Universität Leipzig

Wirtschaftswissenschaftliche Fakultät

Institut für Wirtschaftsinformatik

Prof. Dr. Ulrich Eisenecker

Dipl.-Wirtsch.-Inf. Pascal Kovacs

Thema

Interaktionskonzept zur Erstellung und Speicherung von dreidimensionalen Teilvisualisierungen

Bachelorarbeit zur Erlangung des akademischen Grades
Bachelor of Science – Wirtschaftsinformatik

vorgelegt von: Marcel Winter

Matrikelnummer: 2869967

Email-Adresse: win.marcel@web.de

Telefonnummer: 0152 279 66 375

Anschrift: Gärtnerstraße 181 / 442

04209 Leipzig

Leipzig, den 24.November 2015

Abstract

Softwarevisualisierung unterstützt verschiedene an einem Softwareentwicklungsprojekt beteiligte Stakeholder, indem es die sonst nur abstrakte und immaterielle Software in eine grafische, vom Menschen leichter erfassbare Form bringt. Am Institut für Wirtschaftsinformatik der Universität Leipzig wurde ein Konzept zur automatisierten Erstellung von Softwarevisualisierungen erarbeitet. Darauf aufbauend folgte die Entwicklung eines Generators, welcher auf Grundlage von Quellcode dreidimensional darstellbare Daten generiert. Um diese Daten zu visualisieren und zu explorieren, wurde zudem eine plattformunabhängige Weboberfläche entwickelt. Durch diese kann der Nutzer die Visualisierung unter anderem beliebig filtern, indem er einzelne Fragmente ausblendet.

Diese Filterungen nehmen derzeit noch viel Zeit in Anspruch, da die Fragmente zuerst in der Visualisierung und anschließend in einer Liste lokalisiert werden müssen. Einmal erstelle Teilsichten können nicht gespeichert werden und müssen bei jedem Aufruf der Weboberfläche erneut reproduziert werden. Das Ziel dieser Arbeit ist es daher, Methoden zur schnelleren Erstellung und Speicherung von Teilsichten zu entwickeln. *Point and Click* und *Drag and Drop* werden als Interaktionsmethoden zur Erstellung der Teilsichten vorgestellt und anhand bestehender, wissenschaftlicher Studien verglichen. *Drag and Drop* wird als die günstigere Methode umgesetzt. Die Implementierung ermöglicht sowohl das Ziehen von Einträgen der integrierten Listenansicht als auch der Fragmente der Visualisierung. Das Konzept zum Speichern und Laden erlaubt es, die IDs der ausgeblendeten Visualisierungsfragmente in eine CSV-Datei zu speichern. Beim Ladevorgang wird die zum Zeitpunkt der Speicherung dargestellte Visualisierung wiederhergestellt. Dadurch entfällt die manuelle Reproduktion der Teilsicht.

Schlüsselwörter

Softwarevisualisierung, 3D, X3Dom, Interaktion, Drag and Drop, Speicherung

Gliederung

Abbildungsverzeichnis

Quellcode-Listings

Abkürzungsverzeichnis

2D	zweidimensional
3D	dreidimensional
API	Application Programming Interface
BPMN	Business Process Model and Notation
CSS	Cascading Style Sheets
CSV	Comma Seperated Values
HTML	Hypertext Markup Language
ID	Identifikator
JS	JavaScript
LOC	Lines of Code
MIME	Multipurpose Internet Mail Extensions
MIT	Massachusetts Institute of Technology
OMG	Object Management Group
UML	Unified Modeling Language
W3C	World Wide Web Consortium
X3D	Extensible 3D
XML	Extensible Markup Language

1 Einleitung

1.1 Motivation

Softwarevisualisierung wird zur Unterstützung des Verständnisses von Software verwendet, indem diese durch meist einfache, geometrische Formen dargestellt wird. Sie soll dadurch ihren abstrakten und immateriellen Charakter verlieren und für Menschen leichter begreifbar gemacht werden. Dies erweist sich bei Softwareentwicklungsprojekten als nützlich, da bei diesen oftmals Menschen mit unterschiedlichem Wissensstand in Bezug auf Software zusammenarbeiten. Die Softwarevisualisierung schafft eine gemeinsame Kommunikationsbasis, indem relevantes für alle Beteiligten verständlich dargestellt wird.

Die vorliegende Arbeit baut auf den jüngsten Fortschritten auf, welche das Institut der Wirtschaftsinformatik an der Universität Leipzig in den letzten Jahren im Bereich der Softwarevisualisierung erreicht hat. Unter anderem wurde ein Konzept erarbeitet, welches beschreibt, wie der Prozess zur Erstellung von dreidimensionalen Softwarevisualisierungen automatisiert werden kann. Dieses Konzept bildete die Grundlage zur Entwicklung eines Generators, welcher Quellcode in dreidimensional visualisierbare Daten überführt. Weiterhin wurde eine plattformunabhängige Browseroberfläche entwickelt, um diese Daten darzustellen und Operationen zur Filterung, Markierung und Untersuchung darauf anzuwenden. Detailliertere Informationen zu den einzelnen Softwarefragmenten werden erst auf Abruf präsentiert, sodass Betrachter ohne entsprechendes technisches Vorwissen nicht überfordert sind, Betrachter mit dem nötigen Know-How aber dennoch darauf zugreifen können. Sämtliche Fragmente sind in eine in die Oberfläche integrierte Listenansicht einsehbar, welche es dem Anwender ermöglicht, diese beliebig Ein- und Auszublenden. Dadurch wird der Fokus auf bestimmte Teile des gesamten Softwareprojekts gerichtet und kann effektiv präsentiert werden.

1.2 Zielstellung

Da das Erstellen einer solchen Teilsicht der Gesamtvisualisierung zu diesem Zeitpunkt noch sehr viel Zeit kostet und einige redundante Schritte erfordert, soll es das Ziel dieser Arbeit sein, diesen Prozess zu vereinfachen und zu beschleunigen. Die Auswahl an Möglichkeiten, mit der Visualisierung zu interagieren, soll durch ein Konzept zum Transfer von Daten von einem Fenster in ein anderes erweitert werden. In diesem Zielfenster, in welches die Daten übertragen werden, muss die Oberfläche bereits geöffnet sein. Um den Ablauf der Implementierung zu strukturieren, werden drei Ziele definiert:

- Ziel 1.1: Auswahl einer geeigneten Interaktionsmethode anhand wissenschaftlicher Kriterien. Als Kandidaten wurden die Point-and-Click- sowie die Drag-and-Drop-Methode betrachtet.
- Ziel 1.2: Interaktion mit der integrierten Listenansicht ermöglichen, um damit Teilvisualisierungen zu erzeugen.
- Ziel 1.3: Interaktion mit der dargestellten Softwarevisualisierung ermöglichen, um damit Teilvisualisierungen zu erzeugen.

Weiterhin soll es möglich sein, erstellte Teilvisualisierungen abzuspeichern und zu einem späteren Zeitpunkt erneut zu laden. Wird die Browseroberfläche geladen, so wird zunächst stets die vollständige Visualisierung dargestellt. Die Teile der Visualisierung, welche nicht zur Teilsicht gehören, müssen anschließend einzeln über den Package Explorer gesucht und deselektiert werden. Um diesen Prozess zu umgehen, werden folgende Ziele definiert:

- Ziel 2.1: Es soll möglich sein, einmal erstellte Teilsichten zu speichern. Die Speicherdatei soll lokal auf dem Rechner abgelegt werden können.
- Ziel 2.2: Implementierung einer Möglichkeit zum Laden der Datei und der darauf folgenden Wiederherstellung der gespeicherten Teilsicht.

1.3 Aufbau der Arbeit

Im Anschluss an das einleitende Kapitel wird im zweiten Kapitel allgemeines Wissen zum Thema Softwarevisualisierung vermittelt. Dem Leser soll dadurch ein Überblick über die in dieser Arbeit behandelte Problematik vermittelt sowie die in diesem Zusammenhang verwendeten Begriffe verständlich gemacht werden. Zudem ist es das Ziel, den Stand der Forschung im Hinblick auf die zu erweiternde Software darzustellen. Das dritte Kapitel erläutert die zur Erreichung der Ziele genutzten Techniken und Vorgehensweisen. Darunter zu finden sind sowohl verwendete Sprachen wie HTML und JavaScript als auch Frameworks und Methodiken.

Im vierten Kapitel wird der Weg zur Erreichung der Ziele 1.1, 1.2 und 1.3 erläutert. Entsprechend folgt zuerst ein Vergleich der Interaktionsmethoden *Point and Click* und *Drag and Drop* unter wissenschaftlichen Aspekten. Die geeignetere Methode wird anschließend verwendet, um die Oberfläche um diese Interaktionsmöglichkeit zu erweitern. Das fünfte Kapitel thematisiert die Implementierung von Funktionalitäten zum Laden und Speichern von Teilsichten und umfasst somit die Maßnahmen, die getroffen werden, um die Ziele 2.1 und 2.2 zu erreichen. Das sechste Kapitel fasst die Ergebnisse dieser Arbeit zusammen und zeigt Anknüpfpunkte für weitere Arbeiten auf diesem Gebiet auf.

2 Stand der Forschung

Dieses Kapitel vermittelt grundlegendes Wissen zu den in dieser Arbeit behandelten Themen. Zu Beginn wird der Begriff der Visualisierung allgemein definiert sowie der Nutzen ihrer Anwendung aufgezeigt. Anschließend wird Bezug zur Softwarevisualisierung hergestellt. Dabei werden verschiedene existierende Ansätze präsentiert. Darauf folgt die Beschreibung der Arbeitsweise des Generators zur Erstellung visualisierbarer Daten aus Quellcode sowie deren Darstellung in der dafür konzipierten Weboberfläche.

2.1 Grundlagen der Softwarevisualisierung

2.1.1 Einordnung

Die Softwarevisualisierung ist eine spezielle Art der Visualisierung, welche durch Bereitstellen von Werkzeugen und Methoden zur Erstellung von rollen- und aufgabenspezifischen Sichten den Entwicklungs- und Wartungsprozess von Softwaresystemen unterstützt (vgl. [Müller et al. 2011]). Bevor jedoch auf die Softwarevisualisierung im Speziellen eingegangen wird, soll zunächst die Visualisierung im Allgemeinen betrachtet werden. Die Definition von Card et al. lautet:

„The use of computer-supported, interactive, visual representations of data to amplify cognition.“ [Card et al. 1999, 6]

Demnach fassen Visualisierungen die meist abstrakt oder verteilt vorliegenden Daten in einer visuellen Form zusammen. Diese erlaubt dem Betrachter eine schnellere Auffassung der Daten durch Exploration und Interaktion. Dabei darf nicht vergessen werden, dass es unterschiedliche Gruppen von Betrachtern mit jeweils unterschiedlichen Fähigkeitenprofilen gibt. Am Beispiel der Softwarevisualisierung wären hier unter anderem die Entwickler und die Kunden eines Softwaresystems zu nennen. Beide Zielgruppen betrachten die Software unterschiedlich. So interessieren den Entwickler eher technische, detaillierte, den Kunden eher fachliche, gesamtheitliche Daten. Das Kriterium, welches misst, wie gut sich die Visualisierung zur Vermittlung der Informationen und damit zur Förderung des Verständnisses eignet, wird Effektivität (engl. Effectiveness) genannt. Ein weiteres Kriterium ist die Expressivität oder Ausdrucksmächtigkeit (engl. Expressiveness). Diese gibt an, ob die Informationen deutlich und unverfälscht wiedergegeben werden. Beide Kriterien wurden von Mackinlay definiert, um visuelle Abbildungen von Daten bewerten zu können (vgl. [Mackinlay 1986]).

Card et al. ordnen Visualisierungen anhand der Art der zugrundeliegenden Daten in zwei Teilgebiete ein: Zum einen die wissenschaftliche und zum anderen die Informationsvisualisierung (vgl. [Card et al. 1999, 6]). Die wissenschaftliche Visualisierung fußt auf Messergebnissen oder Daten, welche aus Simulationen gewonnen werden. Ein Beispiel wären die Daten, welche sich aus der Messung des Luftdrucks an einem gleichbleibenden Ort über einen bestimmten Zeitraum ergeben. Diese Daten basieren auf Prozessen der physischen Welt und werden daher physische Daten genannt. Demgegenüber stehen abstrakte Daten, die in der Informationsvisualisierung betrachtet werden. Diese sind in der physischen Welt nicht sicht- oder messbar, beispielsweise Börsenkurse oder demographische Kennzahlen.

Software ist ein immaterielles Produkt und kann weder angefasst noch gesehen werden. Zudem ist sie, anders als technische Produkte, schwer zu vermessen (vgl. [Balzert 2009, 9f]). Aufgrund dieses abstrakten Charakters wird die Softwarevisualisierung in die Informationsvisualisierung eingeordnet (vgl. [Diehl 2007, 3]).

2.1.2 Definition

Laut Müller existieren einige Definitionen der Softwarevisualisierung, deren Gemeinsamkeit der Bezug zu *Struktur und Verhalten* von Softwaresystemen ist (vgl. [Müller 2009, 7]). Woran es diesen Definitionen mangelt, ist das Einbeziehen der Eigenschaft von Software, sich weiterzuentwickeln. Nur die Struktur und das Verhalten einer Software zu betrachten, kommt einer Momentaufnahme in ihrer Lebensspanne gleich. Aus diesem Grund nennt Müller die Definition von Reiss, welche besagt, Softwarevisualisierung sei "*the development and evaluation of methods for graphically representing different aspects of software, including its structure, its abstract and concrete execution, and its evolution*" [Reiss 2005]. Müller hebt die Bedeutung der *Evolution* in der Softwareentwicklung hervor, welche Reiss in seiner Definition zusammen mit dem bereits genannten Aspekten aufzählt. Weiterhin bezieht Müller die folgende Definition von Diehl mit ein, welcher die eben zitierte Definition adaptiert und zusammenfasst:

"*[...] the visualization of artifacts related to software and its development process. [...] visualizing the structure, behavior, and evolution of software.*" [Diehl 2007, 3]

Müller folgt beiden Definitionen im Rahmen seiner Arbeit (vgl. [Müller 2009, 7]). So dient die Softwarevisualisierung dem Verständnis von Software durch die Darstellung relevanter Teile der Softwaresysteme und Artefakte, wobei Artefakte Zwischen- und Endergebnisse des Prozesses der Softwareentwicklung darstellen. Diese werden den Kategorien Struktur, Verhalten und Evolution zugeordnet. (vgl. [Diehl 2007])

2.1.3 Aufgaben

Bohnet et al. zeigen drei Bereiche auf, in denen die Softwarevisualisierung von Nutzen ist: Zuerst wird sie beim *Entwurf und der Entwicklung* von Software eingesetzt, indem anhand von Diagrammen statische und dynamische Informationen dargestellt werden. Verbreitet sind hier sowohl die Unified Modeling Language (UML) zur Darstellung von Klassen und deren Attributen und Methoden sowie die Beziehungen untereinander als auch die Business Process Model and Notation (BPMN), welche zur Abbildung von Geschäftsprozessen verwendet wird. Beide Notationen werden von der Object Management Group (OMG), einem internationalen Konsortium zur Entwicklung technologischer Standards, verwaltet.
Neben der Visualisierung technischer Daten eines Softwareentwicklungsprozesses ist es auch möglich, die *Überwachung und Verwaltung* dieses Prozesses zu visualisieren. Die Evolution des Softwaresystems kann anhand von historisierten Daten dargestellt werden, um dem Management einen Eindruck zu vermitteln, an welchen Stellen das System wie schnell wächst. Zuletzt wird Softwarevisualisierung in der Phase der *Wartung, Pflege und Erweiterung* herangezogen (vgl. [Bohnet et al. 2006, 4]). Dadurch, dass sowohl statische, dynamische als auch historisierte Daten vorliegen, können Optimierungs- und Erweiterungsaufgaben erleichtert werden, da redundante oder fehlerhafte Teile des Softwaresystems schneller lokalisiert werden können (vgl. [Müller 2009, 8]).

2.1.4 Metaphern

Eine Metapher hat den Zweck, das Verständnis einer Sache durch die Betrachtung einer anderen zu erreichen (vgl. [Lakoff und Johnson 1980, 5]). Eine visuelle Metapher ist die grafische Repräsentation einer abstrakten Einheit mit dem Ziel, dem Betrachter die Eigenschaften dieser Einheit in einer greifbaren Art zu vermitteln. Diehl nennt in diesem Zusammenhang die Bezeichnung der grafischen Nutzeroberfläche verbreiteter Betriebssysteme als Schreibtisch (engl. Desktop). (vgl. [Diehl 2007, 31]) Ein anderes Beispiel sind hierarchische, gerichtete, unzyklische Graphen. Diese werden als Bäume (engl. Trees) bezeichnet, deren Ursprungsknoten als Wurzel (engl. Root), die Kanten zwischen den Knoten als Zweige (engl. Branches) und die Knoten ohne ausgehende Kante als Blätter (engl leafs).

Gračanin et al. teilen visuelle Metaphern in zwei Gruppen auf: Abstrakte und natürliche Metaphern. Während abstrakte Metaphern die Artefakte lediglich anhand von einfachen geometrischen Figuren wie Kugeln, Kreise, Vierecke und Linien darstellen, wird bei natürlichen Metaphern versucht, diese Figuren in einen aus der realen Welt bekannten Kontext zu versetzen (vgl. [Gračanin 2005]). Im Folgenden wird in Abbildung 2-1 eine dreidimensionale, hierarchische Visualisierung nach Balzer und Deussen gezeigt, welche

keine aus der realen Welt bekannte Form besitzt (vgl. [Balzer und Deussen 2004]). Im Vergleich dazu bildet die Metapher in Abbildung 2-2 die Teile der Software in der Form von Sonnensystemen ab. Dabei werden die Java-Packages von Sonnen repräsentiert, um die die Objekte und Klassen als Planeten fliegen, deren Größe sich nach den *Lines of Code* (LOC) richtet.

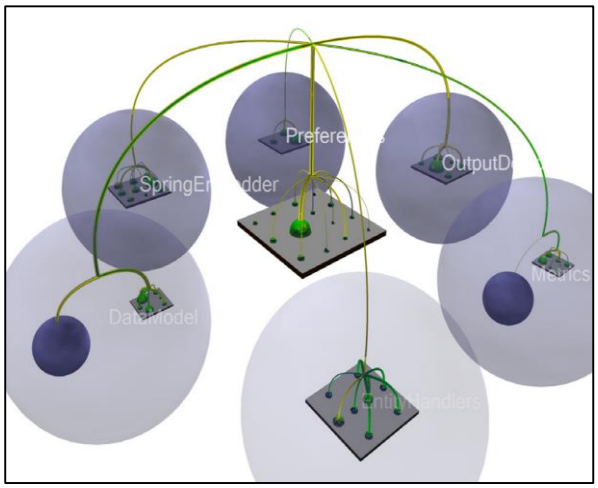

Abbildung 2-1: 3D-Visualisierung einer Softwarehierarchie

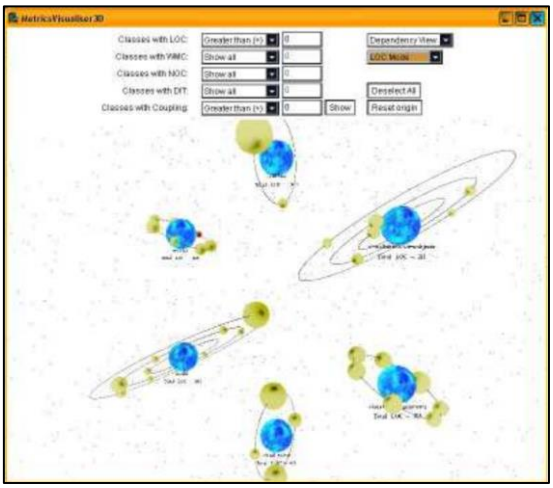

Abbildung 2-2: Sonnensystemmetapher

Neben der Unterteilung in zwei Untergruppen definieren Gračanin et al. ebenfalls Kriterien zur Wahl der geeigneten Metapher. Eine davon ist die semantische Vielfalt und Komplexität (engl. Semantic richness and complexity), die voraussetzt, dass die Metapher genug unterschiedliche Darstellungsformen für alle zu visualisierenden Aspekte der Software besitzt. Es sollen genug eindeutige Äquivalente bereitstehen, ohne aber den Betrachter zu überfordern oder ihm das Gefühl zu geben, sich in der Darstellung zu verlieren. Daher wird an dieser Stelle formuliert, dass "*[t]he scope of the representation determines to a certain extent the nature of the metaphor to be chosen.*" [Gračanin et al. 2005]

Das andere Kriterium ist die Konsistenz (engl. Consistency). Die Abbildung der Softwareartefakte soll über die gesamte Visualisierung hinweg konsistent bleiben. Mehrere Artefakte dürfen nicht durch eine Metapher dargestellt werden. Ebenso dürfen nicht mehrere Metaphern verwendet werden, um ein Artefakt darzustellen. Im Beispiel der Abbildung 2-2 dürften dementsprechend keine Sonnen auftauchen, die Klassen symbolisieren. Ebenso wäre es nicht erlaubt, andere Himmelskörper einzuführen, die ebenfalls Packages repräsentieren (vgl. [Gračanin et al. 2005]).

2.1.5 Generierung einer Softwarevisualisierung

Im Rahmen seiner Arbeit beschreibt Müller die Voraussetzungen zur Entwicklung eines Generators, welcher aus gegebenen Artefakten einer Software auf weitgehend automatisiertem Wege visualisierbare Daten erstellt. Seine Forschungsergebnisse setzt er in

Form eines Prototypen um, welcher als Plugin für die Entwicklungsumgebung Eclipse konzipiert wurde (vgl. [Müller 2009]). Dadurch mindert er das Problem der „*vom Softwareentwicklungsprozess entkoppelten Werkzeuge zur Visualisierung*“ [Müller 2009]. Die Schritte, welche durch den Generator ausgeführt werden, sind in der Visualisierungspipeline von in Abbildung 2-3 zu sehen (vgl. [Müller und Zeckzer 2015]). Diese Pipeline basiert auf dem Haber-McNabb-Datenflussmodell (vgl. [Haber und McNabb 1990]), welches um den Arbeitsschritt der Analyse ergänzt wurde (vgl. [dos Santos und Brodlie 2004]) und beschreibt den Weg der unverarbeiteten Rohdaten bis hin zur dreidimensionalen Darstellung.

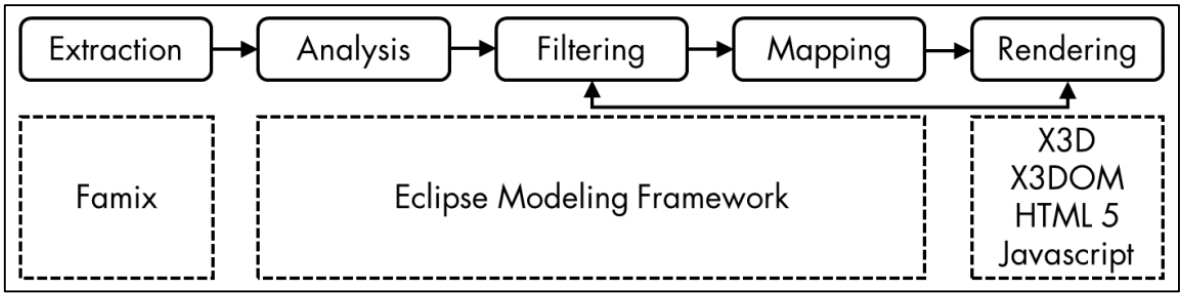

Abbildung 2-3: Visualisierungspipeline nach Müller und Zeckzer

Im ersten Schritt werden die für die Visualisierung benötigten Daten aus der Software *extrahiert* (engl. extraction). Während der *Analyse* (engl. Analysis) werden die Daten anhand eines Metamodells auf konforme Syntax und Semantik überprüft. Ebenso kann der Anwender eigene Regeln vordefinieren (vgl. [Müller und Zeckzer 2015]). Auf die Analyse folgt die *Filterung* (engl. Filtering), bei dem relevante Datenmengen selektiert werden. Nur diese werden dem nächsten Prozessschritt übergeben und am Ende visualisiert. Dies hat den Vorteil, dass der Anwender bereits im Vorfeld die ihm wichtigen Daten selektieren kann. Der nächste Schritt ist das *Mapping*, der elementare Teil dieses Prozesses. An dieser Stelle wird bestimmt, wie die gefilterten Daten auf die geometrischen Formen abgebildet werden sollen. Dies geschieht in mehreren Schritten: Zuerst werden die Daten als plattformunabhängiges Modell abgebildet, um die Größen und Positionen innerhalb der Visualisierung bestimmen zu können. Diese Daten werden anschließend auf ein plattformspezifisches Modell, in diesem Fall Extensible 3D (X3D), abgebildet. Um die Plattformunabhängigkeit wiederherzustellen, wird das X3D-Modell in das X3DOM-Format konvertiert. Dieses kann nun im letzten Schritt, dem *Rendering*, im Browser dargestellt werden. In dieser Phase kann eine weitere Form der Filterung angewendet werden, die unter anderem das Suchen sowie des Ein- und Ausblenden einzelner oder mehrerer Fragmente beinhaltet.

2.2 Darstellung der Softwarevisualisierung und Interaktion

2.2.1 Die Oberfläche

Abbildung 2-4 zeigt eine Darstellung der Oberfläche im Browser, in welcher eine dreidimensionale Softwarevisualisierung zu sehen ist. Die Darstellung durch einen Browser unterstützt das plattformunabhängige Konzept des Projekts, da bereits die durch den Generator erstellten X3D-Dateien plattformunabhängig eingesetzt werden können.

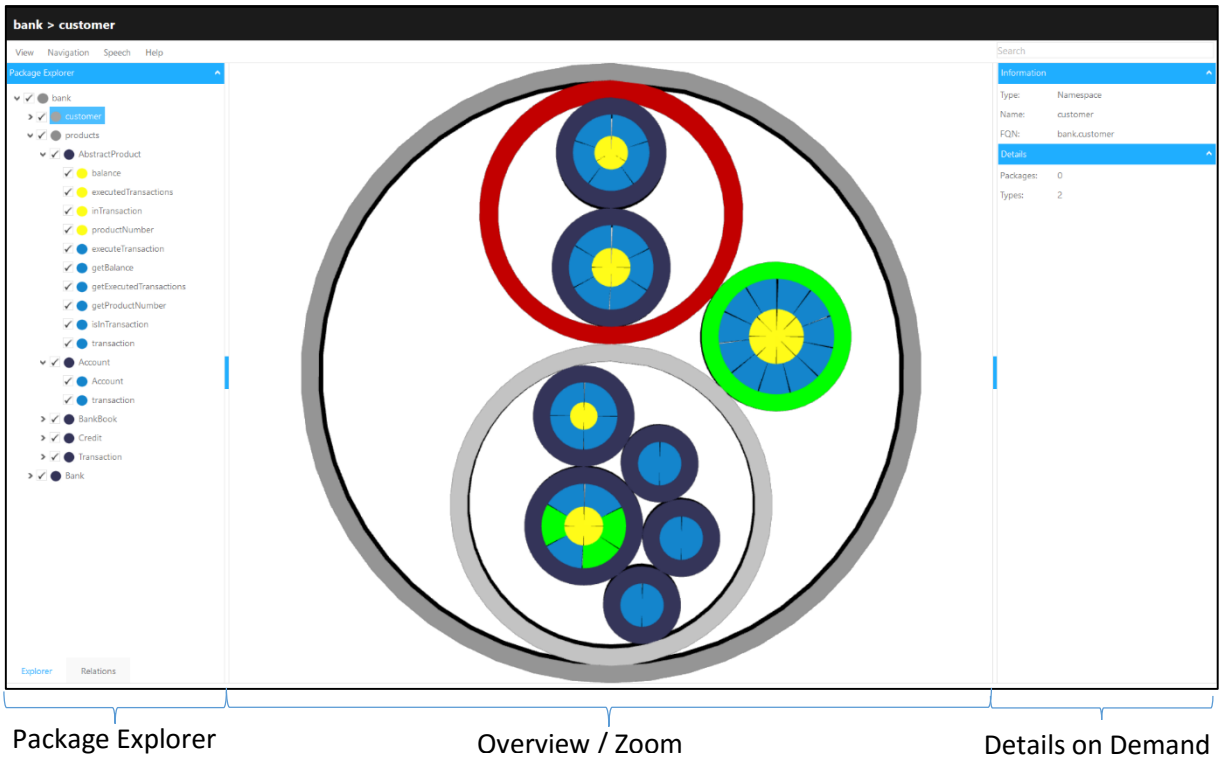

Abbildung 2-4: Oberfläche der Softwarevisualisierung

Wie zu sehen ist, gliedert sich die Oberfläche in drei Bereiche. Der *Overviewbereich* befindet sich in der Mitte. In diesem Bereich kann die Visualisierung unter Zuhilfenahme der Maus untersucht werden. Die einzelnen Fragmente, aus denen die Visualisierung aufgebaut ist, werden Glyphen genannt. Anhand ihrer geometrischen Beschaffenheit können die topologischen Beziehungen der Datensätze untereinander sowie deren Eigenschaften abgelesen werden (vgl. [Borgo et al. 2013]). Die standardmäßigen Interaktionen umfassen Drehung, Verschiebung und Zoom. Es ist möglich, beliebig viele Glyphen durch Anklicken mit der rechten Maustaste zu markieren. Diese werden grün hervorgehoben. Weiterhin kann auch eine Glyphe selektiert werden, indem sie mit der linken Maustaste angeklickt wird. In diesem Beispiel wird diese rot hervorgehoben.

Zur selektierten Glyphe werden im *Details-on-Demand-Bereich*, welcher sich rechts vom Overviewbereich befindet, weitere Informationen angezeigt. Neben den allgemeinen Informationen wie beispielsweise den Namen und den Typ (Attribut, Klasse etc.) werden abhängig vom Typ spezielle Informationen angezeigt. So kann der Anwender hier ablesen, wie viele Attribute und Methoden eine Klasse hat und ob sie Subtypen besitzt oder von einer anderen Klasse erbt.

Links neben der Visualisierung befindet sich der sogenannte *Package Explorer*, in welchem die der Visualisierung zugrundeliegende Hierarchie dargestellt wird. Über diesen ist es möglich, einzelne Glyphen auszublenden, indem die Häkchen aus den entsprechenden Kästchen entfernt werden. Der Wurzelknoten dieses Beispiels trägt den Namen *bank*, welcher in der Visualisierung den äußersten Ring darstellt. Auf der nächstniedrigeren Hierarchieebene befinden sich die Knoten *customer*, *products* und *Bank*. Der Zusammenhang zwischen der Hierarchie und der Visualisierung wird im folgenden Kapitel näher betrachtet.

2.2.2 Die Recursive Disc Metaphor

Die Recursive-Disc-Metapher wurde von Müller und Zeckzer entwickelt und stellt eine Softwarevisualisierungsmetapher für das objektorientierte Paradigma dar. Dabei beziehen sie sich auf die Arbeit mit glyphbasierten Visualisierungen von Borgo et al. (vgl. [Müller und Zeckzer 2015]). Die Metapher beschreibt eine kreisförmige Anordnung, wodurch die Elemente platzsparend angeordnet werden können (vgl. [Müller und Zeckzer 2015]). Sowohl Position als auch geometrische Eigenschaften geben Aufschluss über die Struktur der Software und die Attribute der einzelnen Datensätze. So können einige Glyphen andere Glyphen beinhalten, was Aufschluss über die hierarchische Struktur gibt. Weiterhin wird innerhalb der Eigenschaften der Form, Farbe, Transparenz, Größe, Position und Ausrichtung variiert. Abbildung 2-5 zeigt alle derzeit darstellbaren Bestandteile eines Quellcodes. Ein Package (grau) umschließt zwei Klassen (dunkelblau), welche weiterhin Methoden (hellblau) und Attribute (gelb) beinhalten. Mit Hinblick auf Anwender mit Rot-Grün-Sehschwäche wurden die Farben rot und grün nicht in der Standardansicht verwendet, weshalb auf das Blau-Gelb-Schema zurückgegriffen wurde (vgl [Müller und Zeckzer 2015]). Das Package sowie die Klassen sind vollständige Ringe, die Attribute und Methoden hingegen nur Kreis- und Ringsegmente.

Weiterhin bestimmt die Anzahl der Anweisungen einer Methode proportional die Größe der Glyphe, die sie repräsentiert. Attributglyphen besitzen kein solches Kriterium, der gelbe Kern wird zu gleichen Teilen aufgespalten.

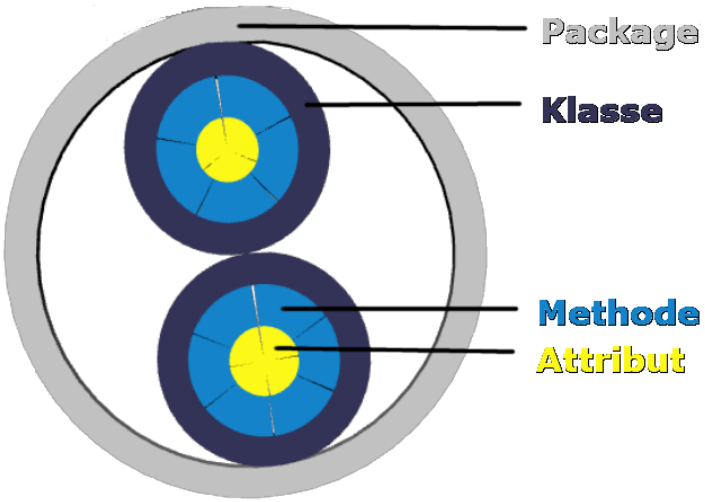

Abbildung 2-5: Visualisierte Quellcodebestandteile

Die Klassenglyphe mit der größten Fläche wird im Zentrum dargestellt. Die Größe einer Klasse wird bestimmt durch die Anzahl ihrer Attribute, die Größe der beinhalteten Methoden und gegebenenfalls die Größe ihrer inneren Klassen (vgl [Müller und Zeckzer 2015]). Um die zentrale Klasse werden alle anderen Klassen der Größe nach absteigend im Uhrzeigersinn angeordnet. Dabei gilt weiterhin, dass die Komplexität einer Klasse bestimmt, wie nahe deren Glyphe am Zentrum ist. Ja größer die Komplexität ausfällt, desto geringer ist der Abstand.

3 Technische Grundlagen

In diesem Kapitel werden alle existierenden Techniken und Konzepte erläutert, welche die Basis der Umsetzung der Ziele dieser Arbeit bilden. Es sind lediglich diese aufgeführt, die auch bei der Implementierung oder im Vorfeld zur vergleichenden Betrachtung herangezogen werden.

3.1 HTML5

Die Hypertext Markup Language (HTML) ist eine textbasierte Auszeichnungssprache, welche die Grundlage zur Erstellung von Webseiten im World Wide Web darstellt. Ursprünglich wurde sie entwickelt, um wissenschaftliche Dokumente semantisch zu beschreiben. Das allgemein gehaltene Konzept der Nutzung von sogenannten Tags wurde auch auf andere Dokumenttypen wie beispielsweise die Extensible Markup Language (XML) angewendet.

HTML ist eine standardisierte Sprache, wobei die Einhaltung und Weiterentwicklung des Standards vom World Wide Web Consortium (W3C) überwacht wird. Derzeitiger Vorsitzender ist Tim Berners-Lee, welcher die Sprache HTML erfand und das Gremium zu dessen Standardisierung 1994 ins Leben rief (vgl. [Longman 1998]). HTML soll verwendet werden, um die Elemente einer Website syntaktisch zu ordnen und deren Eigenschaften durch Attribute zu beschreiben. Dies geschieht durch Auszeichnungen (engl. Markups) innerhalb des Texts, sogenannte *Tags*. Diese grenzen einzelne Elemente voneinander ab und sollen anhand des folgenden Beispiels erklärt werden:

*<abc id=“beispiel“>*xyz*</abc>*

Der Text „xyz“ wird von zwei Tags mit dem Namen „abc“ umgeben, es handelt sich also um ein abc-*Tagpaar* mit dem Inhalt „xyz“. Jedes Tagpaar besteht aus einem *Starttag* (in diesem Fall <abc id=“beispiel“>) und einem *Endtag* (</abc>) und kann sowohl Texte als auch andere Tags beinhalten. Weiterhin können HTML-Elemente bestimmte *Attribute* enthalten, welche weitere Informationen über dieses beinhalten. Attribute werden stets im Start-Tag angegeben. Im obigen Beispiel besitzt das abc-Element das Attribut *id* mit dem Wert „beispiel“.

Durch Schachtelung von Tags ineinander sind HTML-Dokumente stets in einer Baumstruktur gegliedert. Quellcode-Listing 3-1 zeigt ein einfach aufgebautes Dokument nach HTML5-Standard, wie es in der Spezifikation des W3C zu finden ist (vgl. [W3C 2014]). Dieser ist der aktuelle Standard, welcher vom W3C vorgegeben wird.

```
 1 <!DOCTYPE html>
 2 <html>
 3   <head>
 4     <title>Sample page</title>
 5   </head>
 6   <body>
 7     <h1>Sample page</h1>
 8     <p>This is a <a href="demo.html">simple</a> sample.</p>
 9     <!-- this is a comment -->
10   </body>
11 </html>
```

Quellcode-Listing 3-1: Beispiel eines einfachen HTML-Dokuments

Unter anderem erweitert der neue Standard den bis dahin gültigen um viele multimediale Funktionen wie beispielsweise die Darstellung dynamischer 2D- und 3D-Grafiken, natives *Drag and Drop* von Elementen sowie den Zugriff auf lokalen Speicher mit standardisierter Schnittstelle zur Erstellung und Verwaltung von Dateisystemen im Browser. Ohne diese Funktionalitäten wäre die in dieser Arbeit beschriebene Form der Softwarevisualisierung nicht oder nur durch die Nutzung externer Bibliotheken realisierbar.
Die Gestaltung der Webseite sowie das Scripting zur Interaktion mit den Elementen erfolgen nicht durch HTML, sondern in den dafür vorgesehen Sprachen Cascading Style Sheets (CSS) und JavaScript (JS).

Das Laden und Speichern der Teilvisualisierungen geschieht über die File API, welche ebenfalls innerhalb des HTML5-Standards definiert wird. Sie ermöglicht es, Dateien im Browser zu verwalten. Dazu zählt neben dem Speichern, Hochladen und automatisierten Auslesen der Dateien auch das Reservieren von lokalem Speicher sowie das Anlegen und Verwalten eigener Verzeichnisstrukturen. Auf letztere können alle gängigen Operationen wie das Kopieren und Verschieben angewendet werden. Im Zuge dieser Arbeit wird der Fokus bei der Implementierung auf das zur API gehörende FileReader-Objekt gesetzt.

3.2 JavaScript

JavaScript ist eine an die Programmiersprache C angelehnte, typenlose, objektorientierte dynamische Skriptsprache, welche zumeist im Browser Verwendung findet, um das Verhalten der Elemente einer HTML-Seite zu definieren. Sie ist die Skriptsprache des World Wide Web, da die Mehrheit aller Webseiten JavaScript nutzt und sie von allen modernen Browsern unterstützt wird (vgl. [Flanagan 2011]). Entwickelt wurde sie 1995 von Brendan Eich. Im Jahre 1997 wurde sie durch die private non-profit Normungsorganisation Ecma International in den ECMA-262-Standard überführt (vgl. [ECMA 2015]). Aus diesem Grund lautet der eigentliche Name der Skriptsprache nicht JavaScript, sondern ECMAScript und hat mit Java bis auf Ähnlichkeiten in der Syntax wenig gemeinsam.

JavaScript kann sowohl client- als auch serverseitig eingesetzt werden, wobei der clientseitige Einsatz verbreiteter ist und serverseitig andere Skriptsprachen verwendet werden. Ohne JavaScript sind die Möglichkeiten, mit Webseiten zu interagieren, eingeschränkt, da HTML primär zur Erstellung der Syntax und CSS zur Gestaltung der Seite entwickelt wurden. An dieser Stelle wird auf eine Beschreibung der Funktionalitäten von JavaScript verzichtet. Für detailliere Informationen sei dem interessierten Leser das Buch „JavaScript: The Definitive Guide“ von Flanagan empfohlen [Flanagan 2011].
Es gibt zahlreiche Plugins, mit denen der Funktionsumfang von JavaScript erweitert werden kann. Zu den bekanntesten zählt jQuery[1], welches zurzeit im Softwarevisualisierungsprojekt eingesetzt wird. Es wäre daher möglich, diese Bibliothek für die Erfüllung der Ziele zu nutzen. Um jedoch die Abhängigkeit von externen Bibliotheken so gering wie möglich zu halten, soll diese Funktionalität direkt im HTML5-Standard implementiert werden.

Das Verhalten der Oberfläche ist in Javascript geschrieben. Auffällig ist hier die verwendete Architektur: Sämtliche Anweisungen sind in Controllern zusammengefasst. Diese Controller sind globale Variablen, welchen eine besondere Art von Funktion zugewiesen wird, sogenannte *Closures*. Diese fassen zusammengehörige Variablen und Funktionen zusammen und versehen sie mit einem eigenen Namensraum, sodass sie mit einer Klasse einer objektorientierten Sprache vergleichbar sind. Der Zugriff auf die einzelnen Bestandteile eines Controllers erfolgt durch den Punktoperator. Um eine Funktion nach außen sichtbar zu machen, muss sie mit *return* zurückgegeben werden. Controller werden so geschrieben, dass ihr Fehlen sich nicht auf die anderen Funktionalitäten auswirkt. Ihre Einbindung erweitert lediglich den Funktionsumfang. Sie können daher mit Plugins verglichen werden.

Weiterhin orientiert sich das Verhalten an einem Eventsystem. Für verschiedene Aktionen mit der Oberfläche können Events definiert werden. Diese werden zentral allen Controllern weitergeleitet, die sich dafür registriert haben. Auf diese Weise ist es beispielsweise möglich, dass Aktionen im Overview-Bereich weitere Aktionen im Package Explorer und im Details-on-Demand-Bereich auslösen, obwohl alle drei von verschiedenen Controllern gesteuert werden.

Es folgt ein Beispiel des Eventsystems anhand eines Klicks auf eine Glyphe im Overview-Bereich. Quellcode-Listing 3-2 zeigt den Aufruf der Funktion *publishOnEntitySelected(event.partID, “canvas“)* innerhalb des Canvas-Controllers beim

1 http://jquery.com/, gelesen am 16.11.2015

Klicken auf eine Glyphe mit der linken Maustaste (Zeile 3). Diese Funktion ist in der *Application.js* definiert und aktiviert das Event *onEntitySelected*.

```
1 function handleOnClick(event) {
2   (event.button == 1) {
3   publishOnEntitySelected(event.partID, "canvas");
      […]
```

Quellcode-Listing 3-2: Aufruf der Funktion publishOnEntitySelected

Alle Controller, die auf das Eintreten dieses Events warten sollen, müssen die Funktion *subscribeOnEntitySelected(listener)* aufrufen, welche für alle Controller zugänglich ist (Quellcode-Listing 3-3). Dadurch registrieren sie sich dafür und legen fest, welche Funktion beim Eintritt des Events ausgeführt werden soll.

```
1 function subscribeOnEntitySelected(listener) {
2   $(document).on("onEntitySelected", listener);
3 }
```

Quellcode-Listing 3-3: Definition der Funktion subscribeOnEntitySelected

3.3 Drag and Drop

Drag and Drop (dt. „Ziehen und Ablegen“) ist eine verbreitete Möglichkeit, mit grafischen Benutzeroberflächen zu interagieren. Voraussetzung für die Ausführung dieser Operation ist ein Zeigegerät wie beispielsweise eine Maus. Auf Geräten mit Touchscreen erfolgt die Eingabe üblicherweise mit den Fingern oder einem Touchpen.

Die Drag-and-Drop-Operation besteht im Grunde darin, dass das zu ziehende Element angeklickt und durch gedrückt halten der Maustaste zum Zielort gezogen wird. Das Loslassen der Maustaste leitet den Drop ein. Abbildung 1-2 zeigt eine solche Operation am Beispiel eines Bildes, welches in den Ordner „Bilder“ gezogen wird.

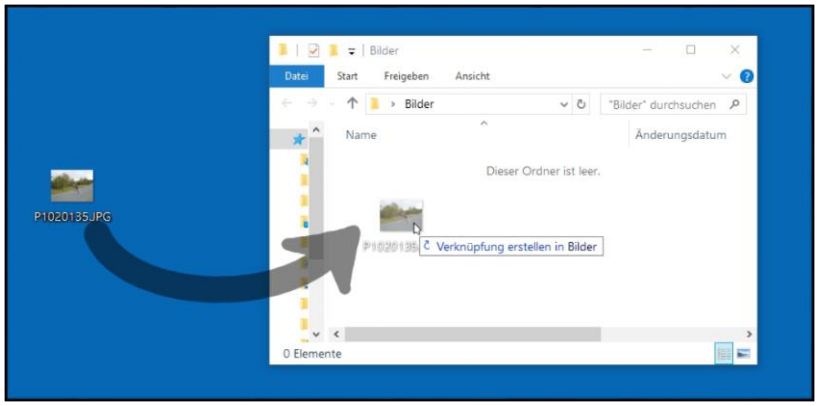

Abbildung 3-1: Einfache Drag-and-Drop-Operation

Meist erhält der Anwender visuelles Feedback während dieser Operation. So folgt zum Beispiel ein halbtransparentes Abbild des gezogenen Elements dem Cursor, während es gezogen wird. Die Bereiche, in welche das Element fallengelassen werden kann, können

außerdem durch einen farbigen Rahmen hervorgehoben werden. Verläuft das Fallenlassen wie vorgesehen, so wird das Element je nach programmiertem Verhalten im Zielbereich angezeigt oder es löst ein anderweitiges Ereignis aus, zum Beispiel das Öffnen einer Anwendung. Bei nicht zielführendem Fallenlassen geschieht entweder nichts und das Element ist weiterhin am Startpunkt zu finden. Gegebenenfalls wird eine Fehlermeldung angezeigt.

Die zu übertragenden Daten werden anders als bei der Copy-and-Paste-Operation nicht in die Zwischenablage (engl. Clipboard), sondern in einen Puffer gespeichert. Sowohl Zwischenablage als auch Puffer sind vom Betriebssystem zur Verfügung gestellte Speicherbereiche, auf die anwendungsübergreifend zugegriffen werden kann. Der Unterschied ist, dass der Inhalt der Zwischenablage solange darin verbleibt, bis er überschrieben wird, während der Puffer kurzlebiger ist. Beispielsweise würden die Daten eines Bildes, welches gerade durch *Drag and Drop* bewegt wird, nur solange im Puffer bleiben, bis es wieder irgendwo abgelegt wird. Danach können diese Daten nicht mehr über den Puffer abgerufen werden (vgl. [o.V. 2015]). Im Gegensatz dazu kann eine Bilddatei, die durch den Befehl *Kopieren* in die Zwischenablage gespeichert wird, solange und beliebig oft eingefügt werden, bis die Zwischenablage erneut beschrieben wird.

In der Vergangenheit wurden mehrere Versuche unternommen, *Drag and Drop* im Browser zu implementieren. So entstanden mehrere externe Bibliotheken, welche diesen Mechanismus vereinfachen. Auch die Bibliothek jQuery verfügt über eine solche Implementierung. Die erste Standardisierung hielt mit der Veröffentlichung des HTML5-Standards Einzug, durch welchen HTML-Elementen nativ das Attribut *draggable* zugewiesen werden kann. Dies erlaubt es dem Anwender, diese Elemente von deren Position wegzuziehen. Weiterhin definiert der Standard Events, welche unter bestimmten Voraussetzungen aktiv sind. Mittels JavaScript-Schnittstelle ist es möglich, darauf zu reagieren. Indem eine Funktion definiert wird, welche an das Event *dragover* geknüpft wird, könnte beispielsweise der Rahmen der Droparea hervorgehoben werden, während sich der Cursor über dieser befindet.

3.4 Point and Click

Point and Click (dt. „[auf etwas] zeigen und klicken“) ist eine weitere, verbreitete Interaktionsmöglichkeit für grafische Benutzeroberflächen. Vorwiegend werden dadurch Elemente selektiert oder Aktionen ausgeführt. Wird mit Hilfe der Maus navigiert, so wird der Cursor auf ein Element bewegt und durch Drücken einer Maustaste angeklickt. Anders als bei der Drag-and-Drop-Operation wird die Taste wieder losgelassen, bevor der Cursor weiter bewegt wird. Bei Computermäusen mit mehreren Tasten haben diese meist unterschiedliche Bedeutung. Während die linke Maustaste oft zur Selektion verwendet wird, öffnet sich bei der Verwendung der rechten Maustaste ein Kontextmenü in unmittelbarer Nähe des Cursors, in welchem elementspezifische Operationen aufgelistet sind. Analog wird bei der Verwendung eines Touchscreens das Element kurz mit dem Finger oder dem Touchpen angetippt und gleich wieder losgelassen. Auch bei *Point and Click* ist es üblich, dem Anwender ein visuelles Feedback zu geben. Das selektierte Objekt wird zumeist farbig umrandet oder eingefärbt dargestellt. Neben dem visuellen Feedback ist es auch möglich, ein auditives Signal zu geben.

3.5 Verwaltung der Struktur der Visualisierung durch zTree

zTree[1] ist ein Plugin für jQuery zur Verwaltung von Baumstrukturen und wird bereits im Projekt verwendet, um die hierarchisch geordneten Knoten der Visualisierung zu verwalten. Knoten, die in der Hierarchie über einem betrachteten Knoten liegen, werden im Folgenden als „Eltern“ oder „Elternknoten“ bezeichnet, untergeordnete Knoten als „Kinder“ oder „Kindknoten“. Die Interaktion mit dem von zTree erzeugten Baum erfolgt hauptsächlich über den bereits vorgestellten Package Explorer. Es handelt sich um Open-Source-Software, welche unter der MIT-Lizenz zur freien Nutzung und Verbreitung freigegeben ist. Die Verwaltung der Knoten geschieht über eine breite Auswahl an Abfrage- und Interaktionsmöglichkeiten, wobei jede Interaktion sowie jeder Statuswechsel ein Event auslöst. Funktionen können darauf programmiert sein, mit der Ausführung bis zum Eintritt eines bestimmten Events zu warten. Im folgenden Quellcode-Listing 3-4 wird auf das Setzen oder Entfernen des Häkchens im Package Explorer reagiert. Das Event, welches bei diesen Aktionen ausgelöst wird, heißt *onCheck*. Die Funktion *zTreeOnCheck(event, treeId, treeNode)* legt eine Referenz auf den Baum mit dem Namen „packageExplorerTree“ an (Zeile 2), welcher zuvor an anderer Stelle initialisiert wurde. Über diese werden alle Knoten, bei denen das Häkchen gesetzt ist, ermittelt und als Array unter *nodes* speichert. Dazu zählen

1 http://www.ztree.me/v3/api.php

auch Knoten, bei denen nur eine Teilmenge ihrer Kinder markiert ist. Der Wert an der fünften Stelle des Arrays soll anschließend über ein Hinweisfenster angezeigt werden.

```
1 function zTreeOnCheck(event, treeId, treeNode) {
2    var treeObj = $.fn.zTree.getZTreeObj("packageExplorerTree");
3    var nodes = treeObj.getCheckedNodes(true);
4    alert(nodes[4].id);
5 }
```

Quellcode-Listing 3-4: Funktion zur Ermittlung der ID eines Knotens mittels zTree

Abbildung 3-2 zeigt das Ergebnis der Ausführung. Im Package Explorer ist zu erkennen, dass einige Listeneinträge nicht markiert sind. Daher werden diese nicht mit in das Array aufgenommen und es werden nur die markierten Einträge (bank, products, AbstractProduct, balance, executedTransactions, inTransaction etc.) ihrer hierarchischen Ordnung nach gespeichert. *executedTransactions* ist von oben nach unten gezählt der fünfte Eintrag, weshalb dessen ID auch in dem Hinweisfenster ausgegeben wird.

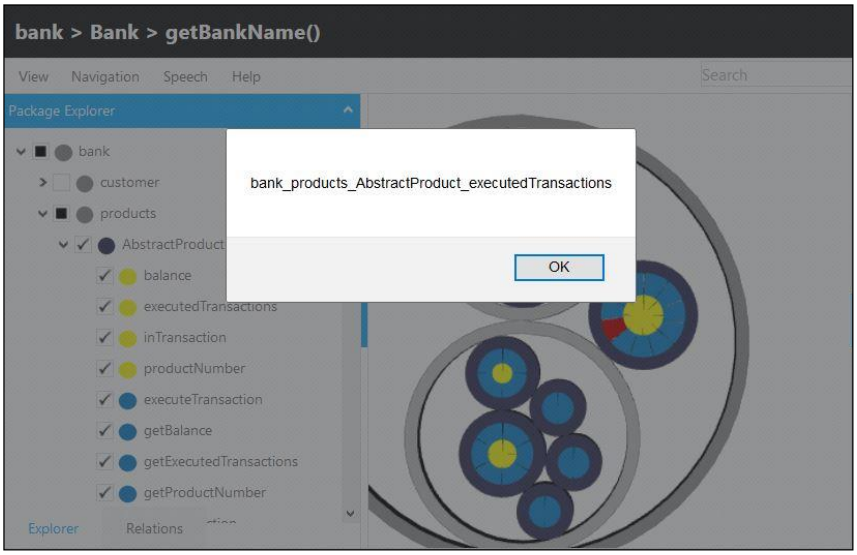

Abbildung 3-2: Ergebnis der Ausführung der Beispielfunktion zTreeOnCheck

Weiterhin besitzt zTree eine eigene Schnittstelle zur Nutzung von *Drag and Drop*. Diese wird jedoch im Zuge dieser Arbeit aus zwei Gründen nicht verwendet: Zum einen würde so die Abhängigkeit vom Plugin erhöht und zum anderen kann auf die Events von zTree nur reagiert werden, wenn *Drag and Drop* innerhalb des Package Explorers angewendet wird. Nun sollen aber die Glyphen der 3D-Visualisierung auch über *Drag and Drop* verschoben werden, welche jedoch von den Listenern des Plugins nicht überwacht werden.

4 Erzeugung von Teilsichten

In diesem Kapitel wird zunächst definiert, was eine Teilsicht im Kontext der hier behandelten Softwarevisualisierung ist. Im Anschluss werden die Vor- und Nachteile der beiden Interaktionsmethoden *Point and Click* und *Drag and Drop* erläutert und die Implementierung der günstigeren Methode dokumentiert.

4.1 Definition einer Teilsicht

Im Rahmen der hier behandelten Softwarevisualisierung resultiert die Teilsicht aus dem interaktiven Ein- und Ausblenden von Glyphen durch das Setzen der Häkchen beim Package Explorer. Abbildung 4-1 zeigt die im Package Explorer dargestellte Baumstruktur einer Softwarevisualisierung. Alle Knoten sind aktiv und werden im Overview dargestellt (Abbildung 4.2). Alle Glyphen können in diesem Zustand betrachtet werden. Es besteht eine vollständige, ungefilterte Sicht auf die Visualisierung.

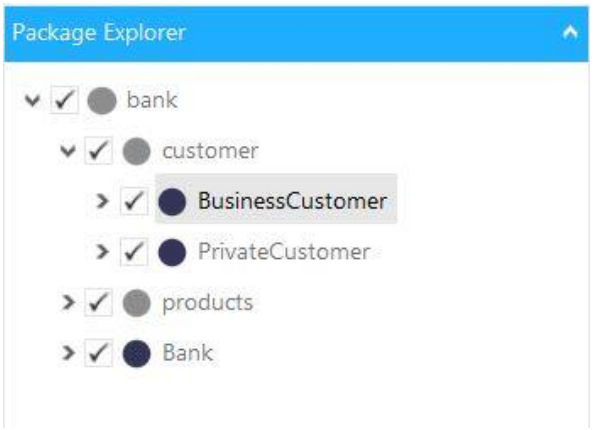

Abbildung 4-1: Vollständig selektierter Package Explorer

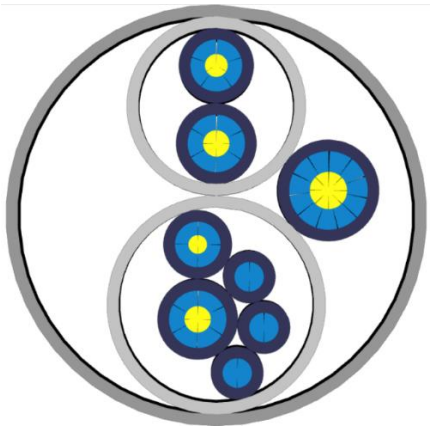

Abbildung 4-2: Visualisierung im Overview

Den Gegensatz dazu bildet die Teilsicht, welche sich durch die Visualisierung einer gefilterten Anzahl Glyphen charakterisiert. Die Informationen zur Visualisierung stehen unverändert in der X3D-Datei, welche vom Generator erstellt wurde. Diese wird auch zu Beginn durch den Browser geladen, wodurch alle Knoten im Backend vorhanden sind. Jedoch wird nur ein Teil davon visualisiert, der Rest ist ausgeblendet. Bisher war diese Filterung zur Darstellungszeit nur durch den Package Explorer möglich. Abbildung 4-3 zeigt den Package Explorer nach vorgenommener Filterung.

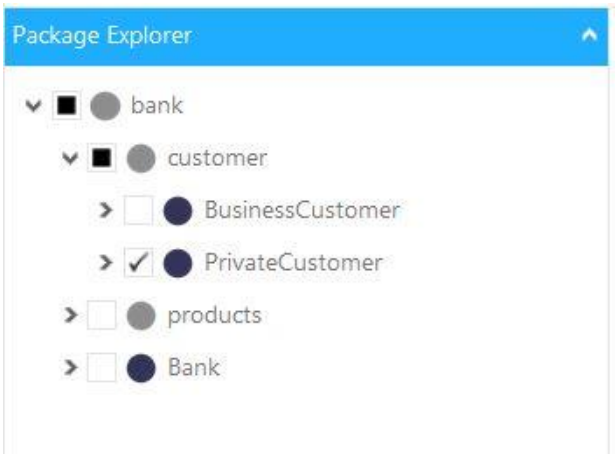

Abbildung 4-3: Teilweise selektierter Package Explorer

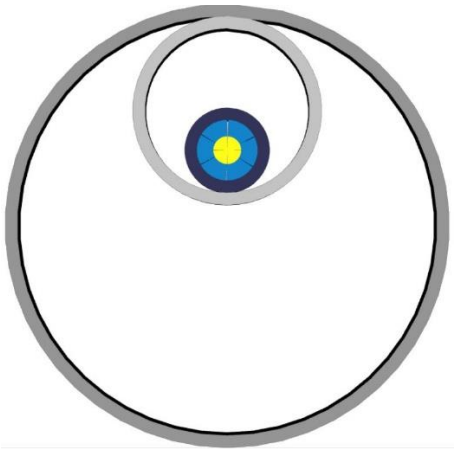

Abbildung 4-4: Teilweise Visualisierung im Overview

Wie zu sehen ist, wurde das Package „products“ und die Klassen „Bank“ und „BusinessCustomer“ ausgeblendet. Die daraus resultierende Visualisierung ist in Abbildung 4-4 zu sehen, welche die Visualisierung der Abbildung 4-2 ohne die ausgeblendeten Klassen darstellt. Es werden neben der verbleibenden Klasse „PrivateCustomer“ auch deren Elternknoten „customer“ und „bank“ beibehalten. Es ist möglich, diese Klasse auch auszublenden. In diesem Falle würden nur die Packages „bank“ und „customer“ visualisiert. Es ist jedoch nicht möglich, eine der Elternknoten auszublenden und die Klasse beizubehalten. Ein Knoten kann nicht ohne dessen hierarchisch übergeordneten Knoten existieren. Dies muss auch in der zu implementierenden Interaktionsmethode beachtet werden. Würde ein Attribut von einem Fenster in das andere übertragen werden, so dürfte es dort nicht allein dargestellt bleiben. Es werden daher stets alle diesem Attribut übergeordneten Knoten zusätzlich dargestellt. Wird ein Knoten übertragen, welcher ebenfalls andere Knoten enthält, so werden diese ihm untergeordneten Knoten ebenfalls übernommen. Dies ist bei Klassen und Packages der Fall.

4.2 Vergleich von Drag and Drop und Point and Click

Es gilt nun zu klären, welche der beiden vorgestellten Interaktionsmethoden geeigneter für die Erstellung der Teilsicht über mehrere Fenster hinweg ist. Die Durchführung dieser Aktion soll möglichst einfach und intuitiv gehalten werden. In der Vergangenheit wurden bereits mehrere Studien durchgeführt, um die Geschwindigkeit und Fehlerrate sowie die Akzeptanz von *Point and Click* und *Drag and Drop* zu vergleichen. Eine Studie aus dem 1996 prüfte eben diese Kriterien anhand von 32 Jungen und 36 Mädchen im Alter von neun bis dreizehn Jahren, welche in zwei Testreihen jeweils Aufgaben mit Hilfe beider Interaktionsmethoden lösen mussten (vgl. [Inkpen et al. 1997]). Die Aufgabe der ersten Testreihe bestand darin, mehrmals eine auf dem Bildschirm angezeigte Box in eine andere

zu transportieren. Die Größe und der Abstand zwischen den Boxen variierte dabei während des Durchlaufs. In der zweiten Testreihe mussten die Kinder einen komplexeren Sachverhalt im Spiel „The Incredible Machine“ lösen, indem sie einen Motor und ein Förderband durch ein elastisches Band verbanden. Glückte diese Aktion, begann sich das Förderband zu bewegen.

Das Verhalten des Bandes unterschied sich beim Versuch, es zu verbinden, je nach verwendeter Interaktionsmethode. Bei *Point and Click* wurde das Band durch Klick auf den Motor selektiert und folgte dem Cursor, auch wenn bei erneutem Klicken das Förderband verfehlt wurde. Dies sei möglich, da diese Methode aus zwei diskreten Aktionen (Klick jeweils zum Aufheben und Ablegen) besteht, wobei das Missglücken einer Aktion nicht die gesamte Bewegung annulliert. Im Gegensatz dazu kehrte das Band, welches bei der Drag-and-Drop-Methode mit gedrückter Maustaste gezogen wurde, bei nicht geglücktem Ablegen wieder in seine Ausgangsposition zurück. Dieser Unterschied wurde im dazugehörigen Paper mit einer analogen Problemstellung erläutert: Der Versuch, Motor und Förderband zu verbinden ähnelt dem, ein Seil zwischen zwei Bäumen zu spannen. Das Seil wird an einem Ast des ersten Baumes befestigt und dann zum zweiten getragen, wo es an einen seiner Äste geworfen wird. Missglückt dies, wird im Falle von *Point and Click* das Seil aufgefangen und ein weiterer Versuch gestartet, da das Missglücken der Ablegen/Werfen-Aktion nicht das gesamte Vorhaben rückgängig macht. Im Falle von *Drag and Drop* wird das Seil, wenn es an inkorrekter Position abgelegt wird, nicht aufgefangen, wodurch der komplette Vorgang wiederholt werden muss.

Das Ergebnis dieser Studie zeigt signifikante Unterschiede zwischen beiden Interaktionsmethoden. *Point and Click* wurde von den Mädchen bevorzugt und sie führten die Aufgaben mit Hilfe dieser Methode auch schneller und mit weniger Fehlern aus als mit *Drag and Drop*. Auch die Jungen führten die Aufgaben mit Point-and-Click-Methode schneller aus. Fehlerrate und Präferenz ergaben bei ihnen keine signifikanten Unterschiede. Nun wird erörtert, ob die Rahmenbedingungen dieser Studie auf die geplante Anwendung in der Softwarevisualisierung anwendbar sind, denn obwohl *Point and Click* als zu bevorzugende Interaktionsmethode aus der Studie hervorgeht, so muss sie nicht auf alle Benutzeroberflächen anwendbar sein. Dabei wird davon ausgegangen, dass bei der Verwendung von *Point and Click* mindestens ein weiterer Button in die Benutzeroberfläche der Softwarevisualisierung zu integrieren wäre, um die Informationen zum selektierten Objekt sowohl in die Zwischenablage zu speichern als auch wieder von dort zu laden.

Ausgehend vom Aufbau der Benutzeroberfläche würde es einfacher sein, die Glyphen der Visualisierung in ein anderes Fenster zu transferieren als ein Band zwischen Motor und Förderband zu spannen, abhängig davon, wie groß der Zoom auf die Glyphen zum Zeitpunkt der Akquise ist. Durch Fitt's Gesetz wird die Korrelation zwischen der Größe der Zielfläche, der Distanz zu dieser und der dadurch resultierenden, benötigten Zeit ausgedrückt (vgl. [MacKenzie et al. 1991]). Die Boxen, Motoren und Fließbänder, mit welchem die Kinder in der Studie interagierten, nahmen vergleichsweise wenig Platz auf dem Bildschirm ein, wodurch es schwieriger war, den Cursor auf die gewünschte Fläche zu bewegen. Dies gilt sowohl für *Point and Click* als auch für *Drag and Drop*, wodurch bei der Glyphenakquise kein Unterschied festzustellen ist. Jedoch gewinnt *Drag and Drop* bei den darauffolgenden Aktionen, denn sobald die Glyphe akquiriert wurde, kann sie sogleich in das nächste Fenster mit geöffneter Softwarevisualisierung gezogen werden. Dabei kann dort das gesamte body-Element, also die gesamte Fläche der angezeigten HTML-Seite, als Droparea definiert werden, wodurch die Fläche des zweiten Ziels maximiert wird. Ausgehend von Fitts' Gesetz wird die für die Aktion benötigte Zeit verringert. Da es innerhalb der HTML-Seite des Zielfensters auch keinen Bereich gibt, in welchem das Ablegen des Elements nicht akzeptiert wird, schrumpft die Fehlerwahrscheinlichkeit durch fehlerhafte Drops. Bei der Verwendung von *Point and Click* müsste nach der Elementselektion im Ausgangsfenster der Button zur Speicherung der Glypheninformation in die Zwischenablage angeklickt werden, um sie anschließend im Zielfenster durch betätigen eines weiteren Buttons zu laden. Da die Buttons – bedingt durch den eingeschränkten Platz – nicht beliebig groß gewählt werden können, wird mehr Zeit benötigt, um sie anzusteuern. Hinzukommt, dass drei diskrete Aktionen ausgeführt werden müssen: Glyphe auswählen, Speichern-Button anklicken und Laden-Button anklicken. Dadurch übersteigt die Länge des zurückzulegenden Weges die der Drag-and-Drop-Methode.

Ein weiterer Nachteil der Point-and-Click-Methode stellt die geringe Nutzerfreundlichkeit dar, die durch die drei beinhalteten Aktionen hervorgerufen wird. Da die Kommunikation über mehrere Fenster hinweg möglich sein soll, bleibt jedoch an dieser Stelle keine andere Wahl, da ein externer Speicherort (in diesem Falle der Zwischenspeicher des Rechners) benötigt wird. Im Gegensatz dazu ist die Drag-and-Drop-Methode einfacher gehalten: Es wird die Glyphe ausgewählt, in das Zielfenster gezogen und dort an einem beliebigen Ort abgelegt. Die Kraft, die beim Gedrückthalten der Maustaste aufgewendet werden muss, wird oft als Vorteil der Drag-and-Drop-Methode gesehen, da der Anwender durch die Anspannung des Fingers das Gefühl bekommt, etwas festzuhalten.

Buxton schreibt dazu: „*In well-structured manual input there is a **kinesthetic connectivity** to reinforce the **conceptual connectivity** of the task*." [Buxton 1986, Hervorhebungen im Original]

Visuelle Hervorhebungen während des Dragvorgangs, beispielsweise die farbliche Hervorhebung des Zielfensters oder das Einblenden einer halbtransparenten Vorschau am Cursor, können zusätzlich zur kinästhetischen Wahrnehmung die Durchführung der Aktion verbessern, indem Fehler vermieden werden. So schreiben Sellen et al.: "*[By] providing sensory feedback, a common class of error (mode errors) can be significantly reduced for both novices and experts. Both visual and kinesthetic feedback can significantly improve performance.*" [Sellen et al. 1992]

4.3 Implementierung für den Package Explorer

Die visualisierten Glyphen besitzen je nach gewählter Metapher unterschiedliche Formen und sind jeweils auf andere Art und Weise angeordnet und geschachtelt. Während beispielsweise die Recursive Disc Metaphor zwar im dreidimensionalen Raum dargestellt wird, ist ihre Grundstruktur zweidimensional. Dadurch kann jede Glyphe ohne weiteres selektiert werden. Andere Metaphern schachteln ihre Glyphen jedoch vollständig, sodass sich Glyphen innerhalb des Körpers einer anderen befinden, was die Selektion erschwert. Aufgrund dessen ist es derzeit nicht möglich, eine einheitliche Anwendung von Drag-and-Drop auf die Visualisierung zu gewährleisten.

Um die Informationen dennoch von einem Fenster in ein anderes ziehen zu können, wird auf die Listeneinträge des Package Explorers zurückgegriffen. Der Grund dafür ist, dass der Package Explorer ebenfalls sämtliche Knoten enthält und unabhängig von der gewählten Metapher verfügbar ist. Beim Ziehen des Listeneintrags soll dessen ID ausgelesen und als zu übertragende Information weitergegeben werden. Beim Fallenlassen soll diese ID eingelesen und verwendet werden, um den entsprechenden Knoten einzublenden.

Dieser Prozess benötigt einige Vorbereitung. Quellcode-Listing 4-1 zeigt die ersten Schritte: Zunächst wird in Zeile 1 die Variable *dragData* initialisiert, welche später die ID speichert und an den Puffer weitergegeben wird. Die Funktion *initialize()* sucht die benötigten HTML-Elemente anhand ihrer ID heraus und weißt sie an, auf bestimmte Events zu reagieren.

```
 1 var dragData = null;
 2
 3 function initialize() {
 4
 5   var dragDestination = document.body;
 6   var packageExplorerDragSource = document.getElementById(
 7     "packageExplorerTree_1");
 8
 9   packageExplorerDragSource.addEventListener(
10     'dragstart', handlePackageExplorerDragStart, false);
11
12   dragDestination.addEventListener(
13     'dragover', handleDragOver, false);
14   dragDestination.addEventListener(
15     'dragenter', handleDragEnter, false);
16   dragDestination.addEventListener(
17     'dragleave', handleDragLeave, false);
18   dragDestination.addEventListener(
19     'drop', handleDrop, false);
20
21   subscribeOnEntityEnter(onEntityEnter);
22   subscribeOnEntityLeave(onEntityLeave);
23 }
```

Quellcode-Listing 4-1: Vorbereitung der EventListener

Zunächst wird in Zeile 1 die Variable *dragData* initialisiert, welche später die ID speichert und an den Puffer weitergegeben wird. Die Funktion *initialize()* sucht die benötigten HTML-Elemente anhand ihrer ID heraus und weißt sie an, auf bestimmte Events zu reagieren. So wird dem Package Explorer in Zeile 9f angewiesen, auf den Beginn der Dragbewegung zu reagieren, indem er die Funktion *handlePackageExplorerDragStart* aufruft. Dem *body* werden in Zeile 12 bis 17 EventListener für das Eintreten, Verweilen und Verlassen des Cursors sowie für das Fallenlassen der Information im Puffer (Zeile 18f) zugewiesen. Die Funktion *initialize()* wird beim Laden des Skripts aufgerufen.

Leider besteht an dieser Stelle das Problem, dass nicht alle HTML-Elemente standardmäßig gezogen werden können. Diese Eigenschaft muss für die Listeneinträge nachträglich vergeben werden. Daher folgt an dieser Stelle die Betrachtung ihrer Struktur. Abbildung 4-5 zeigt rechts einen Listeneintrag und links den entsprechenden HTML-Code. Er besteht aus einem Hyperlink (*a*), welcher noch zwei span-Elemente enthält: Den Kreis in typspezifischer Farbe und den Namen des Knotens. Der Hyperlink ist wiederrum eingebettet in einen Listeneintrag (*li*) einer unsortierten Liste (*ul*).

Abbildung 4-5: HTML-Code eines Listeneintrags und dessen Darstellung

Hyperlinks können standardmäßig gezogen werden, span-Elemente nicht. Daher kann das Ziehen des Listeneintrags nur durchgeführt werden, wenn sich der Cursor beim Beginn der Ziehbewegung in der Fläche zwischen Kreis und Text befindet. Da diese Fläche nicht sehr groß ist und daher genau gezielt werden muss, ist diese Form der Interaktion fehleranfällig und zeitraubend. Die Lösung besteht darin, den Hyperlink jedes Listeneintrags manuell mit dem Attribut *draggable* zu versehen. Da die *span*-Elemente Kinder des Hyperlinks sind, wird diese Eigenschaft an sie weitervererbt. Die Schwierigkeit hier ist, dass beim Laden der Oberfläche nur das root-Element angezeigt wird und alle hierarchisch untergeordneten Knoten erst durch einen Klick auf den nebenstehenden Pfeil sichtbar werden. Aus diesem Grund muss das Setzen des Attributs *draggable* mit jeder Aktualisierung des Baumes geschehen.

```
 1 function zTreeOnExpand() {
 2   // prepares the Package Explorer List Items to be draggable
 3   var root = document.getElementById("packageExplorerTree");
 4   makeDraggable(root);
 5 }
 6
 7 function makeDraggable(currentNode) {
 8   for (var i = 0; i < currentNode.children.length; i++) {
 9     child = currentNode.children[i];
10     switch (child.tagName) {
11       case "UL": makeDraggable(child); break;
12       case "LI": makeDraggable(child); break;
13       case "A": child.setAttribute("draggable", "true");
14     }
15  }
16 }
```

Quellcode-Listing 4-2: Listeneinträge mit Attribut „draggable“ versehen

Quellcode-Listing 4-2 zeigt, wie der Sachverhalt gelöst wurde: Die Funktion *zTreeOnExpand()* wird immer dann aufgerufen, wenn der Baum aufgeklappt wird, um weitere Listeneinträge sichtbar zu machen. Darin wird der oberste Listeneintrag selektiert (Zeile 3) und als Parameter der Funktion *makeDraggable(currentNode)* übergeben. Diese wiederum durchläuft alle angezeigten Listeneinträge und deren Kind-Elemente und prüft, um welche Art von HTML-Tag es sich handelt. Bei unsortierten Listen und deren

Listeneinträgen (Zeile 11f) dringt die Funktion eine Hierarchieebene tiefer vor. Stößt der Suchlauf auf ein Hyperlinkelement, so wird dessen Attribut *draggable* auf „true“ gesetzt (Zeile 13). Um auf das Ausklappen des Package Explorers reagieren zu können, wurden beide Funktionen nicht innerhalb des Drag-and-Drop-Controllers, sondern im Package-Explorer-Controller geschrieben. Dieser beinhaltet die Initialisierung des Package Explorers (*prepareTreeView(),* hier nicht aufgeführt), welche den callback-Events wie beispielsweise *onExpand* die entsprechenden Funktionen zuordnet. Einen zweiten Baum innerhalb des Drag-and-Drop-Controllers zu initialisieren und die Daten des eigentlichen Baumes zu übernehmen, führte zu Fehlern. Eine Alternative für eine zukünftige Überarbeitung ist es, die callback-Events des Baumes an die *Application.js* weiterzureichen, um sie dem Drag-and-Drop-Controller zur Verfügung zu stellen.

Nun, da alle Listeneinträge gezogen werden können, wird festgelegt, was bei diesem Vorgang geschehen soll. Da es das Ziel ist, die ID des Knotens zu übertragen, muss zunächst festgestellt werden, welcher Listeneintrag ausgewählt wurde. Da vom Namensraum des Drag-and-Drop-Controllers nicht auf die Information, auf welchen Listeneintrag geklickt wurde, zugegriffen werden kann, ist auch diese Funktion innerhalb des Package-Explorer-Controllers implementiert. Diese trägt den Namen *zTreeOnMouseDown(event, treeId, treeNode)* und reagiert auf das Herunterdrücken der linken Maustaste. Abbildung 4-3 zeigt diese Funktion, welche die ID des korrespondierenden Knotens ausliest und in Zeile 3 durch eine vorbereitete Schnittstelle in die Variable *dragData* des Drag-and-Drop-Controllers schreibt.

```
1 function zTreeOnMouseDown(event, treeId, treeNode) {
2   if (treeNode) {
3     dragAndDropController.setDragData(treeNode.id);
4   }
5 }
```

Quellcode-Listing 4-3: Speichern der ID bei Herunterdrücken der Maustaste

Diese Variable wird nun weiterverwendet, indem sie in den Puffer der Dragoperation geschrieben wird. Quellcode-Listing 4-4 enthält die Funktion *handlePackageExplorerDragStart(e),* welche auf das Bewegen des Cursors bei gedrückter linker Maustaste innerhalb das Package-Explorer-Baumes reagiert. In dieser Funktion wird in Zeile 2 das Verhalten beim Drag-and-Drop-Vorgang festgelegt. Es wird nur das Bewegen des Elements erlaubt („move“). Das Kopieren sowie das Verlinken sind ebenfalls möglich, aber an dieser Stelle nicht nötig. In Zeile 3f wird der MIME-Type der zu transferierenden Daten sowie deren Inhalt festgelegt. Als Inhalt dient die ID des Knotens, welche in der Variable *dragData* gespeichert ist. Auffällig ist hier, dass der Funktion *getDragData()* die

Controllerinstanz *dragAndDropController* vorangestellt werden muss, obwohl sich sowohl *getDragData()* als auch *handlePackageExplorerDragStart()* innerhalb des Drag-and-Drop-Controllers befinden. Hier wird eine Eigenheit von JavaScript umgangen, welche zur Folge hat, dass Variablen, die von einem anderen Controller aus belegt werden, dessen Namensraum annehmen. Das ist auch mit der Variable *dragData* geschehen, als sie vom Package-Explorer-Controller aus durch die Funktion *setDragData(treeNode.id)* überschrieben wurde.

```
 1 function handlePackageExplorerDragStart(e) {
 2   e.dataTransfer.effectAllowed = 'move';
 3   e.dataTransfer.setData('text/object',
 4                           dragAndDropController.getDragData());
 5 }
 6
 7 function handleDragOver(e) {
 8   if (e.preventDefault) {
 9     e.preventDefault();
10   }
11
12   e.dataTransfer.dropEffect = "move";
13   return false;
14 }
```

Quellcode-Listing 4-4: Behandlung des Dragvorgangs

Da die ID nun innerhalb des Puffers der Drag-and-Drop-Operation gespeichert wird, muss sie noch ausgelesen werden. Solange sich der Cursor innerhalb des *body* befindet, ist die Funktion *handleDragOver(e)* aktiv. Diese verhindert in Zeile 9 mit *e.preventDefault()* alle Aktionen, die der Browser beim Fallenlassen standardmäßig ausführen würde, beispielsweise das Navigieren zu einem Link oder das Aufrufen eines Bildes. Das Aussehen des Cursors beim Navigieren über die Dropzone wird Zeile 12 festgelegt, indem das Attribut *dropEffect* mit „move“ belegt wird.

Nun wird die ID erfolgreich in den Zielbereich übertragen. Es fehlt lediglich die Definition des Verhaltens beim Fallenlassen. Diese erfolgt in der *handleDrop(e)*-Funktion, welche beim Loslassen der Maustaste aufgerufen wird. In den Zeilen 3 bis 6 des Quellcode-Listings 4-5 werden die benötigten Variablen initialisiert. *treeObj* wird verwendet, um mit dem zTree zu kommunizieren. In der Variable *dropData* wird die ID gespeichert, die vorher durch die Drag-and-Drop-Operation im Puffer gespeichert wurde. *currentNode* speichert den zugehörigen Knoten, welcher über die zTree-Funktion *getNodeByParam* und der ID aus *dropData* ermittelt wird. In diese werden alle IDs der Knoten gespeichert, die durch die Operation selektiert und dargestellt werden sollen.

```
 1 function handleDrop(e) {
 2
 3   var treeObj = $.fn.zTree.getZTreeObj("packageExplorerTree");
 4   var currentNode = treeObj.getNodeByParam("id", dropData[0], null);
 5   var dropData = [e.dataTransfer.getData('text/object')];
 6   var finalData = [];
 7
 8   // Add all parents to finalData
 9   finalData.push(currentNode.id);
10   while (currentNode.getParentNode() != null) {
11     finalData.push(currentNode.getParentNode().id);
12     currentNode = currentNode.getParentNode();
13   }
14
15   currentNode = treeObj.getNodeByParam("id", dropData[0], null);
16   getAllChildren(currentNode, finalData);

     […]
```

Quellcode-Listing 4-5: Implementierung von handleDrop(e)

Nun wird das Array *finalData* befüllt. Als ersten Eintrag erhält es die ID von *currentNode* (Zeile 11) und durch die folgende while-Schleife alle Elternknoten. Anschließend wird *currentNode* wieder mit der ursprünglichen ID befüllt und die Funktion *getAllChildren(currentNode, finalData)* aufgerufen, welche alle Kinder das korrespondierenden Knotens sucht und in *finalData* speichert. Quellcode-Listing 4-6 zeigt diese Funktion im Detail.

```
1 function getAllChildren(currentNode, finalData) {
2   if (currentNode.isParent) {
3     for (var i = 0; i < currentNode.children.length; i++) {
4       finalData.push(currentNode.children[i].id);
5       getAllChildren(currentNode.children[i], finalData);
6     }
7   }
8 }
```

Quellcode-Listing 4-6: Rekursive Ermittlung aller Kindsknoten

Diese Funktion arbeitet rekursiv. Jeder Knoten, der in der Hierarchie unter dem ursprünglich übergebenen Knoten liegt, wird in die Liste aufgenommen und wiederum überprüft, ob er Kindknoten besitzt. Es befinden sich nun sowohl die ID des über *Drag and Drop* übertragenen Knotens als auch die IDs seiner Eltern- und Kindknoten im Array *finalData*. Quellcode-Listing 4-7 enthält die folgenden Schritte zur Selektierung der Knoten im Zielfenster. *finalData* wird in einer for-Schleife (Zeile 2ff) iterativ durchlaufen. Anhand der IDs werden die Knoten durch die zTree-Funktion *treeObj.checkNode(node, true)* selektiert. Diese Änderungen haben allerdings noch keine Auswirkungen auf die Visualisierung im Overview-Bereich oder den Package Explorer. Um diese zu aktualisieren, werden die geänderten Knoten ermittelt und in die Variable *nodes* gespeichert (Zeile 7). Die folgende for-Schleife (Zeile 9ff) ist eine Maßnahme, um Fehler in der Darstellung zu vermeiden, die beim abwechselnden Gebrauch von *Drag and Drop* und manueller Modifikation des

Package Explorers auftreten. Nun wird die Funktion *publishOnVisibilityChanged* mit *finalData* als Parameter aufgerufen. Dadurch sind die Änderungen im zTree jetzt auch im Package Explorer und der Visualisierung sichtbar.

```
     […]

 1   var node = null;
 2   for (var i = 0; i < finalData.length; i++) {
 3     node = treeObj.getNodeByParam("id", finalData[i]);
 4     treeObj.checkNode(node, true);
 5   }
 6
 7   var nodes = treeObj.getChangeCheckedNodes();
 8
 9   for(i = 0; i < nodes.length; i++) {
10     nodes[i].checkedOld = nodes[i].checked;
11   }
12
13   publishOnVisibilityChanged(finalData, true, "packageExplorerTree");
14   return false;
15 }
```

Quellcode-Listing 4-7: Implementierung von handleDrop(e) (Fortsetzung)

4.4 Implementierung für die Visualisierung

Eines der Ziele dieser Bachelorarbeit war es, die Drag-and-Drop-Operation auch für die im Overview-Bereich angezeigten Glyphen zu ermöglichen. In Kapitel 4.3 wird erläutert, dass diese Funktionalität aufgrund der Unterschiede im Aufbau von Visualisierungsmetaphern nicht für alle Anwendungsfälle geeignet ist. Da es aber Metaphern gibt, die die direkte Selektion von Glyphen erlauben, soll diese Möglichkeit dennoch implementiert werden. Die Drop-Funktion verrichtet dieselben Aufgaben wie bei der Anwendung von *Drag and Drop* auf den Package Explorer, daher kann diese hier wiederverwendet werden. Der Unterschied besteht lediglich in der Akquirierung der ID, die übertragen werden soll.

Da die 3D-Szene in einem HTML-Canvas dargestellt wird, muss dieser auch als Listener für das Dragevent festgelegt werden. Dies wird umgesetzt, indem die Funktion *initialize()* um die Codezeilen aus dem Quellcode-Listing 4-8 ergänzt wird. Analog zum Package Explorer wird auch hier eine Referenz auf das HTML-Element gespeichert und anschließend als Listener für das Event *dragstart* festgelegt.

```
 1   var canvasDragSource =
 2       document.getElementById("packageExplorerTree_1");
 3
 4   packageExplorerDragSource.addEventListener(
 5       'dragstart', handlePackageExplorerDragStart, false);
```

Quellcode-Listing 4-8: Canvas als Listener festlegen

Weiterhin wird festgelegt, wie sich die 3D-Szene vor und während des Zieh-Vorgangs verhält. Das Ziehen der Visualisierung soll nur möglich sein, wenn sich der Cursor zu Beginn der Operation über einer Glyphe befindet. Tut er dies nicht, so soll das Bewegen der Maus bei gedrückter Maustaste wie bisher als Drehen der Visualisierung interpretiert werden. Um das zu realisieren, wird die Navigation der Visualisierung für den Zeitraum, in welchem sich der Cursor über eine Glyphe befindet, deaktiviert. Weiterhin ist es notwendig, das Attribut *draggable* des HTML-Canvas, in welchem die Visualisierung dargestellt wird, auf *true* zu setzen.

Quellcode-Listing 4-9 zeigt die entsprechende Umsetzung. Die Funktion *onEntityEnter(event, entity)* wird aufgerufen, sobald der Cursor über eine beliebige Glyphe fährt. Innerhalb dieser Funktion wird in Zeile 3 das Attribut *draggable* des HTML-Canvas auf *true* gesetzt. In Zeile 5 wird die ID der unter dem Cursor befindlichen Glyphe in die Variable *dragData* gespeichert. Diese wird auch für die Anwendung von *Drag and Drop* des Package Explorers verwendet. Anschließend wird eine Referenz auf das HTML-Element *navigationInfo* in eine namensgleiche Variable gespeichert. Dieses Element ist an der Visualisierung beteiligt und bestimmt maßgeblich dessen Verhalten. Um die Navigation innerhalb der Visualisierung zu unterbinden, werden die Attribute *type* und *explorationmode* auf „none“ gesetzt (Zeile 8f).

```
 1 function onEntityEnter(event, entity) {
 2   var canvas = document.getElementById("x3dom-x3dElement-canvas")
 3   canvas.setAttribute("draggable",true);
 4
 5   setDragData(entity.id);
 6
 7   var navigationInfo = document.getElementById("navigationInfo");
 8   navigationInfo.setAttribute("type","none");
 9   navigationInfo.setAttribute("explorationmode","none");
10 }
```

Quellcode-Listing 4-9: Ziehen der Visualisierung erlauben

Die Funktion *onEntityLeave()*, welche aufgerufen wird, sobald sich der Cursor nicht mehr über einer Glyphe befindet, setzt die eben vergebenen Werte auf analoge Weise zurück. Sie ist im Quellcode-Listing 4-10 zu sehen. So erhält das HTML-Canvas in Zeile 3 den Ausgangswert *false* und die Attribute des Elements *navigationInfo* erhalten die Werte, die die uneingeschränkte Navigation der Visualisierung ermöglichen (Zeile 7f).

```
1 function onEntityLeave() {
2   var canvas = document.getElementById("x3dom-x3dElement-canvas")
3   canvas.setAttribute("draggable",false);
4
5   var navigationInfo = document.getElementById("navigationInfo");
6
7   navigationInfo.setAttribute("type","examine");
8   navigationInfo.setAttribute("explorationmode","all");
9 }
```

Quellcode-Listing 4-10: Ziehen der Visualisierung unterbinden

Klickt der Anwender jetzt auf eine Glyphe und bewegt die Maus, interpretiert der Browser diese Operation als Ziehen und löst das *dragstart*-Event für das HTML-Canvas aus. Daraus folgt der Aufruf der Funktion *handleCanvasDragStart(e)*, welche im Quellcode-Listing 4-11 zu sehen ist.

```
1 function handleCanvasDragStart(e) {
2   e.dataTransfer.effectAllowed = 'move';
3   e.dataTransfer.setData('text/object', getDragData());
4 }
```

Quellcode-Listing 4-11: Weitergabe der Glyph-ID an den Puffer

Diese Funktion erfüllt die gleiche Funktionalität wie die Funktion *handlePackageExplorerDragStart(e)* mit dem Unterschied, dass die Funktion *getDragData()* nicht mehr mit vorangestelltem *dragAndDropController* aufgerufen werden muss, da die Variable *dragData* im selben Namensraum beschrieben wurde. Ab diesem Punkt sind keine weiteren Implementierungen mehr nötig. Das Canvas-Element kann nun in ein anderes Fenster gezogen werden, wobei die ID der zuletzt überfahrenen Glyphe übertragen wird. Das Fallenlassen im Zielfenster leitet dann die Funktion *handleDrop(e)* ein, welche den Rest des Transfers übernimmt. In der Therorie funktioniert der Drag-and-Drop-Vorgang nun. Anschließende Tests zeigen aber das Gegenteil: Die Visualisierung lässt das Ziehen nicht zu. Ein Blick in die zum Browser gehörige Konsole gibt Aufschluss. Es erscheint folgende Fehlermeldung:

TypeError: canvasDragSource is null

canvasDragSource speichert die Referenz auf den HTML-Canvas und wird in der Funktion *initialize()* initialisiert. Der Canvas existiert nicht in der zugrundeliegenden HTML-Datei, sondern wird erst beim Laden der Oberfläche durch den Canvas-Controller erstellt, welcher von der Application.js noch vor dem Drag-and-Drop-Controller aufgerufen wird. Aus diesem Grund sollte das Element zum Zeitpunkt des Aufrufs des Drag-and-Drop-Controllers bereits existieren. Dies ist aber nicht der Fall. Durch Nutzung des browsereigenen Debuggers wird ein Breakpoint gesetzt, um die Abarbeitung der Skripte zum Zeitpunkt der

Initialisierung der Variable *canvasDragSource* zu stoppen. Es zeigt sich keine Auffälligkeit. Werden die Skripts nun fortgesetzt und die Seite vollständig geladen, erscheint die eben erwähnte Fehlermeldung nicht mehr und das Canvas lässt das Ziehen zu. Der Grund dafür scheint darin zu liegen, dass - obwohl der Canvas-Controller eher aufgerufen wird - das Canvas verzögert geladen wird, So steht es zum Zeitpunkt der Initialisierung der Variable im Drag-and-Drop-Controller nicht zur Verfügung. Durch das Setzen des Breakpoints wird der Erstellung des Canvas genug Zeit zu geben, wodurch es nun zur Verfügung steht.

Diese Lösung stellt jedoch nur einen Workaround dar. Langfristig ist es nötig, ein eigenes Event zu implementieren, welches aktiv wird, sobald der Canvas vollständig geladen wurde. Bis zum Eintreten dieses Events wird der Drag-and-Drop-Controller nicht geladen. Die Umsetzung wird im Folgenden erläutert. Die Änderungen in der Application.js sind im Quellcode-Listing 4-12 zu sehen. In Zeile 1 wird das Event definiert und „onCanvasPrepared“ genannt. Weiterhin wird in derselben Datei in die Funktion *subscribeOnCanvasPrepared* definiert, für die sich die anderen Controller registrieren können (Zeile 3f). Sie sorgt dafür, dass dem Event eine Funktion zugeordnet werden kann, deren Ausführung das Auslösen des Events voraussetzt. Um das Event von einem Controller auslösen zu können, wird in Zeile 7 die Funktion *publishOnCanvasPrepared* definiert.

```
 1 var onCanvasPreparedEvent = $.Event("onCanvasPrepared");
 2
 3 function subscribeOnCanvasPrepared(listener) {
 4     $(document).on("onCanvasPrepared", listener);
 5 }
 6
 7 function publishOnCanvasPrepared(sender) {
 8    onCanvasPreparedEvent.sender = sender;
 9    $(document).trigger(onCanvasPreparedEvent);
10 }
```

Quellcode-Listing 4-12: Implementieren des Events zur Vollendung des Canvas

Die Funktion *publishOnCanvasPrepared* wird am Ende der *prepare*-Funktion des Canvas-Controllers aufgerufen, um die Vollendung des Canvas zu signalisieren. Der Drag-and-Drop-Controller verwendet die Funktion *subscribeOnCanvasPrepared*, um aus dieses Event reagieren zu können. Die Funktion, die als Listener festgelegt wird, trägt den Namen *onCanvasPrepared* und wird in Quellcode-Listing 4-13 gezeigt. Innerhalb dieser befindet sich nun in Zeile 2 die Speicherung des Canvas in die Variable canvasDragStart. Anschließend wird er in Zeile 4 als Listener für das Dragstart-Event eingesetzt.

```
1 function onCanvasPrepared() {
2    canvasDragSource =
3        document.getElementById("x3dom-x3dElement-canvas");
4    canvasDragSource.addEventListener(
5        'dragstart', handleCanvasDragStart, false);
6 }
```

Quellcode-Listing 4-13: Verwendung des Canvas als Eventlistener

Abbildung 4-6 zeigt das Ergebnis der Implementierung. Der linke Teil stellt ein Browserfenster dar, von welchem eine Glyphe gezogen wird. Am Cursor ist die gesamte Visualisierung halbtransparent dargestellt. Der rechte Teil der Abbildung zeigt ein zweites Browserfenster mit geöffneter Oberfläche. Es wurden alle Knoten deselektiert, sodass der Overview-Bereich keine Glyphen enthält. Anschließend wurden die Informationen aus dem anderen Fenster darauf fallengelassen, sodass nun die zuvor gezogene Glyphe mitsamt aller hierarchisch über ihr liegenden Glyphen dargestellt wird.

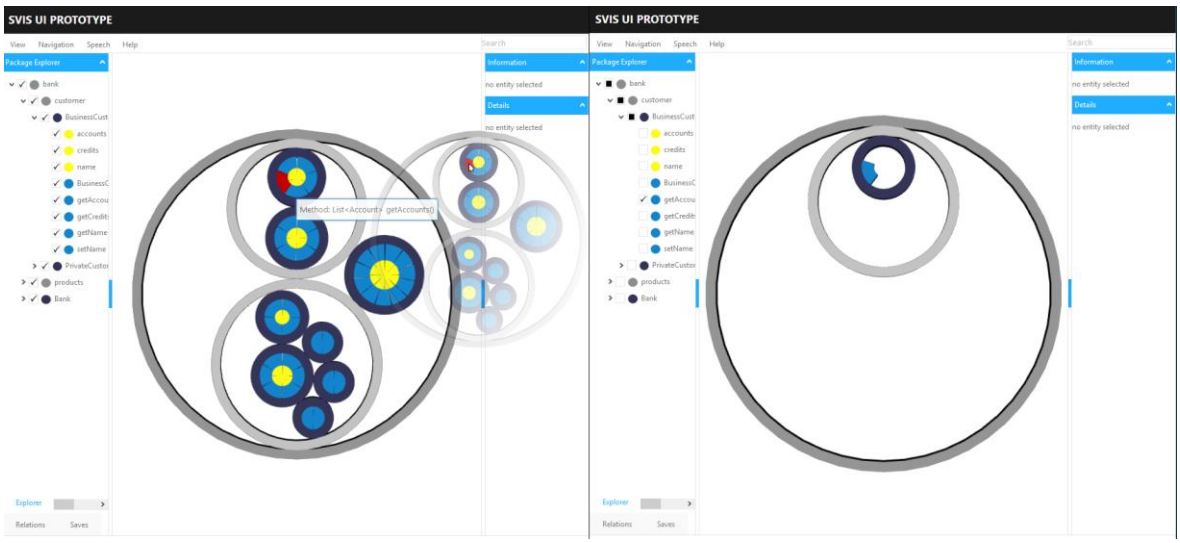

Abbildung 4-6: Glyphe wird gezogen und in anderem Fenster fallengelassen

5 Speichern und Laden von Teilsichten

In diesem Kapitel werden die Arbeitsschritte der Implementierung einer Möglichkeit zur Speicherung von erstellten Teilsichten vorgestellt. Zu Beginn werden die Vorüberlegungen zur Umsetzung beschrieben. Die Umsetzung der Funktionalitäten zum Speichern und Laden werden getrennt in eigenen Unterkapiteln dokumentiert.

5.1 Grundlagen zur Implementierung

Die Speicherung der Teilsichten soll in separaten Dateien erfolgen, die vom Browser aus erstellt werden sollen. Deren Inhalt steht erst während der Visualisierung fest, da die Anzahl der dargestellten Knoten vom Anwendungsfall abhängig ist. Es werden nur die IDs der Knoten gespeichert, da diese eine eindeutige Zuordnung zu den Elementen erlauben. Auf diese Weise werden keine redundanten Daten gespeichert und die Größe der Datei wird möglichst gering gehalten. Aufgrund dessen werden die Werte lediglich kommasepariert und aufeinanderfolgend in die Datei geschrieben. Zur Kennzeichnung dieser Eigenschaft wird als Dateityp das Comma-Separated-Value-Format (CSV-Format) gewählt. Der Name der Datei kann frei vergeben werden, da er nicht zur Auswertung herangezogen wird. Standardmäßig wird beim Herunterladen das aktuelle Datum und die aktuelle Uhrzeit als Dateiname vorgeschlagen.

Im Rahmen der Ausarbeitung des Bachelorarbeitsthemas wurde diskutiert, ob es sinnvoll ist, Metadaten wie den Namen der Visualisierung oder einen Zeitstempel zu den zu speichernden IDs hinzuzufügen. Dadurch wäre es möglich, Mechanismen zu implementieren, die das Auslesen einer Teilsicht unterbinden, sollte sie nicht zur geladenen Visualisierung gehören. Die Informationen in der Datei können ebenso veraltet sein, falls sich die Visualisierung seit der Speicherung verändert hat. In beiden Fällen bestünde keine Garantie der Kongruenz in Name, Anzahl und Struktur der Knoten zwischen der Visualisierung und der Menge der gespeicherten IDs. Diese Funktion wurde jedoch vorerst verworfen, da die benötigten Informationen nicht in den Quelldateien der Visualisierung vorhanden sind und somit erst der Output des Generators angepasst werden müsste. Als einzusetzende Skriptsprache wurde JavaScript gewählt, da es nativ von allen Browsern unterstützt wird und die Weboberfläche auch durch JavaScript aufgebaut ist. Ähnlich zur Implementierung von *Drag and Drop* wird auch hier eine eigene Skriptdatei mit eigener, modularer Controllerklasse erstellt, um die Manipulation des Kernquellcodes zu minimieren.

Die Schaltflächen zum Speichern und Laden wurden in den linken Teil der Oberfläche integriert, da sich dort bereits der Package Explorer befindet, mit dem die manuelle Erstellung einer Teilsicht möglich ist und daher der funktionale Zusammenhang entsprechend groß ist. Zur Abgrenzung von Package Explorer und Speicherdialog wurde letzterer in einen eigenen Tab ausgelagert. Abbildung 5-1 zeigt die Oberfläche mit geöffnetem Dialog. Er besteht aus einem Button zum Hochladen einer Datei und einem Button zur Speicherung der Teilsicht. Neben dem Hochladen-Button wird der Name der zuletzt hochgeladenen Datei angezeigt.

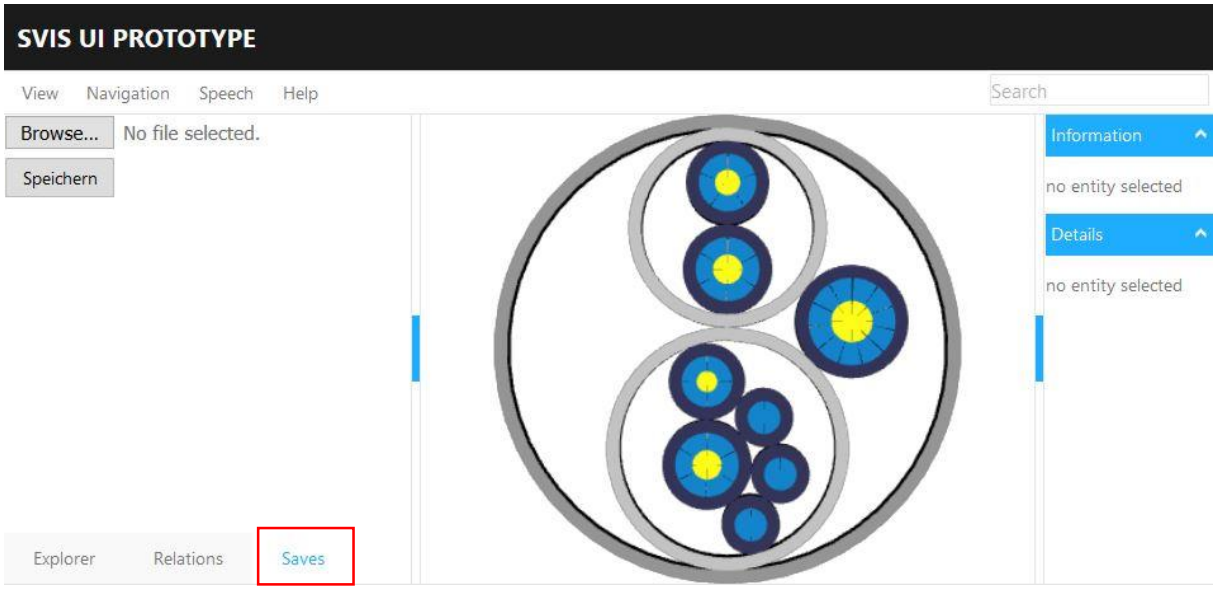

Abbildung 5-1: Dialog zum Speichern und Laden

5.2 Speichern einer Teilsicht

Die Grundidee ist es, mit Hilfe des Frameworks zTree alle nicht-selektierten Knoten auszuzählen. Die dafür zu nutzende Funktion generiert ein Array aller gefundenen Knoten, aus welchen iterativ die jeweiligen IDs extrahiert und kommasepariert an einen String angeheftet werden. Dieser String kann in Form einer Datei heruntergeladen werden.

Quellcode-Listing 5-1 zeigt den Quellcode der Speicherfunktionalität. Um die Interaktion mit dem durch zTree erstellten Baum zu ermöglichen, wird dieser zunächst durch die Funktion *getZTreeObj(„packageExplorerTree“)* in die Variable *treeObj* gespeichert. Der Parameter der Funktion ist der Name des Baumes, welcher der Visualisierung zugrunde liegt. Anschließend werden die nicht-selektierten Knoten durch den Parameter *false* der Funktion *getCheckedNodes(false)* ausgelesen (Zeile 3). Durch Übergabe von *true* wird hingegen nach allen selektierten Knoten gesucht. Weiterhin wird die Variable *csvContent* deklariert und bereits mit den Metadaten zu MIME-Type und Zeichensatz vorbelegt (Zeile 4). In Zeile 5 wird geprüft, ob das Array der herausgesuchten, nicht-selektierten Knoten nicht

leer ist. Ist dies der Fall, so hat der Anwender noch keine Teilsicht erstellt und die Ausgangsvisualisierung wird vollständig angezeigt. Um den Anwender darauf hinzuweisen, wird eine Fehlermeldung ausgegeben (Zeile 15).

```
 1 function saveView() {
 2   var treeObj = $.fn.zTree.getZTreeObj("packageExplorerTree");
 3   var nodes = treeObj.getCheckedNodes(false);
 4   var csvContent = "data:text/csv;charset=utf-8";
 5   if (nodes.length > 0) {
 6     /* Write the nodeIDs to csvContent in a comma-seperated manner */
 7     for (i = 0; i < nodes.length; i++) {
 8       csvContent += "," + nodes[i].id;
 9     }
10     /* Get current date and use it as the file's default name */
11     var d = new Date().toISOString().slice(0, 19);
12     this.setAttribute("href", csvContent);
13     this.setAttribute("download", "view-" + d + ".csv");
14   } else {
15     alert("Das Modell ist noch in der default-Ansicht.");
16   }
17 }
```

Quellcode-Listing 5-1: Speicherung einer Teilsicht

Ist das Array befüllt, so werden die IDs der Einträge durch die Nutzung einer *for*-Schleife iterativ ausgelesen und jeweils mit vorangestelltem Komma dem String *csvContent* angefügt (Zeile 7f). Zur Generierung des Dateinamens wird das aktuelle Datum in eine Zeichenfolge konvertiert und der Teil, welcher die Uhrzeit beinhaltet, abgeschnitten (Zeile 11). Das Setzen der Attribute *href* und *download* in Zeile 12 und 13 ermöglicht es, den Inhalt des Strings *csvContent* herunterzuladen.

5.3 Laden einer Teilsicht

Das Laden einer Teilsicht umfasst grob drei Schritte: Zunächst werden die IDs aus der Datei ausgelesen und in ein Array gespeichert. Danach werden alle Knoten aktiviert, wodurch die gesamte Visualisierung dargestellt wird. Anschließend werden nur die Knoten, deren IDs sich im Array befinden, ausgeblendet. Quellcode-Listing 5-2 zeigt die entsprechende Implementierung. Zunächst wird, analog zur *saveView*-Funktion, eine Variable *treeObj* deklariert, über die mit dem Baum interagiert wird. Um später alle Knoten einzublenden, werden diese als Array *allNodes* gespeichert (Zeile 3). Deren IDs werden iterativ extrahiert und in das Array *allNodeIDs* gespeichert. Weiterhin wird die Datei, welche zu diesem Zeitpunkt hochgeladen wurde, der Variable *file* zugewiesen. Der Button zum Hochladen der CSV-Datei nimmt lediglich eine Datei an. Dennoch muss angegeben werden, dass die Datei an der ersten Stelle der Liste steht.

```
 1 function loadView(e) {
 2   var treeObj = $.fn.zTree.getZTreeObj("packageExplorerTree");
 3   var allNodes = treeObj.transformToArray(treeObj.getNodes());
 4   var allNodeIDs = [];
 5
 6   for (var i = 0; i < allNodes.length; i++) {
 7     allNodeIDs.push(allNodes[i].id);
 8   }
 9
10   var file = e.target.files[0];
11   var reader = new FileReader();

      […]
```

Quellcode-Listing 5-2: Laden einer Teilsicht

Von dem von HTML nativ unterstützen *FileReader* wird anschließend eine Instanz mit dem Namen *reader* erstellt. Quellcode Listing 5-3 zeigt die Definition des Verhaltens, sobald der Reader eingesetzt wird.

```
      […]

 1   reader.onload = function(f) {
 2     var finalData = reader.result.split(',');
 3
 4  /* Necessary to display all glyphs in case that one partial view has
 5     already been loaded */
 6     treeObj.checkAllNodes(true);
 7     publishOnVisibilityChanged(allNodeIDs, true,
 8                                "packageExplorerTree");
 9
10     /* Uncheck all nodes in the file */
11     var currentNode = null;
12     for (var i = 0; i < finalData.length; i++) {
13       currentNode = treeObj.getNodeByParam("id", finalData[i]);
14       treeObj.checkNode(currentNode, false);
15     }
16     /* make the changes visible */
17     publishOnVisibilityChanged(finalData, false,
18                                "packageExplorerTree");
19   };
20   reader.readAsText(file);
21 }
```

Quellcode-Listing 5-3: Laden einer Teilsicht (Fortsetzung)

Zunächst werden die IDs der Datei durch die *split*-Funktion nach den Kommas aufgeteilt. Der Rückgabewert dieser Funktion ist ein Array, welches die Einträge der Datei enthält. Es wird unter dem Namen *finalData* abgespeichert. Anschließend erfolgt die Selektion aller Knoten durch *treeObj.checkAllNodes(true)*. Um diese Änderung auch in der Visualisierung darzustellen, wird die Funktion *publishOnVisibilityChanged* aufgerufen. Ihr wird als Parameter unter anderem die Liste der IDs aller Knoten (*allNodeIDs*) übergeben, welche im Vorfeld erstellt wurde.

Anschließend erfolgt die Deselektion der Knoten, deren IDs aus der Datei gelesen wurden. Zur Verbesserung der Lesbarkeit wird die Variable *currentNode* verwendet, welche den aktuell zu bearbeitenden Knoten darstellt. Der Knoten wird dabei zuerst anhand seines Parameters *id* ermittelt und anschließend durch *checkNode(currentNode, false)* deselektiert. Nachdem Durchlauf der Schleife erfolgt erneut der Aufruf der Funktion *publishOnVisibilityChanged*, welche die Knoten in der Visualisierung ausblendet. Die eben definierte Funktion wird aufgerufen, sobald der *FileReader* die Datei vollständig geladen hat. Dies geschieht durch den Aufruf der Funktion *reader.readAsText(file)*, welcher die hochgeladene Datei in Form der Variable *file* als Parameter übernimmt.

6 Fazit und Ausblick

Im Zuge dieser Arbeit wurden Möglichkeiten zur Erstellung einer dreidimensionalen Softwarevisualisierung ausgearbeitet und in die an der Universität Leipzig entwickelte Weboberfläche integriert. Wünschenswert wäre es, wenn die erreichten Ziele einen positiven Beitrag zur Interaktion mit Softwarevisualisierungen erwirken, indem sie die Arbeit mit dieser benutzerfreundlicher und leichter gestalten und somit mehr Menschen zur Nutzung dieser Möglichkeit der Dokumentation von Software zu bewegen.

Um die Änderungen festzuhalten, wurden zwei Hauptziele definiert und verfolgt. Eines setzte sich mit der Erstellung einer Softwarevisualisierung unter Zuhilfenahme einer bereits bestehenden auseinander. Das zweite Ziel beschrieb die Entwicklung einer Möglichkeit, eine erstelle Visualisierung zu speichern und anschließend wieder zu laden. Weiterhin konnten nebenbei neue Erkenntnisse zum Verhalten der verwendeten Software gewonnen werden, auf die in zukünftigen Projekten zurückgegriffen werden kann. Daran anknüpfend werden an dieser Stelle einige Ideen für zukünftige Arbeiten genannt, die auf den hier entwickelten Funktionalitäten aufbauen.

Eine Idee ist es, das Drag-and-Drop-System um Mehrfachselektion zu erweitern. Obwohl die Implementierung von *Drag and Drop* die Erstellung einer anderen Teilsicht durch gezielte Auswahl von Knoten und Glyphen beschleunigt hat, ist es noch mühsam, jede Glyphe einzeln von Fenster zu Fenster zu ziehen. Der gezogene Knoten wird zwar mitsamt aller hierarchisch über- und untergeordneten Knoten übertragen, jedoch muss es nicht immer der Fall sein, dass mehrere zu übertragende Knoten in derselben Hierarchielinie liegen. Alle auszuwählenden Knoten erst zu selektieren und dann mit einer einzigen Drag-and-Drop-Bewegung in das neue Fenster zu ziehen könnte sich als zeitsparender erweisen als für jeden einzelnen Knoten eine Bewegung auszuführen. Durch grafische Hervorhebung der bereits selektieren Knoten würde dem Anwender die Suche nach den nach ausstehenden Knoten vereinfacht. Derzeit muss die neue Visualisierung erst mit der bestehenden abgeglichen werden, um zu erkennen, welche Knoten noch zu übertragen sind.

Eine weitere Idee betrifft die Speicherung der IDs. So wurde in dieser Arbeit erörtert, ob neben den IDs auch Metadaten der Visualisierung gespeichert werden sollen, um eine Versionskontrolle zu ermöglichen. Es treten Fehler in der Visualisierung auf, wenn eine Datei zur falschen Visualisierung geladen wird. Ein vorgeschalteter Kontrollmechanismus kann die Informationen aus der Datei mit den Daten der Visualisierung abgleichen und gegebenenfalls Fehlermeldungen ausgeben, wenn sie nicht übereinstimmen. Um dies zu ermöglichen, müssen die entsprechenden Informationen in der Datei verfügbar sein. Ein

ähnliches Konzept kann auch für den Drag-and-Drop-Mechanismus verwendet werden. Auch an dieser Stelle findet keine Prüfung statt. Weiterhin ist es möglich, die Dateien nicht nur für das Speichern auf der Festplatte freizugeben, sondern mit Hilfe der File API von HTML5 eine eigene Dateistruktur in einem vom Browser bereitgestellten Speicherbereich zu erstellen und die CSV-Dateien darin zu speichern. Derzeit diese API nur für das Einlesen der Dateien verwendet. Sie besitzt jedoch weitaus mehr Potential.

Zuletzt folgt ein Vorschlag zur Trennung von Oberfläche und dargestellter Visualisierung. Es werden zurzeit unterschiedliche HTML-Dokumente verwendet, um verschiedene Visualisierungen darzustellen. Dabei wäre es praktischer, ein Dokument zu öffnen und die gewünschte Visualisierung im Anschluss zu laden. Die Quelldateien der verfügbaren Visualisierungen werden entweder extern auf der Festplatte gespeichert oder – soweit bereits vorhanden – ebenfalls im Dateisystem der File API.

Literaturverzeichnis

[Balzer und Deussen 2004] Balzer, M., Deussen, O., Hierarchy Based 3D Visualization of Large Software Structures, VIS 04: Proceedings of the conference on Visualization 04, IEEE Computer Society, Washington, DC, USA, 2004, ISBN 0-7803-8788-0, S. 598.4

[Balzert 2009] Balzert, H., Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering. 3. Auflage. Spektrum Akademischer Verlag, Heidelberg, 2009.

[Bassil und Keller 2001] Bassil, S., Keller, R.K., Software Visualization Tools: Survey and Analysis, Proceedings of the 9th International Workshop on Program Comprehension, 2001, S. 7–17.

[Bohnet et al. 2006] Bohnet, J., Döllner, J., Gierak, A., Lazic, N., Hagedorn, B., Schöbel, M., Konzepte der Softwarevisualisierung für komplexe, objektorientierte Softwaresysteme, Technischer Bericht, Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam, 2006, URL http://www.hpi.uni-potsdam.de/fileadmin/hpi/source/Technische_ Berichte/HPI_06_KonzepteDerSoftwareVisualisierung.pdf.

[Borgo et al. 2013] Borgo, R., Kehrer, J., Chung, D., Maguire, E., Laramee, R. S., Ward, M., and Chen, M., Glyph-based visualization: Foundations, design guidelines, techniques and applications. Eurographics, 2013.

[Buxton 1986] Buxton, W., There's More to Interaction than Meets the Eye: Some Issues in Manual Input. In Norman, D. A. and Draper, S. W. (Eds.), *User Centered System Design: New Perspectives on Human-Computer Interaction.* Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1986, 319-337.

[Card et al. 1999] Card, S.K., Mackinlay, J.D., Shneiderman, B., Readings in information visualization: using vision to think. Morgan Kaufmann Publishers Inc., San Francisco, 1999.

[Diehl 2007] Diehl, S., Software Visualization – Visualizing the Structure, Behaviour, and Evolution of Software. Springer-Verlag, Berlin, 2007.

[ECMA 2015] ECMA International (Hrsg.), ECMA 2015 Language Specification – ECMA-262 6th Edition, URL: http://www.ecma-international.org/ecma-262/6.0/ index.html#sec-scope, gelesen am 15.11.2015.

[Flanagan 2011] Flanagan, D., JavaScript: The Definitive Guide. 6. Auflage. O'Reilly Media, Sebastopol, 2011.

[Gračanin et al. 2011] Gračanin, D., Matković, K. & Eltoweissy, M., 2005. Software visualization. Innovations in Systems and Software Engineering, S. 221-230.

[Haber und McNabb 1990] Haber, R.B., McNabb, D.A., Visualization idioms: A conceptual model for scientific visualization systems, Visualization in Scien-tific Computing, IEEE Computer Society Press, 1990, S. 74–93.

[Inkpen et al. 1997] Inkpen, K., Booth, K.S., Klawe, M., Drag-and-Drop vs. Point-and-Click Mouse Interaction for Children. Submitted to CHI 97. Atlanta, GA: ACM Press, 1997.

[Lakoff und Johnson 1980] Lakoff, G., Johnson, M., Metaphors We Live By, University of Chicago Press, Chicago, 1980, ISBN 0-226-46801-1.

[MacKenzie et al. 1991] MacKenzie, S., Sellen, A., Buxton, W., A comparison of input devices in elemental pointing and dragging tasks, Published in: Proceeding CHI '91 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, New Orleans, 1991.

[Mackinlay 1986] Mackinlay, J., Automating the design of graphical presentations of relational information. ACM Trans. Graph., 5 (1986) 2, S. 110–141, ISSN 0730-0301.

[Müller 2009] Müller, R., 2009. Konzeption und prototypische Implementierung eines Generators zur Softwarevisualisierung in 3D. Diplomarbeit Universität Leipzig

[Müller et. al. 2011] R. Müller, P. Kovacs, J. Schilbach, U. Eisenecker (2011) Generative Software Visualizaion: Automatic Generation of User-Specific Visualisations, 45-49. *In Proceedings of the International Workshop on Digital Engineering.*

[Müller und Zeckzer 2015] Müller, R., Zeckzer, D., The Recursive Disk Metaphor: A Glyph-based Approach for Software Visualization. Proceedings, 6th International Conference on Information Visualization Theory and Applications (IVAPP 2015), 2015.

[o.V. 2015] o.V., Data buffer, Wikipedia, 2015 URL: https://en.wikipedia.org/wiki/Data_buffer

[Reiss 2005] Reiss, S.P., Software Visualization, Call for Papers, 2005, URL http://www.softvis.org/softvis05.

[dos Santos und Brodlie 2004] dos Santos, S. & Brodlie, K., 2004. Gaining understanding of multivariate and multidimensional data through visualization. Computers & Graphics, S. 311-325

[Schumann und Müller 1999] Schumann, H., Müller, W., Visualisierung - Grundlagen und allgemeine Methoden, 1. Aufl., Springer, 2000, ISBN 978-3540649441.

[Sellen et al. 1992] Sellen, A., Kurtenbach, G. & Buxton, W. (1992). The prevention of mode errors through sensory feedback. *Human Computer Interaction*, 7(2), 141-164.

[Longman 1998] Longman, A.W., W3C (Hrsg.), A history of HTML. URL: http://www.w3.org/People/Raggett/book4/ch02.html, gelesen am 15.11.2015

[W3C 2014] W3C (Hrsg.), HTML5: A vocabulary and associated APIs for HTML and XHTML - W3C Recommendation 28 October 2014. URL: http://www.w3.org/TR/2014/REC-html5-20141028/introduction.html#introduction%201.9, gelesen am 15.11.2015

Ehrenwörtliche Erklärung

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

Darüber hinaus versichere ich, dass die elektronische Version der Bachelorarbeit mit der gedruckten Version übereinstimmt.

_____

Leipzig, 24. November 2015