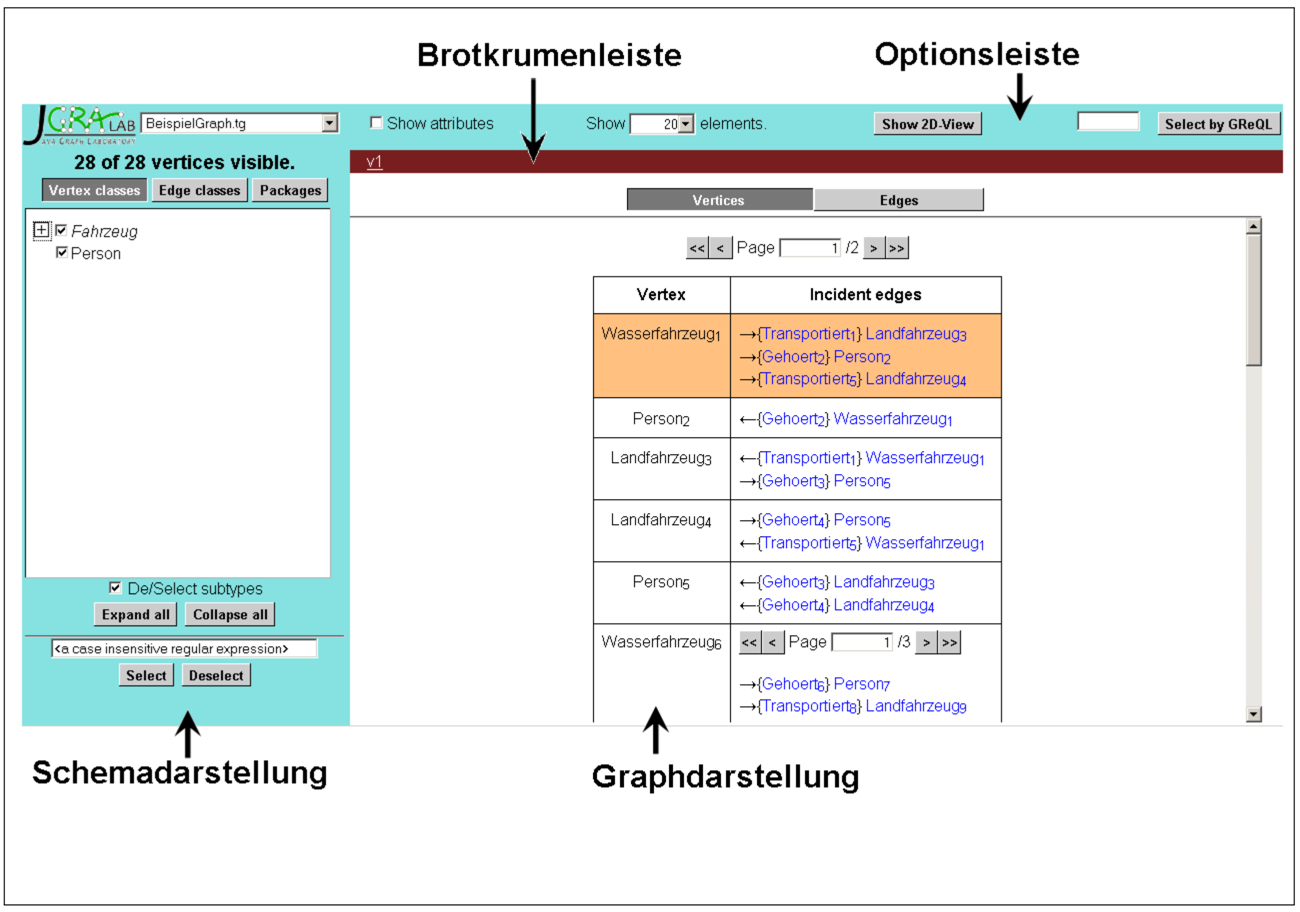


Abbildung 4.2: Die Gesamtübersicht.

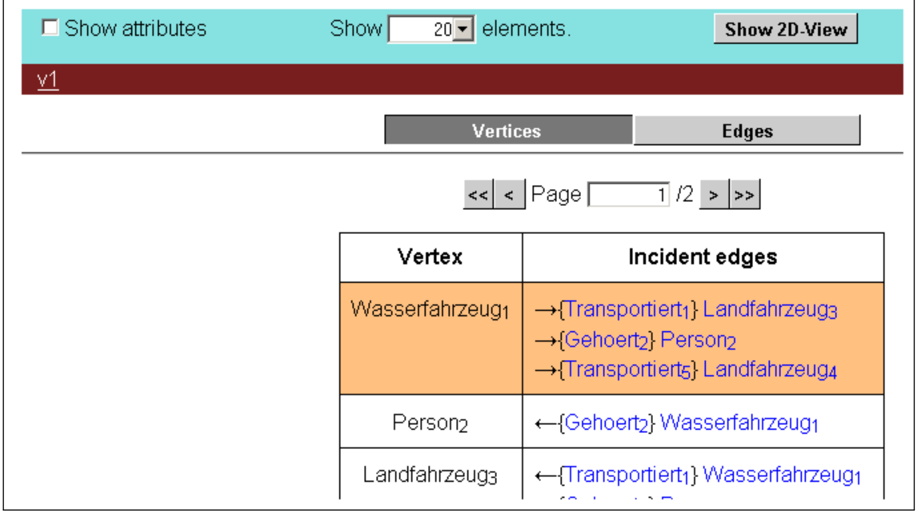


Abbildung 4.3: Die *tabellarische Darstellung*.

Betrachtet man sich die inzidenten Elemente des Knotens Wasserfahrzeug$_1$ aus Abb. 4.3, so stellt man fest, dass es drei ausgehende Kanten gibt, die durch $\rightarrow$ gekennzeichnet sind. Person$_2$ hingegen besitzt nur eine eingehende Kante. In den geschweiften Klammern hinter den Pfeilen steht der Typ der Kante, gefolgt von ihrer tiefergestellten Nummer. Nach den Klammern folgt der adjazente Knoten.

Sollte ein Element mehr als 10 Inzidenzen[1] haben, so *erscheint eine Navigationsleiste*, wie man sie in Abb. 4.4 sehen kann. Durch die Schaltflächen “<<” und “>>” gelangt man zu der ersten bzw. letzten Seite mit Inzidenzen. Mit “<” gelangt man zur vorangegangenen und mit “>” zur nachfolgenden. Alternativ kann man in das Textfeld auch direkt die Seitennummer, zu der man gelangen will, eingeben.

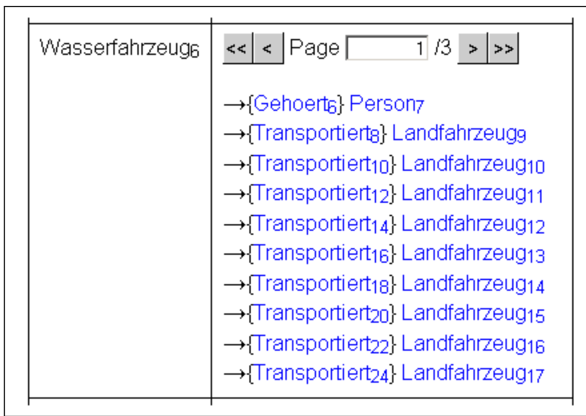


Abbildung 4.4: Die Navigationsleiste der inzidenten Kanten.

Wie man in der Abb. 4.3 sehen kann, ist der Knoten Wasserfahrzeug$_1$ *markiert*. Dies bedeutet, dass er das *aktuell gewählte Element* ist. Um durch den Graphen zu navigieren, klickt man auf ein Element in der Spalte “Incident edges” bzw. “Incident vertices”, so wird das gewählte zum neuen aktuellen Element. Wählt man beispielsweise die Kante Gehoert$_2$, so wird die Tabelle mit den Kanten angezeigt. Des Weiteren wird die Zeile, in der sich e1 befindet, markiert und die Anzeige auf sie zentriert.

Die *Anzahl der Elemente*, die *auf einer Tabellenseite* dargestellt werden, kann man mit der Option “Show ... elements” bestimmen. Hierbei hat der Nutzer die Wahl zwischen 10, 20, 50, 100, 200, 500 und 1000. Sollte ein Graph mehr Elemente haben, als auf eine Tabellenseite passen, so wird eine Navigationsleiste, wie in Abb. 4.3 zu sehen, über der Tabelle eingeblendet. Ihre Funktionsweise ist dabei analog zu der für die inzidenten Elemente.

Durch die Schaltfläche “Show 2D-View” kann man zur *grafischen Darstellung* wechseln. Dabei bleibt das aktuelle Element ausgewählt.

4.4 Grafische Darstellung

Nachdem man zur *grafischen Darstellung* gewechselt hat, erscheint der Graph so, wie in Abb. 4.5 zu sehen ist. Am oberen Rand der Abbildung sind die Optionen “Show attributes” und “Size of environment:” zu sehen.

[1]Dies ist der Default-Wert. Beim Starten des Servers kann ein anderer Wert angegeben werden. Wie das geht, ist im Anhang B zu lesen.

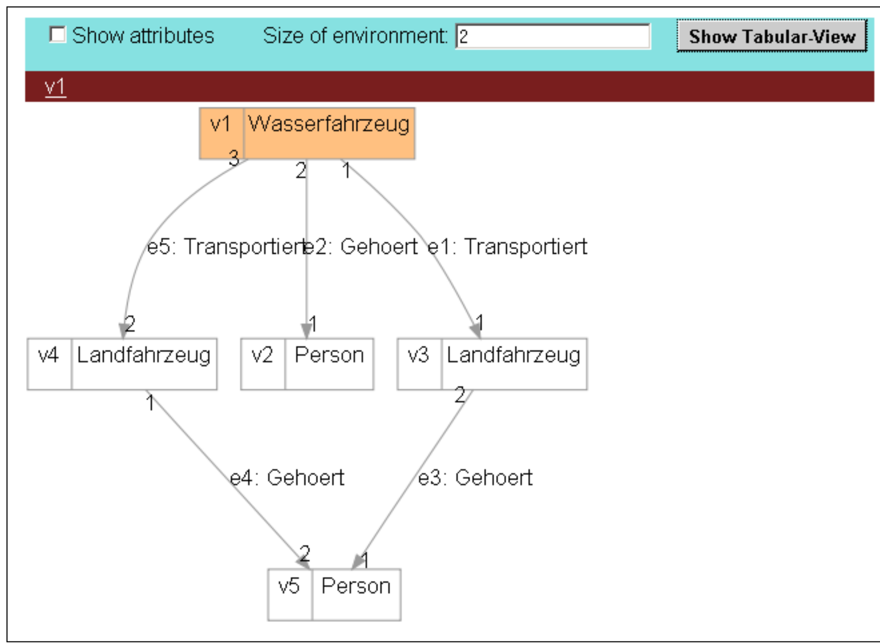


Abbildung 4.5: Die *grafische Darstellung*.

Die Knoten werden *durch Rechtecke dargestellt*, die aus zwei Feldern bestehen. Im linken steht die jeweilige Id und im rechten der Schematyp. Sollte die Option “Show attributes” gewählt sein, so erscheint unterhalb der beiden Felder ein drittes, in dem die Attribute und ihre Werte aufgelistet werden. Sollte die Option abgewählt sein, so erscheinen diese Informationen beim Überfahren mit dem Mauscursor in einem *Tooltip*.

Im Gegensatz zu den Knoten werden die *Kanten durch Pfeile dargestellt*. Die Pfeilspitze symbolisiert ihre Richtung. An jeder Kante steht ihre Id gefolgt von ihrem Schematyp. Ihre Attribute mit den entsprechenden Werten werden darunter aufgelistet, falls die Option “Show attributes” gesetzt wurde. Ansonsten erscheinen sie beim Überfahren mit dem Mauscursor. Darüber hinaus befindet sich am Anfangs- und Endknoten eine Nummer. Diese ist der Index in der Sequenz der inzidenten Kanten des jeweiligen Knotens.

Die Darstellung des gesamten Graphen ist in der Regel nicht möglich, da die Grafik zu groß würde. Aus diesem Grund wird nur die *Umgebung*[2] *des aktuellen Elements* dargestellt. Sie besteht aus allen Knoten und Kanten, die durch einen *Pfad* mit Maximallänge 2 erreichbar sind. Die Länge kann dabei vom Nutzer über die Option “Size of environment:” angepasst werden. Eine gestrichelte Kante an einem Knoten signalisiert, dass es weitere Kanten gibt, die wegen der Beschränkung der *Pfadlänge* nicht angezeigt werden.

Um durch den *Graph zu navigieren*, klickt man auf ein dargestelltes Element. Daraufhin wird es durch eine Veränderung der Farbe als aktuelles Element markiert und seine Umgebung angezeigt. So hebt sich beispielsweise der Knoten `v1` in Abb. 4.5 durch seine Farbe von den anderen ab. Dies signalisiert, dass es sich bei `v1` um das aktuelle Element handelt.

Durch die Schaltfläche “Show Tabular-View” wechselt man zur *tabellarischen Darstellung*.

[2]Die genaue Definition der *Umgebung* eines Graphelements ist in Abschnitt 2.1 zu finden.

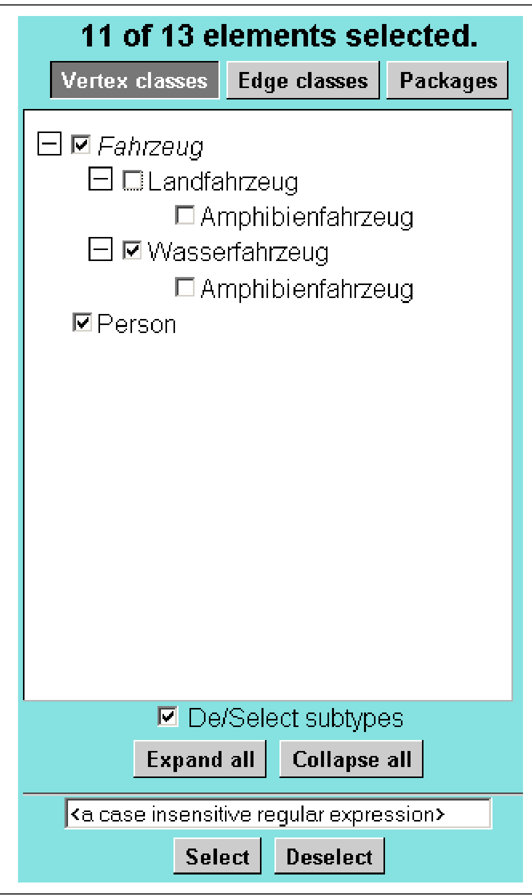


Abbildung 4.6: Die *Schemadarstellung*.

4.5 Auswahl der dargestellten Elemente

Das Betrachten von Graphen mit mehreren tausend Elementen kann unübersichtlich werden. Um dies zu vermeiden, gibt es mehrere Möglichkeiten, wie sich der Nutzer *gewünschte Knoten und Kanten anzeigen* lassen kann bzw. wie er *unerwünschte ausblenden* kann. So lassen sich die Graphelemente nach ihren Typen filtern, was in Unterabschnitt 4.5.1 erklärt wird. Darüber hinaus kann der Nutzer die gewünschten Knoten und Kanten mittels ihrer Ids angeben (siehe hierzu Unterabschnitt 4.5.2). Abschließend besteht noch die Möglichkeit die anzuzeigenden Elemente durch eine *GReQL*-Anfrage zu bestimmen (Unterabschnitt 4.5.3).

4.5.1 Auswahl nach den Schematypen

Damit man die angezeigten Graphelemente *nach ihren Schematypen filtern* kann, gibt es eine *Schemadarstellung*, die in Abb. 4.6 zu sehen ist. Um die Übersichtlichkeit zu erhöhen, wird die Generalisierungshierarchie der Knoten- und Kantentypen getrennt voneinander dargestellt. Zusätzlich gibt es noch eine Sicht, in der die Paketstruktur des Schemas dargestellt wird. Über die Schaltflächen "Vertex classes", "Edge classes" und "Packages" kann zwischen den einzelnen Darstellungen gewechselt werden.

Darunter befindet sich die Repräsentation der Schematypen. In diesem Fall sind es die Knotentypen. Jedes Element des Schemas wird dabei *durch seinen UniqueName*[3] *repräsentiert*. Zeigt man mit

[3]Der *UniqueName* eines Schematyps ist dessen *SimpleName*, solange es keinen zweiten Typ mit diesem Namen gibt. Ansonsten ist es der *qualifizierte Name*.

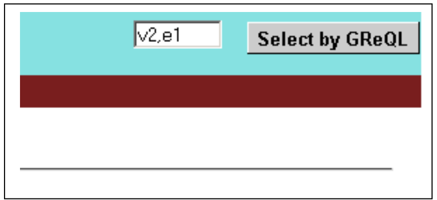


Abbildung 4.7: Die Komponenten zur Angabe einer Graphelementmenge.

dem Cursor auf ihn, so erscheint der *voll qualifizierte Name*. Zusätzlich werden noch seine Attribute mit den jeweiligen Wertebereichen angezeigt. Sollte es sich um einen abstrakten Typ handeln, so wird sein Name kursiv dargestellt. Dies kann man in Abb. 4.6 an `Fahrzeug` erkennen.

Um die Generalisierungshierarchie zu repräsentieren, werden die direkten *Subtypen eines Typs eingerückt* dargestellt. Im Falle von Mehrfachvererbung wird ein Knotentyp *unter jedem seiner Supertypen* aufgeführt. So hat beispielsweise `Landfahrzeug` aus Abb. 4.6 den Untertyp `Amphibienfahrzeug`. Da er noch einen weiteren Supertyp nämlich `Wasserfahrzeug` hat, wird er auch unter diesem aufgeführt.

Um sich die direkten Subtypen anzeigen zu lassen, genügt ein Klick auf das vorangestellte "+". Durch ein erneutes Betätigen verschwinden sie wieder. Durch die Schaltfläche "Expand all" werden alle Subtypen aller Knoten- bzw. Kantentypen angezeigt. Um sich nur die erste Ebene der Generalisierungshierarchie anzeigen zu lassen, klickt man auf "Collapse all".

Vor jedem Typ befindet sich eine Checkbox. Wird sie deselektiert, so werden *alle Instanzen dieses Typs* aus der *tabellarischen* bzw. *grafischen Darstellung des Graphen entfernt*. Um sie wieder anzuzeigen, muss der Typ wieder selektiert werden. Wurde die Option "De/Select Subtypes" ausgewählt, so werden alle Subtypen automatisch mit (de)selektiert. Taucht ein Schematyp in der Darstellung mehrfach auf, so werden bei der An- oder Abwahl eines dieser Vorkommen alle mit an- bzw. abgewählt. So wurde in Abb. 4.6 `Landfahrzeug` deselektiert. Da "De/Select subtypes" ausgewählt ist, wurde der Subtyp `Amphibienfahrzeug` ebenfalls abgewählt. Da er noch einen zweiten Supertyp hat, wird auch das Vorkommen unter `Wasserfahrzeug` deselektiert.

Eine andere Möglichkeit der (De)Selektion besteht in der *Angabe eines regulären Ausdrucks*, der im Textfeld, welches am unter Rand der Abbildung zu sehen ist, eingegeben werden kann. Durch einen Klick auf "Select" werden alle Schematypen, deren *qualifizierter Name* mit dem *regulären Ausdruck* matched, selektiert. Dabei wird die Groß- und Kleinschreibung ignoriert. Durch "Deselect" werden die entsprechenden Schematypen deselektiert.

4.5.2 Auswahl durch die Ids der Elemente

Neber der Filterung nach den Schematypen kann der Nutzer auch die Ids der für ihn *interessanten Graphelemente angeben*. Hierfür ist das Eingabefeld, welches auf der linken Seite der Abb. 4.7 zu sehen ist, gedacht. Wird nur eine Id angegeben, so wird der gewünschte Knoten bzw. die gewünschte Kante zum aktuellen Element der Graphdarstellung. Gibt man mehrere Ids an, so kann jedes nicht-alphanumerische Zeichen als Separator verwendet werden. Die durch die angegebenen Ids identifizierbaren Knoten und Kanten werden in der Graphdarstellung angezeigt. Sollte es Identifier geben, denen keine Elemente des Graphen zugeordnet werden können, so wird der Nutzer darüber informiert.

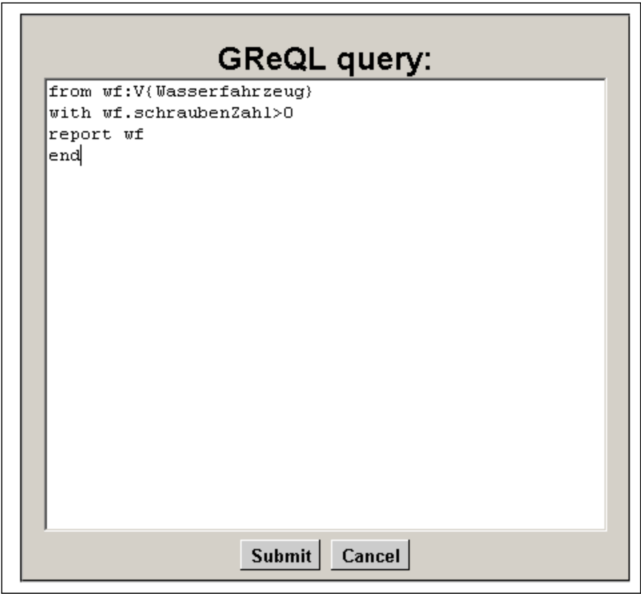


Abbildung 4.8: Box zur Eingabe einer GReQL-Anfrage.

Abbildung 4.9: Die Brotkrumenleiste.

4.5.3 Auswahl durch eine GReQL-Anfrage

Neber der expliziten Angabe der darzustellenden Graphelemente, kann der Nutzer auch eine *GReQL-Anfrage verwenden*. Hierfür klickt er auf die Schaltfläche "Select by GReQL", die sich auf der rechten Seite der Abb. 4.7 befindet. Daraufhin erscheint eine Box, in der die *GReQL*-Anfrage angegeben werden kann, wie in Abb. 4.8 zu sehen. Das Ergebnis der Anfrage darf nur aus Knoten und Kanten bestehen. Durch die Schaltfläche "Submit" wird die Anfrage ausgeführt. Sollte es zu einem Fehler kommen, so wird dieser angezeigt.

4.6 Brotkrumenleiste

Wenn der Nutzer durch den Graphen navigiert, kommt es vor, dass er zu einem bereits besuchten Graphelement zurückkehren will. Um dies zu ermöglichen, wird für jede Kante und jeden Knoten, auf die bzw. den geklickt wird, ein *Eintrag in der Brotkrumenleiste* erzeugt. Dieser besteht aus der Id des entsprechenden Graphelements. Des Weiteren werden Mengen aus Knoten und Kanten, die durch den Nutzer angegeben wurden[4], ebenfalls in der Brotkrumenleiste vermerkt. Sie werden, wie man in Abb. 4.9 sehen kann, beispielsweise durch `{v1,v6}` dargestellt. Sollte die Menge mehr als drei Elemente umfassen, so wird ihre Existenz durch `,...` angedeutet. Die Reihenfolge der Einträge entspricht der Reihenfolge, in der sie durch den Nutzer erzeugt wurden. Dabei ist der Eintrag, der sich am weitesten links befindet, der älteste. Sollte es mehr als 10 Einträge geben, so erscheinen drei Punkte, mit denen man sich die vorangegangenen anzeigen lassen kann.

Um zu einem vorangegangenen Graphelement zurückzukehren, genügt ein *Klick auf den entsprechenden Eintrag* der Brotkrumenleiste. Daraufhin wird der gewählte Knoten bzw. die gewählte

[4] siehe hierzu Unterabschnitt 4.5.2 und 4.5.3

Kante zum aktuellen Element der Graphdarstellung. Alle Einträge rechts von ihm werden grau dargestellt. Dies ist in Abb. 4.9 zu sehen, wo `v5` gewählt wurde. Wird nun auf einen Knoten oder eine Kante in der Graphdarstellung geklickt, so werden die grau dargestellten Einträge gelöscht.

Nachdem nun erklärt wurde, wie der *TGraphBrowser* bedient wird, handelt das folgende Kapitel von dem Aufbau des Servers sowie dem Aufbau der Anfragen und Antworten.

Kapitel 5

Entwurf

In diesem Kapitel wird in den Abschnitten 5.1 und 5.2 zunächst erklärt, wie der Server des *TGraphBrowsers* aufgebaut ist. Der anschließende Abschnitt 5.3 behandelt, wie die Anfragen des Clients aussehen müssen, damit der gewünschte Methodenaufruf beim Server stattfindet. Des Weiteren wird der Aufbau seiner Antworten erklärt.

5.1 Architektur des Servers

Wie man in Abb. 5.1 erkennen kann, besteht der Server aus drei Schichten, die sich gegenseitig aufrufen und zwischen denen Daten ausgetauscht werden. Die erste Schicht heißt `Server`, die zweite `State-Repository` und die dritte `Visualizer`. Die ersten beiden bestehen aus je einer gleichnamigen Komponente. Jedoch die dritte Schicht setzt sich aus dem `2D-Visualizer`, dem `Tabular-Visualizer` und dem `Schema-Visualizer` zusammen.

Da die Notwendigkeit besteht, die *Anfragen der Clients entgegenzunehmen* und ihnen Antworten zurückzuschicken, wurde die Komponente `Server` geschaffen.

`State-Repository` existiert, damit die *Zustände* der beim Client angezeigten Webseite an einem Ort *gespeichert* werden können. Des Weiteren stellt sie die *Methoden zur Verfügung*, die durch HTTP-Anfragen angesprochen werden können. Zu ihrer Abarbeitung nutzt sie die Zustände, sowie die `Visualizer`-Komponenten. Das Ergebnis eines Aufrufs reicht sie an `Server` weiter.

Um die Wartbarkeit und die Verständlichkeit des Programms zu erhöhen, ist es sinnvoll die *tabellarische* und die *grafische Darstellung* durch *jeweils eine eigene Komponente* zu realisieren nämlich `Tabular-Visualizer` und `2D-Visualizer`. Zusätzlich gibt es noch den `Schema-Visualizer`, der die *Repräsentation der Generalisierungshierarchie* der Schematypen erzeugt. Da die drei zuletzt erwähnten Komponenten mit der Visualisierung zu tun haben, werden sie in `Visualizer` zusammengefasst.

Da sich die Nutzer aus einer Sammlung von Graphen auf dem Server einen zum Anzeigen auswählen können, benötigt man einen *Bereich auf der* serverseitigen *Festplatte*, in dem die zur Auswahl stehenden Graphen liegen. Dies ist der `Workspace`. Graphen, die von der lokalen Festplatte des Clients oder durch die Angabe einer URI hochgeladen wurden, werden ebenfalls im `Workspace` abgelegt. Dadurch wird verhindert, dass der selbe Graph mehrmals zum Server übertragen werden muss.

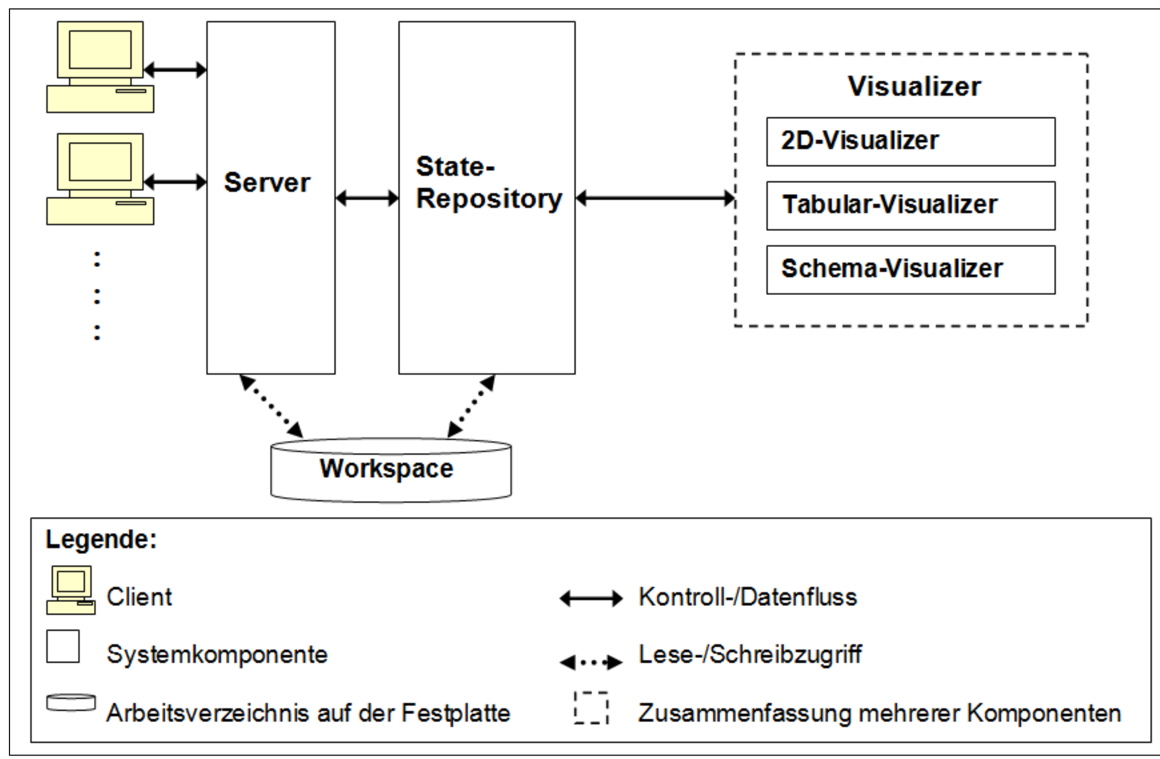


Abbildung 5.1: Systemarchitektur

5.2 Arbeitsweise der einzelnen Komponenten

Nachdem alle Komponenten des Servers eingeführt wurden, wird im Folgenden ihre Arbeitsweise kurz umrissen:

Server: Der `Server` kümmert sich um die *Kommunikation mit den Clients.* Um diese Aufgabe zu erfüllen gibt es die beiden Threads `TGraphBrowser` und `RequestThread`. Ersterer empfängt die Anfragen und schickt die Antworten zurück. `RequestThread` hingegen interpretiert die Anfragen und arbeitet sie ab.
Die HTTP-Antworten bestehen aus dem vom `State-Repository` gelieferten JavaScript-Code, der vom `Server` mit einem dem HTTP-Protokoll entsprechenden Header versehen wird. Des Weiteren bestimmt der `Server` den Speicherverbrauch des zu ladenden Graphen und informiert den Nutzer, falls das Laden die Größenbeschränkung des ihm zugewiesenen Speichers übersteigen würde.

State-Repository: Das `State-Repository` speichert den Zustand der Webseite, die der Client betrachtet. Zu ihm gehören beispielsweise die *Navigations-Historie* für die "Brotkrumenleiste" und der geladene Graph. Des Weiteren erzeugt das `State-Repository` mit Hilfe der `Visualizer`-Komponenten den JavaScript-Code, der dem `Server` geliefert wird.

2D-Visualizer: Der `2D-Visualizer` liefert eine *grafische Repräsentation* der ausgewählten Kanten und Knoten sowie ihrer *Umgebung*. Diese wird so erzeugt, dass sie in eine Webseite eingefügt werden kann. Während der Erzeugung wird der externe Graphlayouter *Graphviz* verwendet.

Tabular-Visualizer: Der `Tabular-Visualizer` liefert eine tabellarische Darstellung der Knoten und Kanten eines Graphen.

Schema-Visualizer: Der `Schema-Visualizer` erzeugt eine Baumdarstellung des Schemas, die die Spezialisierungshierarchie der Schematypen widerspiegelt.

5.3 Aufbau der Nachrichten zwischen Browser und Server

Die Nachrichten, die zwischen dem Server und dem Browser ausgetauscht werden, müssen einen *festgelegten Aufbau* haben, damit sie korrekt interpretiert werden. Der *TGraphBrowser* verwendet zwei unterschiedliche Anfragetypen: die GET- und die POST-Anfrage. Erstere enthält alle Nutzdaten in der ersten Zeile. Ihr Aufbau wird in Unterabschnitt 5.3.1 erklärt. Da die Menge der Nutzdaten einer GET-Anfrage begrenzt ist, benötigt man zusätzlich noch die POST-Anfrage. Bei ihr befinden sich die Information im Body, dessen Aufbau in Unterabschnitt 5.3.2 erklärt wird. Wie die Antwort des Servers aufgebaut ist, erfährt man in Unterabschnitt 5.3.3. Zum Abschluss wird in Unterabschnitt 5.3.4 ein konkretes Beispiel einer Anfrage sowie einer Antwort gegeben.

5.3.1 GET-Anfrage

Damit der Server die Anfrage korrekt verarbeiten kann, muss sie ein bestimmtes Format haben:

GET **/methodName?param1=value1¶m2=value2&...** *HTTP/1.1*

Der kursive Teil wird automatisch erzeugt und wird hier nicht näher betrachtet, da er vom HTTP-Protokoll gefordert wird und bei jeder GET-Anfrage gleich ist. Im Gegensatz dazu muss der fett geschriebene Teil explizit im Programm angegeben werden. Dieser setzt sich aus dem Methodennamen und den Parametern, die durch ein `?` voneinander getrennt werden, zusammen. Letztere werden durch ihren Namen gefolgt von `=` und ihrem Wert angegeben. Sollte eine Methode mehrere Parameter benötigen, so werden diese durch ein `&` voneinander getrennt.

So bedeutet beispielsweise *GET* **/checkLoading?sessionId=4** *HTTP/1.1*, dass die Methode `checkLoading` aufgerufen wird. Sie hat einen Parameter namens `sessionId`, der den Wert "4" hat.

5.3.2 POST-Anfrage

Im Gegensatz zur GET-Anfrage befinden sich bei der POST-Anfrage alle benötigten Daten im Body. Sein Aufbau lässt sich wie folgt definieren:

```
Body ::= <Timestamp> '\n' <SessionId> '\n' <Methodenname>
         {'\n' Parameter};
```

`Timestamp` ist der Zeitpunkt, zu dem der Server die letzte Antwort geschickt hat. Diese Information ist notwendig, damit erkannt werden kann, ob sich die Anfrage auf eine gültige *Session*[1] bezieht. Was das bedeutet, wird anhand eines Beispiels erklärt: Betrachtet ein Nutzer einen Graphen, so wurde beim vorangegangenen Laden eine neue *Session* initiiert. Dabei wurde ein neuer Zustand angelegt. Wird nun der Server neugestartet, geht dieser verloren. Deselektiert der Nutzer nun einen Schematyp, so schickt der Browser eine Anfrage. Diese muss abgewiesen werden, da beim Server der entsprechende Zustand nicht mehr vorhanden ist und die *Session* demnach ungültig ist.

`SessionId` ist eine Zahl, die der Server benötigt, um die Zustandsinformationen der im Browser angezeigten Seite zu erhalten. `Methodenname` gibt den Namen der Methode an, die ausgeführt werden soll. Falls sie aktuelle `Parameter` benötigt, werden diese in den nachfolgenden Zeilen angegeben.

[1] Als *Session* wird eine stehende Verbindung zwischen Server und Client bezeichnet. Dabei bedeutet eine stehende Verbindung, dass beim Server Daten vorliegen, die einem Client eindeutig zugeordnet werden können [Ses10].

5.3.3 Aufbau der Antwort

Damit der Browser ohne großen Aufwand die Antwort des Servers verarbeiten kann, besteht die Antwort aus einem *JSON*-String. Bei *JSON (**J**avaScript **O**bject **N**otation)* handelt es sich um ein Datenaustauschformat. Es baut auf die Strukturen der *Dictionarys* und der *Arrays* auf [JSO09]. In dieser Bachelorarbeit wird allerdings nur erstere verwendet.

Der gesendete *JSON*-String hat dabei die Form:

```
JSON-String ::= '{ "method": ' <Funktion> ' }';
```

Als `Funktion` wird eine anonyme JavaScript-Funktion, die aus dem generierten Code besteht, angegeben.

5.3.4 Beispiel

Der Aufbau einer Anfrage sowie der entsprechenden Antwort soll nun anhand eines konkreten Beispiels gezeigt werden. Die dargestellten Nachrichten entstehen, wenn man den Graph "BeispielGraph" lädt.

```
1  POST /loadGraphFromServer?path=D:/graphen/BeispielGraph.tg HTTP/1.1
2  Host: localhost User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; de;
      rv:1.9.1.7) Gecko/20091221 Firefox/3.5.7 (.NET CLR 3.5.30729)
3  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
4  Accept-Language: de-de,de;q=0.8,en-us;q=0.5,en;q=0.3
5  Accept-Encoding: gzip,deflate
6  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
7  Keep-Alive: 300
8  Connection: keep-alive
9  Content-Type: text/plain; charset=UTF-8
10 Referer: http://localhost/loadGraphFromServer?path=D:/graphen/
      BeispielGraph.tg
11 Content-Length: 28
12 Pragma: no-cache
13 Cache-Control: no-cache
14
15 1264182477353
16 0
17 checkLoading
```

Listing 5.1: Eine POST-Anfrage.

Listing 5.1 zeigt eine POST-Anfrage, die an den Server geschickt wird. Dabei zeigen die Zeilen 1 bis 13 den vom Browser automatisch erzeugten Header. Der `Timestamp` der Anfrage ist in Zeile 15 zu erkennen. In der folgenden befindet sich die `SessionID`. Die aufzurufende Methode heißt `checkLoading` (Zeile 17).

```
1 HTTP/1.0 200 OK
2 Date: Fri Jan 22 18:54:51 CET 2010
3 Server: JibbleWebServer/1.0
4 Content-Type: text/html
5 Expires: Thu, 01 Dec 1994 16:00:00 GMT
6 Last-modified: Fri Jan 22 18:54:51 CET 2010
7
8 { "method": function(){
9 document.getElementById("loadBarForeground").style.width = "0px";
```

```
10 document.getElementById("loadBarNumber").innerHTML = "10␣%";
11 loadId = window.setTimeout("checkLoad()", 1000);
12 timestamp = 1264182889504;
13 } }
```

Listing 5.2: Antwort des Servers.

Listing 5.2 zeigt die Antwort des Servers auf die POST-Anfrage aus Listing 5.1. Im Body (Zeile 8 bis 13) befindet sich das *JSON*-Objekt, welches die Aktualisierung des Fortschrittsbalken im Browser bewirkt.

Nachdem nun der Aufbau des Servers und der Nachrichten geklärt ist, wird im nächsten Kapitel auf die Implementierung des *TGraphBrowsers* eingegangen.

Kapitel 6

Implementation

In diesem Kapitel geht es darum, wie der *TGraphBrowser* implementiert ist. Hierzu wird zunächst in Abschnitt 6.1 erklärt, wie die Kommunikation zwischen Browser und Server von statten geht. Im Anschluss wird im Abschnitt 6.2 darauf eingegangen, wie die Zustände beim Server behandelt werden. Darüber hinaus wird erklärt, aus welchen Daten diese bestehen. Wie der Graph geladen wird, zeigt Abschnitt 6.3. Mit der Erzeugung der *tabellarischen Darstellung* und wie man mit ihr durch den Graph navigieren kann, beschäftigt sich Abschnitt 6.4. Das grafische Pendant wird im folgenden Abschnitt 6.5 erklärt. Im Anschluss wird auf die unterschiedlichen Arten, wie man die Menge der dargestellten Graphelemente auswählen kann, eingegangen (Abschnitt 6.6). Zum Abschluss wird erklärt, wie die Brotkrumenleiste realisiert wurde (Abschnitt 6.7).

6.1 Kommunikation zwischen Browser und Server

In diesem Abschnitt wird zunächst erklärt, wie der Browser GET- und POST-Anfragen an den Server schickt (Unterabschnitt 6.1.1). Nachdem eine empfangen wurde, wird sie verarbeitet und eine Antwort zurückgeschickt. Die Implementation dieser Funktionalität wird in Unterabschnitt 6.1.2 behandelt. Zum Abschluss wird beschrieben, wie der Browser die Antwort verarbeitet (Unterabschnitt 6.1.3).

6.1.1 Senden einer Anfrage

Die Anfragen, die bei der Nutzung des *TGraphBrowsers* anfallen, lassen sich in zwei Kategorien unterteilen: die automatisch gesendeten und die mittels des *XMLHttpRequests* verschickten. Bei den zuerst genannten, handelt es sich um GET-Anfragen. Sie entstehen, wenn der Browser *versucht HTML-Seiten zu laden* oder benötigte Grafiken anzuzeigen. Die Anfragen der zweiten Kategorie werden durch JavaScript-Befehle verschickt.

```
1  /*
2   * Sends an AJAX-GET-Request to the Webserver with the path args.
3   * The response is evaluated in evaluateResponse.
4   *
5   * @param args the url of the GET-Request
6   * @param timeout if true there is a timout after 60 sec.
7   */
8  function sendGetRequest(args, timeout){
9   http=new XMLHttpRequest();
10  http.open("GET", args, true);
```

```
11    http.onreadystatechange = evaluateResponse;
12    http.send(null);
13    if(timeout){
14      id = window.setTimeout("cancel()", 60000);
15    }else{
16      id = null;
17    }
18  }
```

Listing 6.1: Senden einer GET-Anfrage mit *AJAX*.

In Listing 6.1 ist die JavaScript-Funktion `sendGetRequest` aufgeführt. Mit ihr kann der Browser eine *asynchrone GET-Anfrage* zum Server *senden*. Durch den Aufruf der Methode `open` in Zeile 10 wird `args` in den GET-Header eingefügt. Durch Zeile 12 wird die Anfrage schließlich an den Server geschickt.

So wie das *XMLHttpRequest*-Objekt implementiert ist, besteht für den Browser nach dem Abschicken der Anfrage keine Möglichkeit festzustellen, ob er noch eine Antwort vom Server erhalten wird oder ob ein Fehler bei der Übertragung aufgetreten ist. Aus diesem Grund wird in Zeile 14 ein *Timeout* gesetzt, der nach 60 Sekunden die Methode `cancel` ausführt. Bei Ausführung dieser Methode wird der Benutzer gefragt, ob er weitere 60 Sekunden auf eine Antwort des Servers warten möchte oder ob die Verbindung abgebrochen werden soll.

```
1   /*
2    * Sends a AJAX-Post-Request to the Webserver.
3    * The request consists of:
4    * timestamp
5    * sessionId
6    * method
7    * params
8    *
9    * The response is evaluated in evaluateResponse.
10   *
11   * @param method the method name of Java
12   * @param the correct formatted params (each in an extra line)
13   * @param timeout if true there is a timout after 60 sec.
14   */
15  function sendPostRequest(method, params, timeout){
16        if(id==null){
17              http=new XMLHttpRequest();
18              http.open("POST", "␣", true);
19              http.onreadystatechange = evaluateResponse;
20              http.setRequestHeader("Content-Type","text/plain");
21              var content = timestamp+"\n"+sessionId+"\n"+method+(params!=
                   null?"\n"+params:"");
22              http.send(content);
23              if(timeout){
24                    document.getElementsByTagName("body")[0].style.cursor =
                         "wait";
25                    id = window.setTimeout("cancel()", 60000);[...]
26              }else{
27                    id = null;
28              }
29        }
30  }
```

Listing 6.2: Senden einer POST-Anfrage mit *AJAX*

Listing 6.2 zeigt, wie eine POST-Anfrage gesendet wird. Im Gegensatz zu einer GET-Anfrage muss die Headerinformation `Content-Type` auf einen Wert gesetzt werden, die dem Server mitteilt, welche Art von Daten im Body enthalten sind. Dies geschieht in Zeile 20. In den folgenden Zeilen werden die zu übertragenden Informationen in den Body eingefügt und die Anfrage gesendet.

6.1.2 Verarbeitung beim Server

Erreicht die Anfrage den Server, so wird zunächst überprüft, ob eine HTML-Seite oder Grafik angefragt wird. Ist eines von beiden der Fall, wird die gewünschte Datei an den Browser zurückgeschickt. Ansonsten handelt es sich um einen Methodenaufruf. Zunächst wird der Methodenname sowie die aktuellen Parameter extrahiert. Im Anschluss wird per *Reflection* die *entsprechende Methode* des `StateRepositorys` aufgerufen. Dies geschieht durch die Methode `callMethod`. Sie ist in Listing 6.3 dargestellt.

```
1  /**
2    * Calls the received Method.
3    * [...]
4    * @param methodname
5    * name of the method
6    * @param args
7    * parameters of the method[...}
8    * @return result of method invocation
9    */
10 private StringBuilder callMethod(BufferedOutputStream out,
11                   String methodname, String[] args) throws IOException {
12       // get method
13       Method method = StateRepository.definedMethods.get(methodname);
14       if (method == null) {
15             sendErrorMessage(out, "There␣does␣not␣exist␣a␣method␣with␣name
                  ␣" + methodname + ".");
16             return new StringBuilder();
17       }
18       // check parameters
19       Class<?>[] params = method.getParameterTypes();
20       Object[] currentParams = new Object[params.length];
21       for (int i = 0; i < params.length; i++) {
22             if (params[i] == Integer.class) {
23                   currentParams[i] = Integer.parseInt(args[i]);
24             } else if (params[i] == Boolean.class) {
25                   currentParams[i] = Boolean.parseBoolean(args[i]);
26             } else {
27                   currentParams[i] = args[i];
28             }
29       }
30       try { // call method
31             return (StringBuilder) method.invoke(rep, currentParams);
32       } catch [...]
33       return null;
34 }
```

Listing 6.3: Verarbeitung der empfangenen Daten beim Server.

Der extrahierte Methodenname wird durch den Parameter `methodname` übergeben und die aktuellen Parameter als String-Array `args`. Bereits beim Starten des Servers wurde im `StateRepository` eine HashMap `definedMethods` angelegt. Sie mapped den Namen einer

Methode auf das entsprechende Method-Objekt. Dadurch verkürzt sich das Auffinden der durch `methodname` angegebenen Methode auf einen einzigen Zugriff auf die HashMap (siehe Z. 13). In den Zeilen 19 bis 29 werden alle Parameter zu den jeweiligen Parametertypen der aufzurufenden Methode gecastet. Dadurch können fehlerhafte Parametertypen bereits vor der Ausführung abgefangen werden. Im Anschluss wird die entsprechende Methode ausgeführt und der generierte JavaScript-Code zurückgegeben.

Der generierte Code wird dann mit dem HTTP-Header einer Antwort versehen und zum Browser zurückgeschickt.

6.1.3 Abarbeiten der Antwort

Empfangene HTML-Seiten und Grafiken stellt der Browser automatisch dar. Anders sieht dies aus, falls eine Anfrage mittels *XMLHttpRequest* verschickt wurde. Hierfür wird eine JavaScript-Funktion benötigt. Die vom TGraphBrowser verwendete ist in Listing 6.4 zu sehen.

```
1  /*
2   * Evaluates the response.
3   */
4  function evaluateResponse() {
5    if (http.readyState == 4) {
6      if(id!=null){
7        // delete Timeout
8        window.clearTimeout(id);
9      }
10     // evaluate result
11     var response = eval("("+http.responseText+")");
12     response.method();
13   }
14 }
```

Listing 6.4: Verarbeitung der Antwort des Servers.

Die dargestellte Funktion `evaluateResponse` wird bei jeder Statusänderung des *XMLHttpRequest*-Objekts ausgeführt. Durch die Verzweigung in Zeile 5 wird sichergestellt, dass nur vollständig empfangene Antworten verarbeitet werden. Da der *Timeout*, der nach 60 Sekunden nachfragen soll, ob noch länger auf die Antwort des Servers gewartet werden soll, nicht mehr benötigt wird, kann dieser nun gelöscht werden (Z. 6 bis 9). In den folgenden beiden Zeilen (11 und 12) wird zunächst der vom Server empfangene *JSON*-String geparst und im Anschluss ausgeführt.

6.2 Zustände der einzelnen Sessions

Sobald ein Graph zum Laden ausgewählt wurde, ist es notwendig, dass sich der Server Daten über die *Session* merkt. Hierfür wurde die Klasse `StateRepository.State` eingeführt, welche *den Zustand repräsentiert*. Er besteht aus allen für eine *Session* notwendigen Daten, welche in Unterabschnitt 6.2.1 aufgeführt sind. Da der Server von mehreren Nutzern parallel genutzt werden kann, müssen mehrere Zustände verwaltet werden können. Hierzu dient die ArrayList `StateRepository.sessions`. Um Seiteneffekte zu vermeiden, erfolgen Zugriffe auf diese Liste nur innerhalb von `synchronized`-Blöcken.

6.2.1 Inhalt eines Zustands

Für die Verwaltung der Zustände sind folgende Daten notwendig:

sessionId:int Sie ist der Identifier einer *Session*. Da sie gleichzeitig der Index in der ArrayList `StateRepository.sessions` ist, wurde für sie kein eigenes Feld eingeführt. Während der Laufzeit des Servers wird eine `sessionId` nur einmal vergeben.

lastAccess:long In diesem Feld wird der Zeitpunkt gespeichert, zu dem das letzte Mal auf diesen Zustand zugegriffen wurde. Es dient dazu, um feststellen zu können, ob ein *Timeout* für diese Session eingetreten ist (siehe Unterabschnitt 6.2.2).

Der Darstellung des Graphen dienen folgende Felder:

workingCollable:FutureTask<?> Dieses Feld verweist auf ein FutureTask, mit dessen Hilfe der Graph asynchron geladen werden kann. Dies ist notwendig, da sonst die maximale Zeitdauer, die ein Browser auf eine Antwort wartet, überschritten werden könnte.

progress:int Diese Variable speichert den Ladefortschritt.

graph:Graph Dies ist der Zeiger auf den geladenen Graphen.

selectedVertexClasses:HashMap<String,Boolean> Diese HashMap gibt darüber Auskunft, ob ein bestimmter Knotentyp, der über seinen *qualifizierten Namen* identifiziert wird, vom Nutzer deselektiert wurde.

selectedEdgeClasses:HashMap<String,Boolean> Das Äquivalent zu `selectedVertexClasses` für die Kantentypen.

verticesOfTableView:Vertex[] Die Menge aller Knoten, deren Schematyp nicht deselektiert wurde. Sie wird für die tabellarische Darstellung benötigt (siehe Abschnitt 6.4).

edgesOfTableView:Edge[] Das Äquivalent zu `verticesOfTableView` für die Kanten.

navigationHistory:ArrayList<JValue> Diese ArrayList speichert alle Kanten und Knoten, die bei der Navigation besucht wurden. Des Weiteren speichert sie die angezeigten Knotenmengen, falls diese durch *GReQL*-Anfragen des Nutzers oder durch manuelle Angabe von Ids bestimmt wurden.

6.2.2 Löschen eines Zustands

Damit der Speicherverbrauch reduziert wird, werden Zustände von *Sessions*, die *nicht mehr benötigt* werden, entfernt. Um dies feststellen zu können, informiert zum einen der Browser den Server, falls die HTML-Seite des *TGraphBrowsers* verlassen wird. Zum anderen werden Zustände, auf die eine gewisse Zeit nicht zugegriffen wurde, gelöscht.

```
1 <body onload="init();" onunload="closeSession();">
```

Listing 6.5: Der Event-Handler `onunload`.

Wechselt der Nutzer zu einer anderen HTML-Seite oder schließt er den Browser, so kann der Zustand dieser *Session beim Server gelöscht* werden. Damit er über diesen Vorgang informiert wird, wurde, wie man in Listing 6.5 sehen kann, der Event-Handler `onunload` des `body`-Tags gesetzt.

Tritt das Event ein, wird die Methode `closeSession` ausgeführt, die den Server dazu veranlasst, den zu der *Session* gehörenden Zustand, der durch die `SessionId` identifiziert wird, zu löschen.

Da es vorkommen kann, dass die Benachrichtigung über die Beendigung der *Session* den Server nicht erreicht, braucht man eine weitere Sicherung, um nicht mehr benötigte Zustände zu löschen. Aus diesem Grund wurde ein *Timeout für die Zustände* eingefügt. Seine Länge beträgt 10 Minuten. Die Überprüfung, ob ein *Timeout* für einen Zustand eingetreten ist, wird von einem Thread des Typs `DeleteUnusedStates` vorgenommen, der mit dem Server gestartet wurde. Er überprüft anhand des `lastAccess`-Werts alle 60 Sekunden, ob für einen Zustand ein *Timeout* eingetreten ist. Falls dem so ist, wird der Zustand gelöscht[1].

6.3 Laden eines Graphen

Möchte der Nutzer einen auf der lokalen Festplatte des Clients befindlichen oder durch eine URI angegebenen Graphen laden, so prüft der Server zunächst, ob der *Platz im `Workspace` ausreicht*. Sollte es eine Größenbeschränkung geben (siehe Anhang B auf Seite 52), so wird ebenfalls überprüft, ob der Graph eine *zulässige Größe hat*. Sollte er eine der beiden zuvor beschriebenen Bedingungen nicht erfüllen, so wird das Laden verweigert. Ansonsten wird der Graph *im `Workspace` gespeichert*. Sollte bereits ein anderer mit gleichem Namen vorhanden sein, so wird der Nutzer gefragt, ob er diesen überschreiben möchte.

Sobald der Graph im `Workspace` vorliegt, wird auf dem Server ein *neues `State`-Objekt erstellt*, welches den Zustand der nun beginnenden *Session* speichert. Da das Laden länger als die maximale Zeitdauer, die ein Browser auf eine Antwort wartet, dauern kann, muss es asynchron erfolgen. Aus diesem Grund erfolgt es mit Hilfe eines FutureTasks. Es wird durch das Feld `workingCollable` des `State`-Objekts referenziert.

Da der Server den Browser nicht darüber benachrichtigen kann, dass der Graph fertig geladen wurde, muss er eventuell mehrmals nachfragen, ob *das Laden abgeschlossen* wurde. Diese Anfragen werden genutzt, um den Nutzer über den Fortschritt durch einen Ladebalken zu informieren. Um die jeweiligen Prozentwerte zu erhalten, wurde eine `ProgressFunction` implementiert, die den Fortschritt in der int-Variable `progress` speichert.

Sobald der Graph geladen wurde, wird er unter `graph` im `State`-Objekt gespeichert und im Browser erscheint die *Schemadarstellung* sowie die *tabellarische Darstellung* des Graphen.

6.4 Tabellarische Darstellung

Nach dem Laden des Graphen werden alle Knoten und Kanten in den beiden Arrays `verticesOfTableView` und `edgesOfTableView` gespeichert. Sie sind Teil des Zustands einer *Session*. Nachdem beide Arrays einmal erstellt wurden, werden diese nur noch bei der Deselektion eines Schematyps oder der manuellen Festlegung einer Graphelementmenge durch den Nutzer erneut erstellt (siehe Abschnitt 6.6). Dabei werden die Elemente, deren Schematyp nicht deselektiert wurde, per *GReQL*-Anfrage bestimmt.

Um die im Browser angezeigte Tabellenseite erstellen zu können, muss zunächst die Menge der anzuzeigenden Graphelemente bestimmt werden. Wie dies geschieht ist in Unterabschnitt 6.4.1 beschrieben. Im Anschluss wird gezeigt, wie ein Element im HTML-Code repräsentiert wird (Unterabschnitt 6.4.2). Zum Abschluss wird erklärt, wie die Navigation durch den Graphen ermöglicht wird (Unterabschnitt 6.4.3).

[1] Die Länge des *Timeouts* und das Prüfintervall kann beim Start des Servers auf andere Werte gesetzt werden (siehe Anhang B).

6.4.1 Bestimmung der Elemente einer Tabellenseite

Um zu bestimmen, welche Knoten und Kanten auf einer Tabellenseite dargestellt werden sollen, benötigt man zum einen die Id des aktuellen Elements und zum anderen wie viele Elemente auf einer Tabellenseite angezeigt werden sollen. Liegen diese Informationen vor, so wird zunächst die *Position des aktuellen Elements* im Array gesucht. Diese Suche geschieht in vier Schritten:

1. Schaue im jeweiligen Array nach, ob sich das Element an der Position mit dem Index der Id befindet. Falls es dort nicht gefunden wurde, mache mit Schritt 2 weiter.
2. Nutze binäre Suche, um die Position des Elements im Array zu finden. Falls es nicht gefunden wurde, merke die Position, an der das Element hätte stehen sollen und mache mit Schritt 3 weiter.
3. Falls der Typ des aktuellen Elements nicht deselektiert wurde, versuche durch Iteration des Arrays die Position des gesuchten Elements zu finden.
4. Falls das aktuelle Element auch nicht im dritten Schritt gefunden wurde, nimm das Element, das sich an der im zweiten Schritt gemerkten Position befindet, als neues aktuelles Element.

Die binäre Suche im zweiten Schritt wird genutzt, da die meisten Graphelemente nach ihrer Id aufsteigend sortiert sind und dadurch der Aufwand der Suche reduziert werden kann. Da die aufsteigende Sortierung jedoch nicht immer gegeben ist, wird im dritten Schritt das Array nochmals komplett iteriert, jedoch nur, falls sichergestellt ist, dass sich das gesuchte Element überhaupt im Array befindet.

Nachdem der Index des aktuellen Elements bestimmt wurde, kann der Index des ersten Elements der Tabellenseite, auf der sich das aktuelle Element befindet, mit folgender Rechnung bestimmt werden:

$$indexOfFirstElementOnPage = indexOfCurrentElement / elementsPerPage + 1$$

6.4.2 HTML-Darstellung eines Elements

```
1  <tr id="trv1" title="name=null;" style="background-color:rgb(255,192,128);
     ">
2      <td>Wasserfahrzeug<sub>1</sub><a id="v1"/></td>
3      <td id="tdv1" style="text-align:left;">
4            <b style="font-size:large;">&#8594;</b>
5            {
6            <a href="javascript:showElement('e1');">
7                  Transportiert<sub>1</sub>
8            </a>
9            }
10           <a href="javascript:showElement('v3');">
11                 Landfahrzeug<sub>3</sub>
12           </a>
13           <br/>
14           [...]
15     </td>
16 </tr>
```

Listing 6.6: Darstellung eines Elements in der Tabelle.

Von jedem Element, das zur anzuzeigenden Tabellenseite gehört, werden die Id, der Schematyp und die inzidenten Elemente angezeigt. In Listing 6.6 ist zu sehen, wie der zum Knoten $\texttt{Wasserfahrzeug}_1$ gehörende HTML-Code aussieht. Durch das `title`-Attribut in Zeile 1 wird beim Überfahren der Spalte des Knotens `v1` mit dem Cursor dessen Attribute und ihre Werte angezeigt. Sollte "Show attributes" gewählt sein, so wird an seiner statt zwischen der dritten und vierten Zeile ein weiteres `td`-Tag erstellt, in dem die Informationen angezeigt werden. Die Zeichenfolge `→` in Zeile 4 ist die Unicode-Schreibweise für das Zeichen → in HTML-Seiten.

Damit der Nutzer anhand der inzidenten Elemente durch den Graphen navigieren kann, sind die inzidenten Elemente durch Links repräsentiert, wie man an den Zeilen 6 und 10 sehen kann. Mittels des Attributs `href` wird bei einem Klick auf diese die Methode `showElements` aufgerufen. Ihre Funktion ist im folgenden Unterabschnitt erklärt.

6.4.3 Navigation durch den Graph

Wie im vorangegangenen Abschnitt beschrieben, wird beim Klick auf ein inzidentes Element die Methode `showElements` aufgerufen. Sie ist in Listing 6.7 dargestellt. Als Parameter erhält sie die Id des gewünschten Elements. Ihre Funktionsweise wird anhand des exemplarischen Aufrufs `showElement('e1')` erklärt.

```
1  /*
2   * Shows the element with the elementId.
3   *
4   * @param the id of the element which should be shown
5   */
6  function showElement(elementId){
7          [...]
8          if([...]((areVerticesShown() && elementId.charAt(0)=='e')||
9                          (!areVerticesShown() && elementId.charAt(0)=='v'))){
10                 switchTable();
11         }
12         [...]
13         if(!document.getElementById(elementId)){
14                         sendPostRequest("showElementsAsTable",
                               getSelectedNumbersPerPage() + "\n" +
                               isShowAttributesSet() + "\n" + elementId, true);
15         }else{
16                 sendPostRequest("refreshBreadcrumbBar",elementId+"\n"+
                       isTableViewShown+"\n"+true,true);
17                 document.location.href = "#" + elementId;
18                 [...]
19                 changeBackgroundColor(elementId);
20         }
21         [...]
22 }
```

Listing 6.7: Die Methode `showElement`.

Zunächst überprüft die Methode, ob es sich bei dem gesuchten Element um einen Knoten oder eine Kante handelt. Da der aktuelle Parameter mit einem "e" beginnt, handelt es sich um eine Kante. Des Weiteren wird anhand der Formatierung der "Vertices"-Schaltfläche festgestellt, dass zur Zeit die Knoten angezeigt werden. Aus diesem Grund wird die angezeigte Tabelle gewechselt (Zeile 8-11).

Im Anschluss wird überprüft, ob sich die Kante $\texttt{Transportiert}_1$ bereits in der dargestellten Tabelle befindet. Dies wird überprüft, indem in Zeile 13 versucht wird, sich das Element mit der Id

e1 zurückliefern zu lassen. Dabei handelt es sich um einen Anker, der ähnlich dem aus Zeile 2 in Listing 6.6 ist. Falls es dieses Element nicht gibt, wird in Zeile 14 eine Anfrage an den Server geschickt, damit die *entsprechende Tabellenseite erzeugt* wird. Sollte es bereits in der Tabelle vorhanden sein, so wird in Zeile 16 eine Anfrage geschickt, nur die *Brotkrumenleiste zu aktualisieren*. Als nächstes wird die Anzeige auf den Anker e1 zentriert und der Hintergrund der entsprechenden Zeile farblich hervorgehoben.

6.5 Grafische Darstellung

Um die *grafische Darstellung* eines Graphen zu erzeugen, werden in einem ersten Schritt mit einer *GReQL*-Anfrage alle Knoten und Kanten bestimmt, die zur *Umgebung*[2] des aktuellen Elements gehören. Diese werden im Anschluss in eine dot-Datei umgewandelt (Unterabschnitt 6.5.1). Mittels des Graphlayouters *dot* wird daraus eine svg-Datei erzeugt (Unterabschnitt 6.5.2) und schließlich in die HTML-Seite eingebunden (Unterabschnitt 6.5.3).

6.5.1 Erzeugung einer dot-Datei

Nachdem die *Umgebung* bestimmt wurde, können die zu ihr gehörenden Elemente in einer dot-Datei repräsentiert werden. So wird zum Beispiel der Knoten v1 wie folgt repräsentiert:

```
v1 [label="{{v1|Wasserfahrzeug}}" style="filled" fillcolor="#FFC080"
href="javascript:top.showElement('v1');" tooltip="bezeichnung =
\"MSFreedom\"; schraubenZahl = 2; "];
```

Das Attribut `label` gibt dabei die Beschriftung des Knotens an. Sie besteht aus seiner Id sowie seinem Schematyp. Mit dem Attribut `tooltip` wird erreicht, dass beim Überfahren des Knotens mit dem Cursor die Knotenattribute mit ihren Werten angezeigt werden. Sollte die Option "Show attributes" gewählt sein, so werden diese Informationen stattdessen im Attribut `label` angezeigt, das dann wie folgt aussieht: `label="{{v1|Intersection}|bezeichnung = \"MSFreedom\"\lschraubenZahl = 2}"` . Um dem Benutzer die Navigation durch das Anklicken eines Elements zu ermöglichen, wird das Attribut `href` gesetzt. Dieses enthält den Methodenaufruf `top.showElement('v1')`. Das `top` bedeutet, dass die aufgerufene Methode nicht in der svg-Grafik definiert wird, sondern in der HTML-Seite, in der die Grafik eingebunden sein wird. Wie man erkennen kann, wurde das Attribut `fillcolor` auf "#FFC080" gesetzt. Dies bewirkt die Darstellung in rötlicher Farbe, was anzeigen soll, dass es sich bei v1 um das aktuell ausgewählte Element handelt.

Die Repräsentation einer Kante ist ähnlich der eines Knotens. So wird beispielsweise e2 wie folgt repräsentiert:

```
v1 -> v2 [label="e2: Gehoert" tooltip=" " taillabel="2" headlabel="1"
href="javascript:top.showElement('e2');"];
```

Durch `taillabel` und `headlabel` werden die Inzidenznummern der Kante e2 an den Knoten v1 und v2 dargestellt.

6.5.2 Erzeugung einer svg-Datei

Nachdem die *Umgebung* in eine dot-Datei umgewandelt wurde, kann der externe Graphlayouter *dot* genutzt werden, um eine *svg-Grafik zu generieren*. Da es sich bei diesem Programm um ein Kom-

[2]Die Definition des *Umfelds* ist im Unterabschnitt 2.1.1 auf Seite 4 zu finden.

mandozeilentool handelt, muss der Befehl, mit dem es aufgerufen wird, beim Start des Servers als Argument übergeben werden (siehe B).

```
1 creation = Runtime.getRuntime().exec(StateRepository.dot + " -Tsvg -o " +
      svgFileName + " " + dotFileName);
2 creation.waitFor();
```

Listing 6.8: Die Erzeugung der svg-Grafik.

In Listing 6.8 wird gezeigt, wie die Erzeugung der svg-Grafik durch Java gestartet wird. In der ersten Zeile wird das angegebene Kommando mit den Argumenten `-Tsvg -o <svgFile> <dotFile>` versehen und ausgeführt. Mit `-Tsvg` wird angegeben, dass eine svg-Grafik erzeugt werden soll. Mit `-o <svgFile>` wird der Pfad und der Name der zu erzeugenden svg-Datei angegeben. `<dotFile>` ist die dot-File, deren Erstellung im vorangegangenen Unterabschnitt erklärt wurde. Mit dem Befehl `waitFor` in Zeile 2 wird erreicht, dass gewartet wird, bis *dot* fertig ist.

6.5.3 Einbinden der svg-Grafik in die HTML-Seite

Nachdem die svg-Datei erstellt wurde, kann der JavaScript-Code erzeugt werden, mit dem sie in die *HTML-Seite eingebunden* wird. Da die Browser Mozilla Firefox und Safari svg-Grafiken von sich aus unterstützen, kann sie mittels eines `object`-Tags, welches in Listing 6.9 dargestellt ist, eingebunden werden.

```
1 <object data="0Graphennippet.svg" type="image/svg+xml"/>
```

Listing 6.9: Einbindung einer svg-Grafik mittels `object`-Tag.

Da der Internet Explorer bis einschließlich Version 8 keine svg-Grafiken unterstützt, muss das Plugin "Adobe SVG Viewer" installiert werden. Da es das `object`-Tag nur fehlerhaft unterstützt, wird geraten das `embed`-Tag, welches in Listing 6.10 dargestellt ist, an dessen Stelle zu verwenden[3].

```
1 <embed src="0Graphennippet.svg" type="image/svg+xml"/>
```

Listing 6.10: Einbindung einer svg-Grafik mittels `embed`-Tag.

6.5.4 Navigation durch den Graphen

Um durch den Graphen zu navigieren, genügt es auf ein Graphelement zu klicken. Dadurch wird die ausgewählte Kante bzw. der ausgewählte Knoten zum aktuellen Element und wird der Brotkrumenleiste hinzugefügt. Um die Anzeige zu aktualisieren, muss nun die svg-Grafik erneut erzeugt werden. Wie dies geschieht wurde in diesem Abschnitt bereits erklärt.

6.6 Auswahl von Elementen

Um die dargestellte Menge an Knoten und Kanten einzuschränken, hat der Nutzer mehrere Möglichkeiten. Eine besteht darin die dargestellten Elemente nach ihren Schematypen zu filtern. Wie dies realisiert ist, wird in Unterabschnitt 6.6.1 beschrieben. Eine weitere Möglichkeit besteht in

[3] siehe `http://www.adobe.com/svg/viewer/install/`

der direkten Eingabe der Ids von den Elementen, die man sich betrachten möchte. Ihre Realisierung wird in Unterabschnitt 6.6.2 erklärt. Die letzte Alternative besteht darin, *GReQL*-Anfragen zu stellen. Ihre Implementation wird in Unterabschnitt 6.6.3 beschrieben.

6.6.1 Auswahl der Schematypen

Nach dem Laden eines Graphen wird einmal die *Schemadarstellung* des Browsers erzeugt. Dabei werden *alle Schematypen als selektiert* markiert. Dies bedeutet für den Browser, dass alle Checkboxen selektiert sind. Für den Server bedeutet es, dass in den beiden HashMaps `selectedVertexClasses` und `selectedEdgeClasses` die jeweiligen Schematypen auf den Wert "true" gemapped werden. Als Schlüssel dient der *qualifizierte Name* des Typs.

Um zu verstehen, wie die Deselektion und eventuell spätere Selektion eines Schematyps funktioniert, wird zunächst seine HTML-Repräsentation im Unterunterabschnitt 6.6.1.1 erklärt. Im Anschluss (Unterunterabschnitt 6.6.1.2) wird beschrieben, was im Browser passiert, falls ein Typ aus- oder angewählt wird. Zum Schluss wird in Unterunterabschnitt 6.6.1.3 erklärt, was beim Server geschieht, sobald er über die (De)Selektion informiert wird.

6.6.1.1 Darstellung mittels HTML

Die Darstellung der einzelnen Schematypen unterscheidet sich je nachdem, ob sie Subtypen haben oder nicht. So ist beispielsweise der Typ `Person` aus Abb. 4.6, wie in Listing 6.11 zu sehen, aufgebaut.

```
1 <li id="liPerson">
2       <input type="checkbox" id="inputPerson" name="inputPerson" value="
          Person" onclick="function(){deSelect('Person',this.id);
          submitDeselectedTypes();}"/>
3       <p title="Person">
4             Person
5       </p>
6 </li>
```

Listing 6.11: Darstellung eines Typs ohne Subtypen.

Schematypen, die Subtypen haben, sind, wie in Listing 6.12 anhand des Beispiels `Fahrzeug` gezeigt, aufgebaut.

```
1 <li id="liFahrzeug">
2       <a id="aFahrzeug" href="javascript:expand('ulFahrzeug','aFahrzeug');
          ">
3             <img src="plus.png" alt="+"/>
4       </a>
5       <input type="checkbox" id="inputFahrzeug" name="inputFahrzeug" value
          ="Fahrzeug" onclick="function(){deSelect('Fahrzeug',this.id);
          submitDeselectedTypes();}"/>
6       <p title="Fahrzeug">
7             <i>Fahrzeug</i>
8       </p>
9       <ul id="ulFahrzeug" style="disply:none">
10            [...]
11      </ul>
12 </li>
```

Listing 6.12: Darstellung eines Typs mit Subtypen.

Das `img`-Tag zeigt das Bild `plus.png`. Mit seiner Hilfe kann sich der Nutzer die *Subtypen anzeigen lassen*, die sich zwischen dem öffnenden und dem schließenden `ul`-Tag befinden. Ob es angezeigt wird oder nicht, wird durch die *CSS-Eigenschaft `display` gesteuert*. Der Wert "none" bedeutet, dass es versteckt ist. Um es anzuzeigen, muss er nur auf "block" gesetzt werden. Dies geschieht durch die Methode `expand`, die beim Klick auf das Bild ausgeführt wird. Des Weiteren verändert sie das `src`-Attribut des `img`-Tags, wodurch `minus.png` im Browser angezeigt wird.

Wie man erkennen kann, fehlt das `img`-Tag in Listing 6.11. Dadurch würden die Checkboxen der Schematypen einer Ebene nicht mehr korrekt untereinander stehen. Daher wird die Checkbox mittels *CSS*, um die Breite des fehlenden Bildes nach rechts verschoben, dargestellt.

Die kursive Schreibweise der abstrakten Schematypen wird durch die `i`-Tags in Zeile 7 erreicht. Da beim Überfahren mit dem Cursor der *qualifizierte Name* eines Schematyps angezeigt werden soll, wurde in Zeile 6 das Attribute `title` gesetzt.

Die Id eines HTML-Elements setzt sich aus dem *Tagnamen und dem UniqueName* des Schematyps zusammen, da jede Id einzigartig sein muss. Im Falle der Mehrfachvererbung muss der HTML-Repräsentant unter jedem Supertyp aufgeführt werden. Da jedes HTML-Element höchstens ein Elternelement haben darf, muss für jeden Obertyp ein neuer Repräsentant erstellt werden. Um die Einzigartigkeit der Ids zu gewährleisten, wird sie um *":" und einer fortlaufenden Nummer* erweitert.

6.6.1.2 (De)Selektion eines Schematyps

Wenn man einen Typ (de)selektiert, so sollen auch alle Subtypen sowie alle Mehrfachvorkommen dieser Typen (de)selektiert werden. Um dies zu gewährleisten, wurden die drei Methoden `deSelect`, `deSelectMulti` und `deSelectSubtypes` erstellt. Ihre Funktionsweise wird anhand eines Beispiels erklärt: In der Schemadarstellung, die in Abb. 4.6 zu sehen ist, wird der Typ `Landfahrzeug` selektiert.

```
1  /*
2   * Sets all checkboxes wich are in an li wich starts with li+baseName2
3   * to the state of caller.checked.
4   *
5   * @param baseName2 the uniqueName of the represented Class
6   * @param the caller of this method
7   */
8  function deSelect(baseName2, caller){
9         var checked = document.getElementById(caller).checked;
10        var baseName;
11        if(baseName2.lastIndexOf(":")>=0){
12               baseName = baseName2.substring(0,baseName2.lastIndexOf(":"));
13        }else{
14               baseName = baseName2;
15        }
16        // deSelect the orignial element and its children
17        document.getElementById('input'+baseName).checked = checked;
18        deSelectSubtypes('input' + baseName,'ul' + baseName);
19        // deSelect copies of this element
20        deSelectMulti(baseName, checked);
21 }
```

Listing 6.13: Die Methode `deSelect`.

Durch den Klick auf die vorangestellte Checkbox wird zunächst die Methode `deSelect` aufgerufen (Listing 6.13). Als aktuelle Parameter werden der *UniqueName* `Landfahrzeug` sowie die Id der geklickten Checkbox "inputDuesenflugzeug" übergeben.

Da es vorkommen kann, dass der (de)selektierte Schematyp mehrmals in der Baumstruktur vorkommt, muss sichergestellt werden, dass alle Vorkommen (de)selektiert werden. Aus diesem Grund muss zunächst das Vorkommen, das als *erstes in der Baumstruktur* erstellt wurde, auf den gleichen Zustand wie die angeklickte Checkbox gesetzt werden. Es kann daran erkannt werden, dass sein *UniqueName* mit keinem Suffix versehen wurde. Daher wird in den Zeilen 11 bis 15 ein eventuell existierendes Suffix entfernt. Im Anschluss werden mit dem Aufruf der Methode `deSelectSubtypes` in Zeile 18 alle direkten und indirekten Subtypen des soeben beschriebenen Vorkommens selektiert. Schließlich würden mit `deSelectMulti` auch alle Mehrfachvorkommen des Typs `Landfahrzeug` ausgewählt, falls es diese gäbe.

```
1  /*
2   * Sets all checkboxes below ul to the state of changedCheckbox.
3   *
4   * @param changedCheckbox the changed checkbox
5   * @param ul the sublist
6   */
7  function deSelectSubtypes(changedCheckbox, ul){
8        if(document.getElementById("checkSelectAll").checked){
9              var parentUl = document.getElementById(ul);
10             var checkbox = document.getElementById(changedCheckbox);
11             if(parentUl!=null){
12                   var subCheckboxes = parentUl.getElementsByTagName("input
                        ");
13                   if(subCheckboxes!=null){
14                         for(var i=0;i<subCheckboxes.length;i++){
15                               subCheckboxes[i].checked = checkbox.checked;
16                               deSelect(subCheckboxes[i].value,
                                    subCheckboxes[i].id);
17                         }
18                   }
19             }
20       }
21 }
```

Listing 6.14: Die Methode `deSelectSubTypes`.

Die Methode `deSelectSubtypes` (Listing 6.14) *(de)selektiert alle Subtypen*, falls die Option "De/-Select subtypes" gewählt ist. Da dies in Abb. 4.6 der Fall ist, soll auch die Checkbox vor `Amphibienfahrzeug` selektiert werden. Um dies zu erreichen, wird über alle `input`-Elemente, die sich im *DOM*-Baum unterhalb des `ul`-Elements vom `Landfahrzeug`-Repräsentanten befinden, iteriert (Z. 12-18). Durch Zeile 15 wird jedes dieser Elemente selektiert. Da es mehrere `Amphibienfahrzeug`-Vorkommen gibt, müssen diese auch noch selektiert werden. Dazu wird in Zeile 16 die Methode `deSelect` aufgerufen.

```
1  function deSelectMulti(baseName, setChecked){
2        var counter = 0;
3        while(document.getElementById('li'+baseName+':'+counter)!=null){
4              document.getElementById('input'+baseName+':'+counter).checked
                  = setChecked;
5              deSelectSubtypes('input'+baseName+':'+counter,'ul'+baseName+':
                  '+counter);
6              counter++;
7        }
8  }
```

Listing 6.15: Die Methode `deSelectMulti`.

Alle *Mehrfachvorkommen eines Typs* werden von der in Listing 6.15 dargestellten Funktion `deSelectMulti` mittels Iteration gefunden. In jedem Iterationsschritt (Z. 3-7) wird überprüft, ob es ein input-Element mit der um ":0", ":1", etc. erweiterten Id gibt. Falls dem so ist, wird dieses mit samt seinen Subtypen selektiert (Z. 4, 5).

(De)Selektiert man ein Paket, kann das eben beschriebene Verfahren nicht genutzt werden, da sich innerhalb von ihm unterschiedlich benannte Schematypen befinden können. Aus diesem Grund wurde vom Server eine Funktion erstellt, in der die (De)Selektion für jeden Schematyp des gewählten Pakets explizit angegeben wird.

```
1  /*
2   * Selects or deselects all Classes which match the regular expression
3   * of inputRegEx
4   *
5   * @param select if true the matches are selected.
6   * Otherwise they are deselected.
7   */
8  function matchRegEx(select){
9        var regString = document.getElementById('inputRegEx').value;
10       var div=null;
11       if(document.getElementById("aVertex").getAttribute(div.hasAttribute?
             "class":"className")=="geklickt"){
12             div = document.getElementById("ulRootVertex");
13       }else if(document.getElementById("aEdge").getAttribute(div.
             hasAttribute?"class":"className")=="geklickt"){
14             div = document.getElementById("ulRootEdge");
15       }else{
16             div = document.getElementById("ulRootPackage");
17       }
18       // Select all checkboxes
19       var classes = div.getElementsByTagName("input");
20       for(var i=0;i<classes.length;i++){
21             var value=classes[i].value;
22             if(new RegExp(regString,"gi").test(value)){
23                   classes[i].checked = select;
24             }
25       }
26       submitDeselectedTypes();
27 }
```

Listing 6.16: (De)Selektion mittels *regulärer Ausdrücke*.

Eine andere Möglichkeit der (De)Selektion geschieht *mittels regulärer Ausdrücke*. Dies geschieht mit der in Listing 6.16 *dargestellten Methode* `matchRegEx`. Falls alle Schematypen, die mit dem *regulären Ausdruck* matchen, selektiert werden sollen, hat der Parameter `select` den Wert "true" sonst "false".

Zunächst wird bestimmt, ob die Knoten-, die Kantentypen oder die Pakete angezeigt werden (Z. 11 bis 17). In den Zeilen 19 bis 25 wird überprüft, ob ein *qualifizierter Name*, der im `value`-Attribut vermerkt ist, mit dem *regulären Ausdruck* matched. Falls dem so ist, so wird die entsprechende Checkbox (de)selektiert. Beim Matchen wird die Groß- und Kleinschreibung ignoriert.

Sobald Typen selektiert oder deselektiert wurden, muss der Server informiert werden. Es genügt, wenn er mitgeteilt bekommt, welche deselektiert wurden. Um dies festzustellen, wird zunächst

über die Checkboxen der Knotentypen in der Schemadarstellung iteriert. Die *qualifizierten Namen* der deselektierten Typen werden durch "#" separiert. Im Anschluss geschieht das Gleiche noch einmal für die Kantentypen. Die beiden auf diesem Wege erstellten Strings werden schließlich *an den Server gesendet*.

6.6.1.3 Verarbeitung beim Server

Zum Verarbeiten der Anfrage, die der Server empfängt, nachdem ein Schematyp (de)selektiert wurde, wird die Methode `refreshViewAfterTypeSubmit` aufgerufen. Als aktuellen Parameter wird ihr je ein String mit den *qualifizierten Namen* der deselektierten Knoten- und Kantentypen übergeben. Die einzelnen Namen werden dabei durch "#" voneinander getrennt. Zunächst werden die Strings in die einzelnen *qualifizierten Namen* gesplittet und im Anschluss die *Einträge für diese Namen* in den HashMaps `selectedVertexClasses` bzw. `selectedEdgeClasses` *auf "false" gesetzt*. Zum Abschluss wird noch der Code erstellt, mit dessen Hilfe die Graphdarstellung des Browsers aktualisiert wird.

6.6.2 Auswahl durch die Ids der dargestellten Elemente

Nachdem ein Nutzer die Ids der gewünschten Graphelemente z.B. "v2#e1" eingegeben hat, werden alle nicht-alphanumerische Zeichen in Kommas umgeformt. Im Anschluss werden alle Kommas am Anfang und am Ende entfernt. Sollten zwischen zwei Ids mehrere aufeinander folgende Kommas stehen, so werden sie durch ein einziges ersetzt. Der entstandene Text "v2,e1" wird dem Server als ein Parameter der Methode `showTypedElements` übergeben.

Innerhalb dieser Methode wird der String zunächst nach den Kommas gesplittet und die entsprechenden Knoten und Kanten des Graphen bestimmt. Damit sie in der Brotkrumenleiste auftauchen, werden sie in der ArrayList `navigationHistory` des `State`-Objekts abgelegt. Sollte es sich um mehr als ein Element handeln, so werden sie, sofern ihre Schematypen nicht deselektiert wurden, in die im Browser gewählte Darstellungsform umgewandelt. Handelt es sich allerdings nur um ein Element, so wird der JavaScript-Code erzeugt, mit dessen Hilfe die Anzeige auf dieses Element zentriert wird.

6.6.3 Auswahl mit einer GReQL-Anfrage

Sobald der Nutzer auf "Submit" klickt, so wird die eingegebene *GReQL-Anfrage an* den *Server geschickt*. Da bei den POST-Anfragen jeder aktuelle Parameter in einer eigenen Zeile steht, werden vor der Übertragung die Zeilenumbrüche `\n` und `\r` durch Leerzeichen ersetzt.

Sollte bei der Auswertung der Anfrage beim Server ein Fehler auftreten, so wird die Fehlermeldung an den Browser gesendet. Ansonsten wird das Ergebnis der *GReQL*-Anfrage daraufhin überprüft, dass sie nur aus Knoten und Kanten besteht, da der *TGraphBrowser* nur diese anzeigen kann. Sollte eine Anfrage diese Bedingung nicht erfüllen, so wird eine Nachricht erzeugt, die den Nutzer darüber informiert. Handelt es sich allerdings nur um Knoten oder Kanten, so wird diese Graphelementmenge der Liste `navigationHistory` hinzugefügt, damit sie in der Brotkrumenleiste auftaucht. Zum Abschluss werden alle Elemente, deren *Schematyp nicht deselektiert* wurde, im Browser dargestellt.

6.7 Brotkrumenleiste

Immer wenn der Nutzer bei der Navigation durch den Graphen auf eine Kante oder einen Knoten klickt, wird ein Eintrag in der Brotkrumenleiste angelegt. Darüber hinaus wird das Element auch der ArrayList `navigationHistory` auf dem Server hinzugefügt. Das Gleiche passiert, wenn eine Menge von Elementen durch die Eingabe ihrer Id oder durch eine *GReQL*-Anfrage bestimmt wird. Dadurch wird erreicht, dass die Brotkrumenleiste als *grafische Repräsentation der `navigationHistory`* angesehen werden kann.

```
1 <a href="javascript:goBackToElement(6,'v6');" style="color:white;">v6</a>
```

Listing 6.17: Die HTML-Darstellung eines Eintrags der Brotkrumenleiste.

Klickt der Nutzer auf einen Eintrag in der Brotkrumenleiste, so wird, wie man in Listing 6.17 sehen kann, die Methode `goBackToElement` aufgerufen. Die “6” ist dabei der Index in der Liste `navigationHistory` des zu dieser *Session* gehörenden `State`-Objekts, an der sich das gewählte Element befindet. Der zweite Parameter “v6” ist die Id des gewählten Elements, mit der kontrolliert wird, ob es bereits in der Graphdarstellung angezeigt wird. Falls dem so ist, wird die Anzeige auf dieses zentriert und alle Brotkrumeneinträge, die neuer als der gewählte sind, werden, durch das Verändern des *CSS*-Attributs auf “gray”, in grauer Schrift dargestellt . Ansonsten wird zuvor noch die Graphdarstellung erneuert, so dass das gewünschte Element angezeigt wird.

Nachdem nun beschrieben wurde, wie der *TGraphBrowser* implementiert wurde, wird im folgenden Kapitel anhand einer Beispielsession gezeigt, wie das erstellte System genutzt werden kann, um einige Problemstellungen zu lösen.

Kapitel 7

Beispielablauf einer Session

In diesem Kapitel soll der *Einsatz des TGraphBrowsers demonstriert* werden. Hierzu soll ein Beispielablauf einer *Session* beschrieben werden. Der zugrundeliegende Graph heißt “OsmGraph.tg”. Er befindet sich auf der CD, die dieser Bachelorarbeit beiliegt (siehe Anhang A). Da er 900.870 Knoten und 2.400.481 Kanten besitzt, reicht die normale *Heapsize* der JVM nicht aus. Aus diesem Grund muss sie beim Start des Servers auf beispielsweise 1024 MB vergrößert werden. Dies geschieht mittels des VM-Arguments `-Xmx1024M`.

Der OsmGraph stammt aus dem Projekt *JGStreetMap*. In diesem Projekt wurde ein gleichnamiges Navigationssystem entwickelt, dass auf den *TGraphen* des Java-Graphenlabors basiert. Die Informationen, die durch diesen Graph repräsentiert werden, wurden aus den im XML-Format vorliegenden Daten des Open Street Map Projekts[1] gewonnen [Zie08].

Zunächst wird in Abschnitt 7.1 das Schema des Graphen erklärt. Im Anschluss wird der Beispielablauf durch verschiedene Problemstellungen demonstriert (Abschnitt 7.2).

7.1 Schema des Beispielgraphen

Das Schema des Beispielgraphen besteht aus zwei Teilen. Der eine stellt die *Bestandteile eines zweidimensionalen kd-Baums* dar. Der andere repräsentiert die *Informationen einer Straßenkarte*.

Ein *zweidimensionaler kd-Baum* ist ein binärer Suchbaum, mit dessen Hilfe eine Menge von Punkten einer Ebene verwaltet werden kann. Jeder Knoten besitzt einen Wert. Dieser ist bei einer geraden *Entfernung*[2] zur Wurzel die Y-Koordinate. Ansonsten wird die X-Koordinate genommen. Jeder Punkt, der sich unter bzw. links von diesem Wert befindet, ist im linken Teilbaum. Die anderen sind im rechten. An den Blättern des *kd-Baums* befinden sich die jeweiligen Punkte. Eine genauere Beschreibung ist unter [kdT10] zu finden.

In Abb. 7.1 wird der *kd-Baum* durch die Knotenklasse `KDTree` dargestellt. Dieser speichert die Tiefe des Baums und hat eine inzidente Kante `HasRoot` auf das Wurzelelement. Die Klasse `Key` repräsentiert die Baumknoten, die die Werte, nach denen die Punkte aufgeteilt werden, speichern. Dabei gibt es spezielle Knoten für jede der beiden Koordinaten (`XKey` und `YKey`). Die Kind-Eltern-Relation wird durch die beiden Kanten `HasXChild` und `HasYChild` dargestellt. Über `HasSet` kann jedem Baumknoten eine Menge von Punkten zugeordnet werden. Sie ist durch die Knotenklasse `NodeSet` realisiert, deren Elemente über `HasElement` erreichbar sind. Die Punkte der Landkarte heißen `Node`.

[1] `http://www.openstreetmap.org/`

[2] Die *Entfernung* zwischen zwei Knoten wurde in Unterabschnitt 2.1.1 definiert.

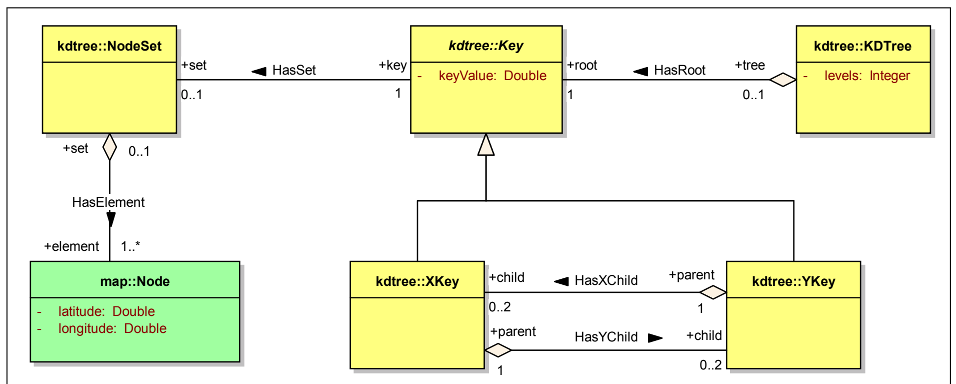


Abbildung 7.1: Die zum *kd-Baum* gehörenden Schematypen. Quelle: Repository des JGStreetMap, Autor: Elisa Ziegler

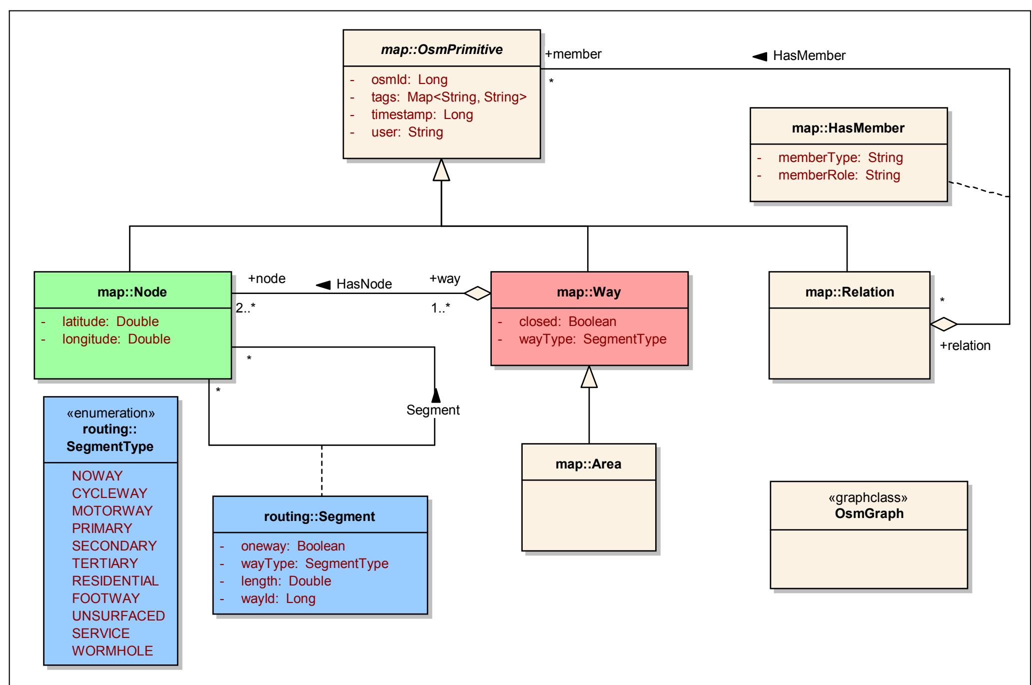


Abbildung 7.2: Der Teil des Schemas, der die Straßenkarteninformationen repräsentiert. Quelle: [Zie08]

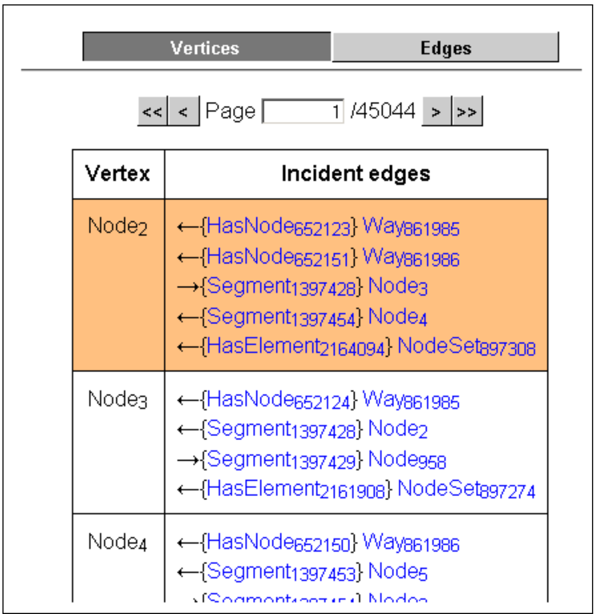


Abbildung 7.3: *Tabellarische Darstellung* nach dem Laden des Graphen.

Die Karteninformationen werden durch die Schematypen aus Abb. 7.2 repräsentiert. In dieser Modellierung besteht eine Straßenkartekarte aus Punkten (repräsentiert durch die Klasse `Node`), Wegen (`Way`) und Relationen (`Relation`). Letztere wäre beispielsweise ein Stadtteil mit allen zu ihm gehörenden Objekten, wie beispielsweise Straßen. Die Zuordnung geschieht über die Kanten des Typs `HasMember`. Eine `Node`-Instanz steht für einen durch Längen- und Breitengrad bestimmten Punkt auf der Straßenkarte. So werden beispielsweise Kreuzungen durch einen Knoten dieses Typs dargestellt. Einem Weg sind alle Punkte der Karte zugeordnet, die sich auf diesem befinden. Hierzu dienen die `HasNode`-Kanten. Durch das Attribut `tags`, kann jedes Element der Karte mit näheren Informationen versehen werden [Zie08].

7.2 Beispielablauf

Nachdem man mit dem Browser Kontakt zum Server aufgebaut hat, erscheint die Startseite, auf der man den zu ladenden Graphen angeben kann. Hier wählt man die Datei "OsmGraph.tg" von der CD, die dieser Bachelorarbeit beiliegt. Daraufhin erscheint ein Fortschrittsbalken, der anzeigt, zu wie viel Prozent der Graph geladen wurde. Sobald 100% erreicht wurde, erscheint die *tabellarische Darstellung* des Graphen.

Auffinden des Knotens *v1*
Wie man in Abb. 7.3 sehen kann, fehlt der Knoten `v1`. Um diesen zu finden, blättert man zunächst über die Schaltfläche ">" eine Seite weiter. Da sich `v1` auch nicht auf dieser befindet, wiederholt man diesen Vorgang noch einige weitere Male. Da man nicht auch noch durch die restlichen der 45.044 existierenden Tabellenseiten blättern möchte, nutzt man das Eingabefeld, welches sich links neber der Schaltfläche "Select by GReQL" befindet. Hier gibt man "v1" ein. Drückt man nun die Enter-Taste, so springt man direkt zum gewünschten Knoten.

Nur den *kd-Baum* anzeigen
Betrachtet man den Knoten `v1`, so stellt man fest, dass er vom Typ `KDTree` ist (siehe Abb. 7.4). Um

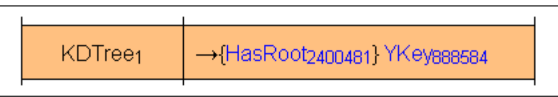


Abbildung 7.4: Der Knoten `v1`.

sich nur noch die Elemente des *kd-Baums* anzeigen zu lassen, wechselt man in der Schemadarstellung durch einen Klick auf "Packages" zu der Paket-Ansicht. In dieser deselektiert man `map` und `routing`. Da man nur an der Baumstruktur interessiert ist, deselektiert man noch die Kantentypen `HasSet` und `HasElement`. Des Weiteren wird noch der Knotentyp `NodeSet` abgewählt. Nun sieht man in der *tabellarischen Darstellung* nur noch die Elemente des *kd-Baums*.

Durch den *kd-Baum* navigieren

Um durch den *kd-Baum* zu navigieren, klickt man als nächstes auf die inzidente Kante $\texttt{HasRoot}_{2400481}$. Daraufhin erscheint sie in der *tabellarischen Darstellung*. Durch einen Klick auf $\texttt{YKey}_{888584}$ gelangt man zum Wurzelknoten. Nun wechselt man in die *grafische Darstellung*. Über die Knoten $\texttt{XKey}_{894728}$ bis $\texttt{XKey}_{894738}$ erreicht man schließlich das Blatt $\texttt{YKey}_{894739}$. Nun wechselt man zurück in die *tabellarische Darstellung*.

Zum Wurzelknoten zurückkehren

Um zum Wurzelknoten zurückzukehren, klickt man in der Brotkrumenleiste zunächst auf die drei Punkte auf der linken Seite und im Anschluss auf `v888584`.

Elemente der Straßenkarte anzeigen

Möchte man sich die Elemente der Straßenkarte anzeigen lassen, so deselektiert man in der *Schemadarstellung* das Paket `kdtree` und selektiert `map`. Da zur Zeit der $\texttt{YKey}_{888584}$ das aktuelle Element ist, wird die letzte Tabellenseite angezeigt, welche beim Knoten $\texttt{Relation}_{888583}$ endet. Das aktuelle Element wird nicht angezeigt, da sein Schematyp deselektiert wurde. Die letzte Tabellenseite ist erschienen, da sich $\texttt{YKey}_{888584}$ auf dieser befunden hätte. Um zur ersten Seite zu gelangen, klickt man entweder auf die Schaltfläche "<<" oder man gibt in das Textfeld der Navigationsleiste "1" ein. Da man auch an den Attributen interessiert ist, aktiviert man die Option "Show Attributes".

Alle Nodes finden, die sich südlich des 50. Breitengrads befinden

Um nun alle `Node`-Instanzen zu finden, die südlich des 50. Breitengrads liegen, nutzt man eine *GReQL*-Anfrage. Klickt man auf die Schaltfläche "Select by GReQL", erscheint eine Box. In diese gibt man folgende Anfrage ein:

```
FROM v:V{map.Node}

WITH v.latitude<50

REPORT v

END
```

Nach einem Klick auf "Submit" werden alle `Node`-Instanzen, die diese Bedingung erfüllen, angezeigt. Um die *Session* zu beenden, genügt es, das Browserfenster zu schließen.

Nachdem nun dieses Beispiel erläutert wurde, werden im nächsten Kapitel einige Möglichkeiten der Erweiterung vorgestellt und eine Zusammenfassung gegeben.

Kapitel 8

Ausblick und Zusammenfassung

In diesem Kapitel wird zunächst ein Ausblick gegeben, wie der *TGraphBrowser* verbessert werden könnte (Abschnitt 8.1). Im Anschluss wird in Abschnitt 8.2 betrachtet, ob die Ziele dieser Bachelorarbeit erreicht sind.

8.1 Ausblick

Durch die Nutzung des Programms sowie durch Gespräche mit den Betreuern dieser Bachelorarbeit sind weitere Ideen entstanden, wie man den *TGraphBrowser* noch benutzerfreundlicher gestalten könnte. Diese Ideen werden in den folgenden Unterabschnitten näher beleuchtet.

8.1.1 Sichtbarkeit der Graphen

Da die hochgeladenen Graphen alle in einem gemeinsamen `Workspace` vorliegen, kann jeder Benutzer auf alle Graphen zugreifen und diese auch wieder löschen. Es kommt jedoch vor, dass ein Nutzer einen Graphen betrachten will, der *keinem anderen zur Verfügung stehen* soll. Um zu verhindern, dass ein Graph von anderen betrachtet werden kann, bestände die Möglichkeit den Nutzer wählen zu lassen, ob er den Graphen in den `Workspace` hochladen möchte, oder ob er direkt vom InputStream geladen werden soll. Als Alternative bestände auch die Möglichkeit, eine Benutzerverwaltung einzurichten, so dass ein Graph entweder für alle oder nur für den jeweiligen Nutzer sichtbar sein soll. Dies könnte beim Hochladen festgelegt werden.

8.1.2 Anpassung der zweidimensionalen Darstellung

Eine weitere Verbesserung bestände darin, dem Nutzer mehr Möglichkeiten zu geben, die *zweidimensionale Darstellung* an seine Bedürfnisse anzupassen. Denkbar wäre beispielsweise den Nutzer wählen zu lassen, ob er die *Inzidenznummern angezeigt* haben will, da sie bei manchen Graphen, deren Knoten mehr als acht inzidente Kanten haben, kaum noch einer bestimmten Kante zuzuordnen sind bzw. nicht mehr zu lesen sind, da sie sich gegenseitig überdecken.

Des Weiteren könnte der Nutzer *zwischen verschiedenen Layoutverfahren wählen*. So bietet *Graphviz* neben *dot* noch weitere Layoutverfahren, wie beispielsweise *twopi*. Dieses platziert alle Knoten in konzentrischen Kreisen um einen zentralen Knoten.

8.1.3 Farbwähler

Da die Farbgebung der HTML-Seiten den Geschmack einiger Nutzer nicht traf, bestünde die Möglichkeit einen Farbwähler zu implementieren, damit jeder Nutzer sich seine *eigenen Farben aussuchen* kann. Da durch die Wahl der Hintergrundfarbe die Schrift unter Umständen nur noch schwer oder gar nicht mehr zu lesen wäre, könnte sie ebenfalls durch den Nutzer gewählt werden oder sie würde automatisch bestimmt. Denkbar wäre z.B. dass es ein Farblayout der Schrift für helle und für dunkle Hintergründe gäbe. Um welche dieser beiden Hintergrundarten es sich handelt, könnte über die Hexadezimalwerte der Farbe bestimmt werden.

8.2 Zusammenfassung

Das Ziel dieser Bachelorarbeit war es ein Programm zu entwickeln, mit dessen Hilfe man sich *Graphen des Java-Graphenlabors (JGraLab) betrachten* kann. Es gab zwar schon im alten JGraLab ein Tool mit dessen Hilfe man einen Graphen in eine statische HTML-Seite umwandeln konnte. Die generierten Seiten hatten jedoch den Nachteil, dass sie bei Graphen mit mehreren tausend Elementen so groß waren, dass Browser beim Versuch die Seite anzuzeigen streikten. Um diesen Nachteil zu beheben wurde ein *Webserver* entwickelt, mit dessen Hilfe man im Browser immer nur einen Teil der Knoten- und Kantenmenge betrachtet.

Durch ihn können dem Nutzer weitere Features zur Verfügung gestellt werden, die man mit einer statischen HTML-Seite nicht hat:

- Zur Betrachtung des Graphen kann der Nutzer *zwischen zwei Darstellungsformen umschalten*: eine tabellarische und eine zweidimensionale. Um eine intuitive Navigation durch den Graphen zu ermöglichen, genügt es in jeder der beiden Darstellungsformen auf ein inzidentes Element zu klicken.
- Da für den Nutzer gegebenenfalls nur ein Teil des Graphen von Interesse ist, wurde ihm die Möglichkeit gegeben die Menge der *dargestellten Graphelemente zu bestimmen*. Zum einen kann er entweder durch explizite Angabe der Ids oder durch eine GReQL-Anfrage die für ihn interessante Knoten- und Kantenmenge festlegen und zum anderen kann er die Graphelemente nach ihren Schematypen filtern.

Mittels des in dieser Bachelorarbeit erstellten *TGraphBrowsers* steht nun ein Werkzeug zur Verfügung, mit dem man selbst Graphen mit mehreren tausend Elementen auf einer übersichtlichen Art betrachten kann.

Anhang A

Inhalt der CD

Auf der beiliegenden CD befinden sich die folgenden Ordner:

- *source*: In diesem Verzeichnis befindet sich der komplette Quellcode des *TGraphBrowsers* als Eclipse-Export.
- *jar*: Dieses Verzeichnis enthält den *TGraphBrowser* als eine ausführbare jar-Datei, sowie die für die Ausführung benötigten html- und png-Dateien.
- *sample*: Dieses Verzeichnis beinhaltet vier Beispielgraphen, die betrachtet werden können. Unter anderem befindet sich dort auch der Graph "OsmGraph.tg", welcher zur Erklärung einer Beispielsession in Kapitel 7 genutzt wird.
- *thesis*: Dieser Ordner enthält die Lyx-Dokumente dieser Bachelorarbeit. Lyx ist ein graphisches Textverarbeitungssystem, mit dessen Hilfe man Latex-Dokumente erstellen kann. Es ist unter der URL `http://www.lyx.org/` zu finden.

Anhang B

Starten des Servers

Um sich einen Graphen mit dem *TGraphBrowser* ansehen zu können, muss zunächst der Server gestartet werden. Falls nicht anders angegeben, nutzt er dabei den Port 80. Da der Server einen `Workspace` benötigt, muss dieser durch das Argument *-w* bzw. *--workspace* gesetzt werden. Darüber hinaus gibt es noch folgende optionale Argumente:

-p, --port Mittels dieses Arguments kann man den Port des Servers festlegen. Wird dieses Argument nicht gesetzt, so wird Port 80 genutzt.

-t, --state_timeout Mit diesem Argument wird der Timeout für die gespeicherten Zustände der einzelnen Sessions gesetzt. Dies bedeutet, dass ein Zustand gelöscht werden kann, falls innerhalb der angegebenen Sekundenzahl keine Anfrage von dem entsprechenden Client eingetroffen ist. Falls dieses Argument nicht gesetzt wird, beträgt der Timeout 600 Sekunden. Die Bedeutung dieses Arguments wird im Unterabschnitt 6.2.2 auf Seite 33 genauer erklärt.

-ci, --check_interval Dieses Argument legt fest, nach wie vielen Sekunden alle Stati des Servers überprüft werden sollen, ob ihr Timeout erreicht ist. Der Default-Wert beträgt 60 Sekunden. Die Bedeutung dieses Arguments wird im Unterabschnitt 6.2.2 auf Seite 33 genauer erklärt.

-s, --size_of_workspace Wird durch dieses Argument eine Megabytezahl angegeben, so wird das Hochladen eines Graphen verweigert, falls im `Workspace` nicht mehr genügend freigegebener Speicher vorhanden ist.

-m, --maximum_filesize Mit diesem Argument legt man fest, wie viele Megabyte eine hochzuladende Datei maximal umfassen darf.

-i, --incidences Durch dieses Argument kann man die Anzahl der angezeigten inzidenten Elemente bestimmen. Dies ist notwendig, da ein Knoten beliebig viele ein- und ausgehende Kanten haben kann. Würde man alle Inzidenzen gleichzeitig anzeigen, könnte dies den Browser überfordern. Der Default-Wert ist 10. Die Bedeutung dieses Arguments wird im Abschnitt 6.4 auf Seite 34 genauer erklärt.

-d, --dot Mit diesem Argument wird das Kommando übergeben, durch das *dot* aufgerufen wird. Wird dieser Parameter nicht gesetzt, so ist die zweidimensionale Darstellung des Graphen nicht verfügbar.

Abbildungsverzeichnis

2.1 Ein Graph mit dem dazugehörigen Schema 3
2.2 Die unterschiedlichen Kommunikationsarten einer Web-Anwendung. Quelle: http://commons.wikimedia.org/w/thumb.php?f=Ajax-vergleich.svg&width=2000px 7

4.1 Die HTML-Seite, bei der man den zu ladenden Graphen auswählen kann. 16
4.2 Die Gesamtübersicht. 17
4.3 Die *tabellarische Darstellung*. 17
4.4 Die Navigationsleiste der inzidenten Kanten. 18
4.5 Die *grafische Darstellung*. 19
4.6 Die *Schemadarstellung*. 20
4.7 Die Komponenten zur Angabe einer Graphelementmenge. 21
4.8 Box zur Eingabe einer GReQL-Anfrage. 22
4.9 Die Brotkrumenleiste. 22

5.1 Systemarchitektur 25

7.1 Die zum *kd-Baum* gehörenden Schematypen. Quelle: Repository des JGStreetMap, Autor: Elisa Ziegler 46
7.2 Der Teil des Schemas, der die Straßenkarteninformationen repräsentiert. Quelle: [Zie08] 46
7.3 *Tabellarische Darstellung* nach dem Laden des Graphen. 47
7.4 Der Knoten `v1`. 48

Listings

2.1 Anwendungsbeispiel des *XMLHttpRequest*. 9
2.2 Die *DOT*-Repräsentation des Graphen aus Abb. 2.1(a). 10
5.1 Eine POST-Anfrage. 27
5.2 Antwort des Servers. 27
6.1 Senden einer GET-Anfrage mit *AJAX*. 29
6.2 Senden einer POST-Anfrage mit *AJAX* 30
6.3 Verarbeitung der empfangenen Daten beim Server. 31
6.4 Verarbeitung der Antwort des Servers. 32
6.5 Der Event-Handler `onunload`. 33
6.6 Darstellung eines Elements in der Tabelle. 35
6.7 Die Methode `showElement`. 36
6.8 Die Erzeugung der svg-Grafik. 38
6.9 Einbindung einer svg-Grafik mittels `object`-Tag. 38
6.10 Einbindung einer svg-Grafik mittels `embed`-Tag. 38
6.11 Darstellung eines Typs ohne Subtypen. 39
6.12 Darstellung eines Typs mit Subtypen. 39
6.13 Die Methode `deSelect`. 40
6.14 Die Methode `deSelectSubTypes`. 41
6.15 Die Methode `deSelectMulti`. 41
6.16 (De)Selektion mittels *regulärer Ausdrücke*. 42
6.17 Die HTML-Darstellung eines Eintrags der Brotkrumenleiste. 44

Literaturverzeichnis

[BB09] BERGMANN, Olaf ; BORMANN, Carsten: *AJAX - Frische Ansätze für das Web-Design*. http://www.teialehrbuch.de/Kostenlose-Kurse/AJAX/. Version: August 2009

[CSM09] *Client-Server-Modell*. http://de.wikipedia.org/wiki/Client-Server-Modell. Version: November 2009

[Ebe09] EBERT, Prof. Dr. J.: *Software Reengineering*. http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST/AGEbert/teaching/ws200910/sre. Version: Dezember 2009

[Gar05] GARRETT, Jesse J.: *AJAX: A New Approach to Web Applications*. http://www.adaptivepath.com/ideas/essays/archives/000385.php. Version: February 2005

[GKN06] GANSNER, Emden ; KOUTSOFIOS, Eleftherios ; NORTH, Stephen: *Drawing graphs with* dot. http://www.graphviz.org/Documentation/dotguide.pdf. Version: Januar 2006

[GKNV93] GANSNER, Emden R. ; KOUTSOFIOS, Eleftherios ; NORTH, Stephen C. ; VO, Kiem-Phong: *A Technique for Drawing Directed Graphs*. IEEE Trans. Software Eng., 1993 (19(3)). – 214–230 S.

[JSO09] *Einführung in JSON*. http://json.org/json-de.html. Version: August 2009

[Kah06] KAHLE, Steffen: *JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen*, Universität Koblenz-Landau, Campus Koblenz, Diplomarbeit, Juni 2006

[kdT10] *k-d-Baum*. http://www-lehre.informatik.uni-osnabrueck.de/~dbs/2001/skript/node34.html. Version: Januar 2010

[Mar06] MARCHEWKA, Katrin: *Entwurf und Definition der Graphanfragesprache GReQL 2*, Universität Koblenz-Landau, Campus Koblenz, Diplomarbeit, September 2006

[Ses10] *Sitzung (Informatik)*. http://de.wikipedia.org/wiki/Sitzung_%28Informatik%29. Version: Januar 2010

[W3C09] *XMLHttpRequest*. http://www.w3.org/TR/2009/WD-XMLHttpRequest-20090820/. Version: August 2009

[Web10] *Webanwendung*. http://de.wikipedia.org/wiki/Webanwendung. Version: Januar 2010

[Wen07] WENZ, Christian: *JavaScript & AJAX - Das umfassende Handbuch*. Galileo Press, 2007 (7. Auflage). – 391–420 S.

[XML10] *Understanding XML Server*. http://www.xml-training-guide.com/xml-server.html. Version: Januar 2010

[Zie08] ZIEGLER, Elisa: *JGStreetMap - Ein Navigationssystem als Anwendungsbeispiel von Graphentechnologie*. https://svn.uni-koblenz.de/gup/re-group/trunk/project/jgstreetmap/documentation/jgstreetmap_anwendungsbsp_graphen.pdf. Version: Oktober 2008

Index

2D-Visualizer, 25

Abstand, 5
AJAX, 8
AJAX-Modell, 8

Brotkrumenleiste, 22, 44

Client-Server-Prinzip, 6

DOT, 10
dot, 9

Entfernung, 45

grafische Darstellung, 16, 18, 24, 37
Graphviz, 9, 25
GReQL, 6, 22, 43

JSON, 27

kd-Baum, 45
klassisches Modell, 6

Pfad, 19
Pfadlänge, 19

regulärer Ausdruck, 21, 42

Schema-Visualizer, 25
Schemadarstellung, 16, 20, 39
Session, 26, 32
State-Repository, 25

tabellarische Darstellung, 16, 24, 34
Tabular-Visualizer, 25
TGraph, 1, 4
Timeout, 34

Umgebung, 5, 19, 37
UniqueName, 20

Webanwendung, 6
Workspace, 24

XMLHttpRequest, 8

Zustand, 32