

# **Semantik-gestütztes Hilfesystem für ein komposites Informationsvisualisierungssystem**

Diplomarbeit  
Technische Universität Dresden  
Oktober 2013

Nikolaus Piccolotto

Betreuer: Dipl.-Medieninf. Martin Voigt  
Hochschullehrer: Prof. Dr.-Ing. Klaus Meißner

Fakultät Informatik  
Institut für Software- und Multimediatechnik  
Seniorprofessur für Multimediatechnik



# Aufgabenstellung

Diese Seite muss vor dem Binden der gedruckten Fassung der Arbeit durch die von Herrn Meißner und dem Studenten eigenhändig unterschriebene originale Aufgabenstellung ersetzt werden. Das zweite abzugebende gebundene Exemplar soll stattdessen eine Kopie dieser originalen Aufgabenstellung enthalten.

# Erklärung

Hiermit erkläre ich, Nikolaus Piccolotto, die vorliegende Diplomarbeit zum Thema

**Semantik-gestütztes Hilfesystem für ein komposites Informationsvisualisierungssystem**

selbstständig und ausschließlich unter Verwendung sowie korrekter Zitierung der im Literaturverzeichnis angegebenen Quellen und Hilfsmittel verfasst zu haben.

Dresden, 29. Oktober 2013

Unterschrift

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Problemstellung und Zielsetzung . . . . .	2
1.3 Aufbau der Arbeit . . . . .	3
<b>2 Stand der Forschung und Technik</b>	<b>4</b>
2.1 Szenario . . . . .	4
2.2 Anforderungsanalyse . . . . .	6
2.2.1 Funktionale Anforderungen . . . . .	6
2.2.2 Nichtfunktionale Anforderungen . . . . .	7
2.3 Grundlagen . . . . .	7
2.3.1 CRUISe, EDYRA & VizBoard . . . . .	7
2.3.2 Semantische Datensätze . . . . .	11
2.3.3 Informationsvisualisierung . . . . .	13
2.3.4 User Assistance . . . . .	20
2.4 Verwandte Arbeiten . . . . .	24
2.4.1 Webbasierte Mashup Plattformen . . . . .	24
2.4.2 Visualisierungstools . . . . .	25
2.4.3 User Assistance . . . . .	28
2.5 Zusammenfassung . . . . .	30
<b>3 Konzeption</b>	<b>33</b>
3.1 Einführung . . . . .	33
3.1.1 Rollenmodell . . . . .	33
3.1.2 User Interface . . . . .	34
3.1.3 Nutzerstudie . . . . .	35
3.2 Intro in die Funktionalität einer Komponente . . . . .	39
3.2.1 User Interface und Interaktion . . . . .	41
3.2.2 Backend . . . . .	42
3.3 Bedienung einer Komponente . . . . .	42
3.3.1 API und Komponentenbeschreibung . . . . .	43
3.3.2 Generierung der Assistance . . . . .	45
3.3.3 User Interface und Interaktion . . . . .	47
3.4 Feedback zu einer Komponente . . . . .	49
3.4.1 User Interface und Interaktion . . . . .	50
3.4.2 Backend . . . . .	50
3.5 Kommunikation zwischen Komponenten . . . . .	51
3.5.1 User Interface und Interaktion . . . . .	51
3.5.2 Backend . . . . .	53

3.6 Verlinkung von unbekannten Konzepten . . . . .	53
3.6.1 Markup und Backend . . . . .	53
3.6.2 User Interface und Interaktion . . . . .	54
3.7 Kommentare in Visualisierungen . . . . .	55
3.7.1 Features . . . . .	56
3.7.2 API und Komponentenbeschreibung . . . . .	58
3.7.3 User Interface und Interaktion . . . . .	60
3.7.4 Backend . . . . .	63
3.8 History . . . . .	65
3.8.1 Backend . . . . .	66
3.8.2 User Interface und Interaktion . . . . .	69
3.9 Meta-Hilfe . . . . .	72
3.9.1 Statische Meta-Hilfe . . . . .	72
3.9.2 Dynamische Meta-Hilfe . . . . .	72
3.9.3 User Interface und Interaktion . . . . .	75
3.10 Synthese . . . . .	76
3.11 Zusammenfassung . . . . .	79
<b>4 Implementierung</b>	<b>82</b>
4.1 Frontend . . . . .	82
4.1.1 Clientseitige Architektur . . . . .	83
4.1.2 API einer Komponente . . . . .	85
4.1.3 Kommentare . . . . .	87
4.1.4 Bedienung . . . . .	89
4.1.5 Kommunikation . . . . .	93
4.2 Backend . . . . .	93
4.2.1 Kommentare . . . . .	95
4.2.2 Bedienung und Kommunikation . . . . .	97
4.3 Zusammenfassung . . . . .	103
<b>5 Evaluation</b>	<b>106</b>
5.1 Nutzerstudie . . . . .	106
5.1.1 Mashupkomposition . . . . .	106
5.1.2 Aufgaben . . . . .	107
5.1.3 Teilnehmer . . . . .	107
5.1.4 Durchführung . . . . .	108
5.1.5 Ergebnisse . . . . .	109
5.2 Entwicklungsaufwand . . . . .	111
5.2.1 Individuelle Hilfefunktion . . . . .	111
5.2.2 Vorgestelltes Hilfesystem . . . . .	112
5.2.3 Auswertung . . . . .	113
5.3 Zusammenfassung . . . . .	113
<b>6 Zusammenfassung und Ausblick</b>	<b>115</b>
6.1 Zusammenfassung . . . . .	115
6.2 Ausblick . . . . .	117

<b>A Anhang</b>	<b>i</b>
A.1 Implementierung . . . . .	i
A.1.1 Java Klassendiagramm eines Kommentars . . . . .	i
A.1.2 JSON Format eines Kommentars . . . . .	i
A.1.3 RDF Format eines Kommentars . . . . .	iii

<b>Literaturverzeichnis</b>	<b>viii</b>
-----------------------------	-------------

# Abbildungsverzeichnis

2.1	Skizze der VizBoard Visualisierungskomponenten . . . . .	5
2.2	CRUISe Architektur aus [Pie12] . . . . .	8
2.3	VizBoard Architektur aus [VPM13] . . . . .	8
2.4	CRUISe Links aus [Pie12] . . . . .	9
2.5	Mashup Runtime Environment Architektur aus [Pie12] . . . . .	10
2.6	Nutzung der VISO in VizBoard . . . . .	11
2.7	Semantic Spectrum [Ber07] . . . . .	12
2.8	Parallel Coordinates . . . . .	13
2.9	Visual Analytics Process nach [Koh+11] . . . . .	14
2.10	Generic Model on Visualization nach [Wij05] . . . . .	14
2.11	Index Chart . . . . .	14
2.12	Karte . . . . .	15
2.13	Scatterplot Matrix . . . . .	15
2.14	Tag Cloud . . . . .	16
2.15	Baum . . . . .	16
2.16	Node-Link-Diagramm . . . . .	16
2.17	Symbol Map . . . . .	17
2.18	Recursive Pattern . . . . .	17
2.19	Nested Circles . . . . .	18
2.20	Faceted Browsing bei Amazon.de . . . . .	18
2.21	Interaktiver Zoom bei Google Maps: Im rechten, ausgezoomten Bild fehlt beispielsweise die Interstate 278 (grüne Markierung) . . . . .	19
2.22	Fisheye: Das Zentrum der Verzerrung befindet sich ungefähr beim hellblauen Knoten in der Mitte . . . . .	19
2.23	Linking & Brushing bei Crossfilter.js: Ausgewählt wurden Flüge mit mehr als 80 Minuten Verspätung, die betroffenen Bereiche in anderen Histogrammen wurden automatisch markiert. . . . .	19
2.24	Intelligente Assistance nach [RRD07] . . . . .	21
2.25	Kognitive Multimedia-Lerntheorie nach [MM02a] . . . . .	22
2.26	Data/Frame Modell nach [KMH06a] . . . . .	23
2.27	Weave User Interface . . . . .	26
3.1	Rollenmodell von VizBoard . . . . .	33
3.2	Grundlage der UI Mockups . . . . .	34
3.3	Verschiedene Möglichkeiten zur Platzierung der Hilfe UI . . . . .	35
3.4	Die ersten zwei Seiten des Paper Mockups . . . . .	36
3.5	Eine Teilnehmerin der Nutzerstudie . . . . .	37
3.6	UI Mockup: Intro . . . . .	41
3.7	UI Mockup: Bedienung (Schritt 1) . . . . .	48
3.8	UI Mockup: Bedienung (Schritt 2) . . . . .	48

3.9	UI Mockup: Comic . . . . .	49
3.10	UI Mockup: Feedback-Funktion . . . . .	50
3.11	UI Mockup: Kommunikationshilfe . . . . .	51
3.12	UI Mockup: Kommunikationshilfe mit eingeschränkten Optionen . . . . .	52
3.13	UI Mockup: Comic erklärt Kommunikation. Verändern des Karten-ausschnittes (Aktion: Panning, Operation: Drag) bewirkt ein Update im Balkendiagramm. . . . .	52
3.14	UI Mockup: Verlinkung von Legenden . . . . .	55
3.15	UI Mockup: Verlinkung von Interaktionselementen . . . . .	55
3.16	Ein Scatterplot mit eingezeichneten Annotationsmöglichkeiten: Leere Bereiche (A), nicht-leere Bereiche (B) und einzelne Datenpunkte (C). . . . .	57
3.17	Ein ortsbezogener Kommentar mit Pfeil und Text . . . . .	60
3.18	UI Mockup: Kommentar hinzufügen . . . . .	61
3.19	UI Mockup: Anzeige aller Kommentare . . . . .	62
3.20	UI Mockup: Alle Kommentare zum ausgewählten Datenpunkt . . . . .	62
3.21	UI Mockup: Ausgewählter Kommentar mit referenziertem Bereich und Datenpunkten . . . . .	63
3.22	Beispielhafte Komposition . . . . .	65
3.23	Modifizierter CRUISe Komponentenlebenszyklus . . . . .	66
3.24	Möglicher Ablauf der Events in der beispielhaften Komposition . . . . .	67
3.25	Ablaufdiagramm des History Algorithmus . . . . .	68
3.26	Erkennung von Nutzerinteraktionen . . . . .	69
3.27	Naiver Algorithmus um CSS zu in relative Einheiten zu transformieren	70
3.28	UI Mockup: Grafische History . . . . .	71
3.29	UI Mockup: Undo/Redo Buttons . . . . .	71
3.30	Architektur der dynamischen Hilfefunktion . . . . .	73
3.31	Datengrundlage für SML Algorithmen . . . . .	73
3.32	Klassifizierender Decision Tree Algorithmus . . . . .	74
3.33	Erkennung von False Positives und False Negatives . . . . .	75
3.34	UI Mockup: Statischer Button für Meta-Hilfe . . . . .	75
3.35	UI Mockup: Statische Meta-Hilfe . . . . .	76
3.36	UI Mockup: Dynamische Meta-Hilfe . . . . .	76
4.1	Architektur von CRUISe mit Assistance . . . . .	83
4.2	Die Methode <code>coordinatesToData</code> . . . . .	86
4.3	Visualisierungselement in einem Liniendiagramm . . . . .	86
4.4	User Interface um einen Kommentar zu verfassen . . . . .	87
4.5	User Interface um Kommentare zu lesen . . . . .	88
4.6	Genauigkeit der Bounding Box eines Elements . . . . .	88
4.7	Problem von flächenbasierten Annotationen bei unterschiedlicher Visualisierungsgröße: Das Zentrum ist in beiden Visualisierungen das Jahr 1990, in der unteren ist aber ein vollkommen anderer Bereich markiert. . . . .	89
4.8	Assistance zur Bedienung, Schritt 1 . . . . .	90

4.9	Die CSS Selektoren würden in der Rootcopy falsche Treffer liefern, da sowohl die Reihenfolge als auch die Elternknoten der DOM Elemente verändert sind. Deswegen werden Originale und Kopien über das Attribut <code>data-vizboard-copy</code> verknüpft und auffindbar gemacht.	91
4.10	Assistance zur Bedienung, Liniendiagramm . . . . .	92
4.11	Assistance zur Bedienung, SIMILE Timeline . . . . .	93
4.12	Erweiterter CapView . . . . .	94
4.13	Assistance zur Kommunikation . . . . .	94
4.14	Verschiedene Serialisierungen eines Kommentars . . . . .	95
4.15	Architektur des Kommentar Backends . . . . .	96
4.16	Generierung der Bilder für das Hilfesystem . . . . .	98
4.17	Die <code>activity-operations</code> Ontologie . . . . .	101
4.18	Beispielhafte Ordnerstruktur einer Komponente im Assistance Generator . . . . .	103
5.1	Objekt der Nutzerstudie . . . . .	107
5.2	Ort der Nutzerstudie . . . . .	108
5.3	Wie viele Teilnehmer die Hilfe aufriefen, aufgeschlüsselt nach Aufgabe	109
5.4	SUS Scores der Teilnehmer . . . . .	110
5.5	SIMILE Timeline . . . . .	111
5.6	Frames des animierten GIFs . . . . .	112
5.7	Benötigte Zeit zur Integration . . . . .	113
A.1	Java Klassendiagramm für Kommentare . . . . .	i

# Tabellenverzeichnis

3.1	Teilnehmer der Nutzerstudie . . . . .	36
3.2	Wertung der Kommunikationshilfe . . . . .	38
5.1	Teilnehmer der Nutzerstudie . . . . .	108

# 1 Einleitung

Im Laufe der letzten Jahre wurden immer mehr Daten veröffentlicht und im Internet auffindbar gemacht. Dazu gehören auch semantische Daten, die von Community-Projekten wie der DBpedia oder Regierungsorganisationen bereitgestellt werden. Die DBpedia ist eine maschinenlesbare Version der Wikipedia, mit ihr können mit Hilfe der Abfragesprache SPARQL automatisch Fragen beantwortet werden, deren manuelle Recherche zu aufwändig wäre. In data.gov.uk finden sich Informationen, die für alle Steuerzahler im Vereinigten Königreich relevant sein können, wie z. B. Organigramme und Gehälter von Regierungsorganisationen. Aus diesen frei verfügbaren Daten können viele Erkenntnisse gewonnen werden.

## 1.1 Motivation

In roher Form können Menschen diese Daten aber nur sehr schwer verarbeiten: Ein Computer kann mehrere Megabyte Text innerhalb von Sekunden lesen und im Arbeitsspeicher halten, ein Mensch benötigt viel Zeit und vergisst große Teile wieder. Hier setzen gemäß dem Sprichwort »Ein Bild sagt mehr als tausend Worte« Informationsvisualisierungen an. Sie bereiten Daten visuell auf, beispielsweise in Form von Diagrammen, und machen so die enthaltenen Informationen für Menschen einfacher zugänglich. So können in den Daten enthaltene Muster, Ausreisser, Zusammenhänge und Trends leichter erkannt werden.

Bei komplexen Daten, die mehrere Variablen unterschiedlichen Typs beinhalten, ist es sinnvoll, mehrere Informationsvisualisierungen zu kombinieren [Rob07; Pha+13; Rib+13]. Sie bieten verschiedene Sichten auf den Datensatz, beispielsweise eine Karte für die enthaltenen Länder und ein Balkendiagramm für die Entwicklung des BIP in den letzten Jahren. Auf diese Weise wird der Erkenntnisgewinn gefördert, da zum Beispiel sichtbar wird, dass Länder mit geringem BIP in ähnlichen Regionen liegen.

Einige Forschungsarbeiten [Cap+11; Pie12; Chu+12; VPM13] befassen sich damit, diese multiplen Sichten in Form eines Mashups von User Interface Bestandteilen umzusetzen. Dessen einzelne Komponenten können dabei von unterschiedlichen, unabhängigen Entwicklern erstellt und in unterschiedlichen Domänen bzw. Anwendungsfällen eingesetzt werden. Das ist sinnvoll, da auf diese Weise die Anzahl an vorhandenen Komponenten und damit der Nutzen für den Anwender maximiert werden kann. Diese Strategie bringt aber auch Nachteile für den Anwender mit sich. Da die Komponenten von unterschiedlichen Entwicklern gebaut werden, können sie unterschiedlich aussehen: Was bei einer Komponente eine Auswahl signalisiert, ist bei der anderen normale Hintergrundfarbe. Die Komponenten verfügen möglicherweise über unterschiedliche Interaktionsmetaphern: Beim Balkendiagramm von Entwickler A müssen die Buttons geklickt werden, beim Liniendiagramm von Entwickler B reicht ein `mouseover`. Hinzu kommt, dass verschiedene Entwickler unterschiedliche

Bezeichnungen für die dieselbe Interaktion wählen können, falls sie eine Hilfefunktion für ihre Komponenten anbieten. So verwendet Entwickler A den Begriff »MouseOver« und Entwickler B »Hover«. Das ist gegenläufig zur Bedienbarkeit des Mashups, da in verschiedenen Usability Guidelines ein konsistentes Vokabular gefordert wird. Aus diesen und anderen Gründen muss dem Anwender eine einheitliche, universelle Hilfestellung angeboten werden.

## 1.2 Problemstellung und Zielsetzung

Aus diesen Gründen stellen sich einem Anwender eines Mashups verschiedene Fragen. Zunächst sieht er mehrere unterschiedliche Komponenten und muss diese nach der Reihe inspizieren und einordnen, sodass er ein ungefähres Bild über ihren Nutzens hat. Außerdem ist unklar, welche Funktionen die Komponenten zur Verfügung stellen und wie diese aufgerufen werden: Kann die Tabelle sortiert werden und wenn ja, wie? Welche Operation auf welchen Elementen führt welche Sortierreihenfolge aus? Welche Alternativen gibt es? Möglicherweise tauschen Komponenten untereinander Nachrichten aus, um gegenseitig ihr User Interface zu aktualisieren. Dieser Vorgang ist für den Benutzer unsichtbar und seine Auswirkungen wahrscheinlich irritierend. Warum aktualisiert sich die Karte, wenn die Interaktion im Balkendiagramm stattfindet? Sind die Komponenten Informationsvisualisierungen, stellen sich zusätzliche Fragen: Was bedeutet ein bestimmter Begriff, wie ist eine Metrik definiert und warum fällt sie so plötzlich ab?

Ein einzelner Komponentenentwickler kann dem Benutzer jedoch nur sehr schwer Antworten auf diese Fragen geben. Hilfefunktionen erfordern ein durchdachtes Konzept, da sie ansonsten nicht effektiv sind. Das erhöht den Entwicklungsaufwand für Komponenten, welchen die Entwickler versuchen zu minimieren. Deswegen werden in den meisten Fällen keine Hilfefunktionen zu finden sein. Wenn ein Entwickler sich die Mühe macht und eine Hilfefunktion implementiert, dann ist diese vom selben Problem betroffen wie die Komponenten selbst: Die Darstellungen, Bezeichnungen und Interaktionen der Hilfe sind uneinheitlich und erfordern daher einen hohen mentalen Aufwand des Nutzers, um sie zu verstehen. Anwender sind bestrebt ihren mentalen Aufwand zu minimieren und werden die Hilfefunktionen daher selten nutzen. Auf der anderen Seite ist es für Komponentenentwickler sehr aufwändig, sich über einheitliche Richtlinien zur Gestaltung von Bildern, Interaktionen, Animationen und Bezeichnungen zu verständern und zu einigen. Selbst wenn sie es schaffen würden, gäbe es keine Garantie, dass sich alle Entwickler vollständig daran halten. Aus diesen Gründen müssen die Hilfefunktionen auf Plattformebene umgesetzt werden.

Ziel dieser Arbeit ist die Entwicklung eines Konzepts für ein semantik-gestütztes Hilfesystem für VizBoard, ein komposites Informationsvisualisierungssystem auf Basis der Mashup Plattform CRUISe. Das Konzept soll die eben beschriebenen Probleme lösen, den Aufwand für Komponentenentwickler so gering wie möglich halten und auf alle Komponenten anwendbar sein. Das Konzept soll prototypisch umgesetzt und in CRUISe integriert werden.

Daraus leiten sich die folgenden Teilziele der Arbeit ab: Da die angebotenen Hilfestellungen auch verständlich sein müssen, soll zunächst erarbeitet werden, wie diese präsentiert werden sollen und worauf geachtet werden muss. Auch ist es wichtig zu wissen, mit welchen verschiedenen Interaktionen und Präsentationen die Kompo-

nenten arbeiten könnten, weswegen ein Überblick über Informationsvisualisierungen geschaffen werden muss. Bestehende Hilfekonzepte in Mashups und Visualisierungstools müssen betrachtet und ihre Anwendbarkeit auf VizBoard untersucht werden. Diese Informationen sollen danach in das Konzept einfließen, welches prototypisch implementiert und anhand einer kleinen Studie evaluiert wird.

### 1.3 Aufbau der Arbeit

Zunächst wird die Aufgabenstellung anhand eines beispielhaften Szenarios verdeutlicht und daraus Anforderungen an das Konzept abgeleitet. Da für die Entwicklung des Konzepts Kenntnisse der betroffenen Themengebiete nötig sind, werden in den Grundlagen CRUISe und VizBoard, semantische Daten, Informationsvisualisierungen und User Assistance eingeführt. Danach werden in den verwandten Arbeiten die Vor- und Nachteile bestehender Lösungen betrachtet und Schlussfolgerungen für das Konzept gezogen, welches in Kapitel 3 vorgestellt wird. In Kapitel 4 wird danach auf die Umsetzung des Prototypen eingegangen und dieser evaluiert (Kapitel 5). Daraufhin wird die Arbeit mit einer Zusammenfassung und einem Ausblick geschlossen (Kapitel 6).

## 2 Stand der Forschung und Technik

Der folgende Abschnitt besteht aus fünf Teilen. Zuerst wird die Aufgabenstellung in einem Szenario verdeutlicht (Abschnitt 2.1). Daraus werden Anforderungen an das Hilfesystem abgeleitet (Abschnitt 2.2) und danach die Grundlagen von semantischen Daten, Informationsvisualisierungen und User Assistance bzw. Sensemaking erläutert (Abschnitt 2.3). Konzepte verwandter Arbeiten (Abschnitt 2.4) werden auf ihre Anwendbarkeit untersucht, bevor das Kapitel mit einer Zusammenfassung (Abschnitt 2.5) beendet wird.

### 2.1 Szenario

Wie in Kapitel 1 erläutert, ist das komposite InfoVis-System Teil der webbasierten Anwendung VizBoard. Sie leitet den Benutzer in mehreren Schritten von der Auswahl eines Datensatzes zur finalen, kompositen Informationsvisualisierung. Im vorletzten Schritt wählt dieser mit Hilfe eines Facettenbrowsers geeignete Visualisierungskomponenten aus, welche danach angezeigt werden. Um die Problemstellung noch einmal zu verdeutlichen, wird im folgenden ein mögliches Szenario beschrieben.

Anna möchte für ihr Biologiestudium mehr über die geografische Verteilung verschiedener Genvariationen herausfinden. Dazu sucht sie im Internet nach einem Datensatz, welchen sie auch findet. Leider ist er in einem für Anna unbekannten Format abgespeichert, nämlich OWL. Sie versucht die Datei mit Microsoft Excel und SPSS zu öffnen, weil sie keine anderen Programme zur Datenverarbeitung kennt, aber scheitert. Anna stellt fest, dass nur ihr Texteditor OWL öffnen und vernünftig darstellen kann. Als sie die Datei überfliegt, kann sie den Inhalt zwar erahnen, aber es ist einfach zu viel Text um ihn vollständig zu lesen. Davon abgesehen sind beispielsweise geografische Breite und Länge als Zahlenkombination keine anschauliche Repräsentation von Orten. Anna würde viel Zeit aufwenden müssen um sehr wenig des Inhalts zu verstehen. Aber selbst wenn sie die Datei in Excel hätte öffnen können, hätte sie nicht gewusst, mit welchen Diagrammen die vorhandenen Daten am Besten verstanden würden. Anna hört von einem Freund, dass VizBoard gut geeignet ist, um semantische Datensätze anzusehen und probiert es aus.

Anna hat ihren Datensatz auch bei VizBoard gefunden und ist neugierig: Sie wählt eine Karte, ein Balkendiagramm, eine Tabelle und eine Treemap aus (Abbildung 2.1); kurz darauf werden ihr die Visualisierungskomponenten angezeigt. Anna benutzt VizBoard zum ersten Mal und macht außer Facebook und YouTube auch sonst nicht viel im Internet, das heißt sie ist zunächst von den vier unterschiedlichen Fenstern etwas überfordert.

VizBoard bietet Anna aber sofort eine einführende Übersicht und erklärt kurz die Darstellungsform und den Inhalt jeder Komponente. Eine denkbare Erklärung der Treemap wäre zum Beispiel:

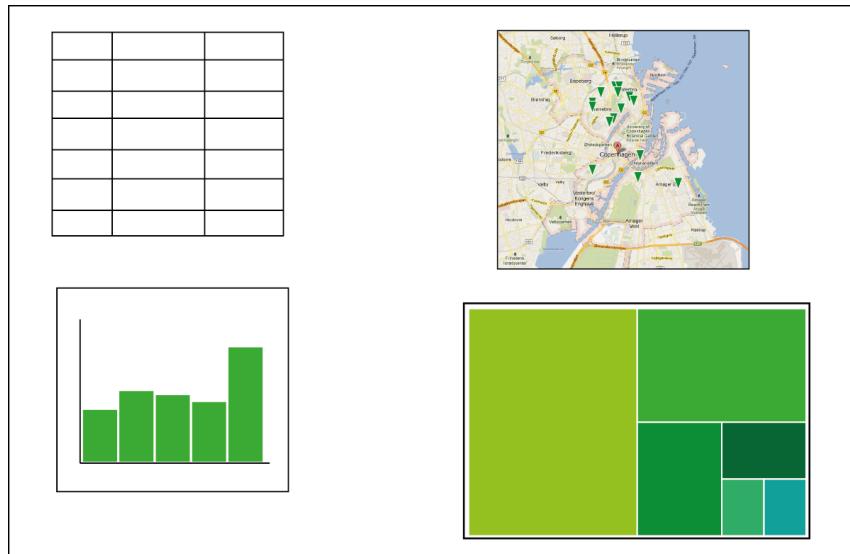


Abbildung 2.1: Skizze der VizBoard Visualisierungskomponenten

Eine Treemap ist eine hierarchische Visualisierung, um Größenverhältnisse anschaulich zu machen. In dieser werden die Anzahl von Genvariationen pro geografischer Region dargestellt.

Damit bekommt Anna einen Überblick über die verfügbaren Visualisierungen und weiß, welches Fenster welche Visualisierung enthält und für was diese gut sind. Nun möchte sie die Tabelle, in der die durch die Treemap visualisierten Daten stehen, nach der Spalte »Anzahl« sortieren. Anna sieht aber nicht, wie sie das machen soll, da in der Tabelle kein offensichtliches Kontrollelement wie z. B. ein Button vorhanden ist. Sie bemerkt ein Fragezeichen in der Titelleiste des Fensters und klickt darauf. Der verfügbare Viewport wird abgedunkelt und es erscheint ein neues Fenster, welches die verfügbaren Aktionen mit Hilfe von Text, Bildern und Animationen erklärt. Anna lernt, dass sie mit einem einfachen Linksklick auf den jeweiligen Kopf einer Tabellenspalte nach dieser sortieren kann und außerdem eine oder mehrere Zeilen auswählen kann. Sie sortiert die Tabelle wie gewollt und wählt die ersten drei Zeilen aus. Plötzlich verkleinert die Karte ihr Zoomlevel und Anna ist verwirrt: Sie hat nur mit der Tabelle interagiert und es bestand keine sichtbare Verbindung zwischen den beiden Fenstern. Allerdings wurde nach der Zeilenauswahl ein Pfeil von der Tabelle zur Karte gezeichnet, welcher mit einem Icon in Form eines Briefes versehen ist. Anna vermutet, dass doch irgendeine Verbindung zwischen den beiden Visualisierungen besteht und klickt auf den Brief. Ähnlich wie vorhin bei der Hilfe zur Tabelle wird der Viewport abgedunkelt und ein neues Fenster wird eingeblendet. Es erklärt die Kommunikation zwischen den Visualisierungen mit Hilfe von Animationen, Text und Bildern. Nun weiß Anna auch, wie die verschiedenen Fenster zusammenhängen und kann sich ihrer eigentlichen Aufgabe widmen.

In der Tabelle findet sie auch eine Spalte »SNP«. Anna weiß zwar, dass sie die Abkürzung schon einmal gesehen hat, kennt aber im Moment ihre Bedeutung nicht. Praktischerweise ist der Spaltenkopf mit der Wikipedia verlinkt und sie wird sofort auf die entsprechende Seite weitergeleitet. Anna erinnert sich, dass »SNP« »Single-nucleotide polymorphism« bedeutet und sie bekommt auch gleich zusätzliche Infor-

mationen zu diesem Thema. Sie widmet sich weiter der Tabelle und stellt fest, dass die Ortsbezeichnung »Kopenhagen« nicht mit der Markierung in der Karte übereinstimmt. Außerdem ist sie erstaunt, wie hoch die Verbreitung eines bestimmten SNPs dort ist und würde gerne die Ursache dafür wissen. In der Hilfe zur Tabelle wurde sie auch über die Möglichkeit, Kommentare an den Daten vorzunehmen, aufgeklärt. Anna kommentiert sowohl die falschen Geokoordinaten als auch ihre Frage über die Verbreitung des SNPs, sodass sie später über Antworten informiert wird. Nun kann Anna sich mit der vierten Visualisierung, dem Balkendiagramm, beschäftigen. Allerdings reagiert es auf keine Mausklicks und macht auch sonst nicht den Eindruck, die Daten akkurat darzustellen. Anna meldet die kaputte Visualisierung über die eingebaute Feedback-Funktion und schließt das Fenster, um sich den anderen drei Visualisierungskomponenten zuzuwenden.

## 2.2 Anforderungsanalyse

Aus dem Szenario (Kapitel 2.1) lassen sich nun verschiedene Anforderungen an ein Hilfesystem für komposite Informationsvisualisierungssysteme ableiten. Diese sind in funktionale und nicht-funktionale Anforderungen unterteilt (Abschnitte 2.2.1 und 2.2.2).

### 2.2.1 Funktionale Anforderungen

Funktionale Anforderungen geben Funktionen an, die das Hilfesystem unterstützen bzw. dem Anwender zur Verfügung stellen muss. Aus dem Szenario ergeben sich folgende:

- **Intro:** Das Hilfesystem soll einen kurzen Überblick über das InfoVis-System geben und Darstellungsform sowie Inhalt jeder Komponente kurz erläutern.
- **Bedienung:** Das Hilfesystem soll erklären können, wie eine Komponente bedient wird. Diese Informationen umfassen beispielsweise welche Operationen welche Aktionen (eventuell auf welchen Daten) ausführen.
- **Feedback:** Fehler in Komponenten sollen über ein Feedback-System gemeldet werden können.
- **Verlinkung:** Das Hilfesystem soll nicht-triviale Begriffe mit einer Wissensbasis, verlinken, sodass nicht nur auf die Begriffsbedeutung hingewiesen werden, sondern dem Benutzer auch zusätzliche Informationen zur Verfügung gestellt werden können.
- **Kommunikation:** Das Hilfesystem soll erklären können, wie gegebene Komponenten miteinander kommunizieren.
- **Kommentare:** Der Benutzer soll die Möglichkeit haben Daten zu kommentieren und Bereiche der Visualisierung zu markieren und mit ebenfalls mit einem Kommentar zu versehen, sodass auch auf fehlende Daten hingewiesen werden kann.

## 2.2.2 Nichtfunktionale Anforderungen

Nichtfunktionale Anforderungen beschreiben Grenzen, innerhalb derer die Funktionalität des Systems arbeiten muss. Für das Hilfesystem erscheinen folgende sinnvoll:

- **Korrektheit:** Eine gegebene Hilfestellung darf keine Fehlinformationen enthalten, weil sie sonst mehr verwirrt als hilft.
- **Vollständigkeit:** Eine gegebene Hilfestellung muss alle Informationen enthalten, die der Nutzer benötigt um danach seine gewünschte Aufgabe ausführen zu können.
- **Verständlichkeit:** Hilfestellungen müssen in einer Form präsentiert werden, die der Benutzer schnell und mit geringem mentalen Aufwand verarbeiten kann.
- **Einheitlichkeit:** Das Look & Feel von Teilen des Hilfesystems (z.B. Kommentare) muss komponentenübergreifend einheitlich sein, damit der Benutzer einmal gelerntes wiederverwenden kann.
- **Minimalität:** Der Komponentenentwickler soll seine Komponente mit möglichst wenig Aufwand zum Hilfesystem kompatibel machen können, ansonsten werden nur sehr wenige Komponenten – und damit der Benutzer – davon profitieren.
- **Universalität:** Das Hilfesystem soll für alle Komponenten und Visualisierungen in gleicher Qualität funktionieren.
- **Wiederverwendbarkeit:** Die Kommentare sollen möglichst in allen Visualisierungen wiederverwendet werden, damit viele Benutzer von den Erkenntnissen anderer profitieren können.
- **Unaufdringlichkeit:** Das Hilfesystem soll den Benutzer nicht von seinen Aufgaben ablenken und nur auf Anfrage zum Einsatz kommen oder es soll selbstständig erkennen, wenn der Benutzer Hilfe benötigt.

## 2.3 Grundlagen

Dieser Abschnitt beschäftigt sich mit den zur Konzeption nötigen Grundlagen. Diese beinhalten die CRUISe Plattform (Abschnitt 2.3.1), semantische Datensätze (Abschnitt 2.3.2), Informationsvisualisierung (Abschnitt 2.3.3) und User Assistance (Abschnitt 2.3.4).

### 2.3.1 CRUISe, EDYRA & VizBoard

CRUISe [Pie+09; Pie12] ist ein Framework zur Konstruktion von webbasierten, kompositen User Interfaces (auch Mashups). Diese bestehen aus mehreren voneinander unabhängigen, wiederverwendbaren User Interface Services (z. B. eine Karte, eine Tabelle und ein Kalender) welche zur Laufzeit hinzugefügt, konfiguriert und ausgetauscht werden können. Abbildung 2.2 zeigt die Architektur von CRUISe. Zuerst

werden Komponenten von Entwicklern erstellt und modelliert (1). Danach interpretiert CRUISe das Kompositionsmodell (2), macht es gegen die Kontextanforderungen (3), bildet ein Ranking (4) und integriert schließlich die am besten geeignete Komponente ins User Interface (5).



Abbildung 2.2: CRUISe Architektur aus [Pie12]

CRUISe geht davon aus, dass professionelle Softwareentwickler Komponenten zu kompositen Sichten modellieren, welche dann von Endnutzern nur noch aufgerufen werden. EDYRA [Rüm+11] greift auf das CRUISe Framework zurück, um es Endnutzern selbst zu ermöglichen, diese Aufgabe auszuführen. Dazu schlägt EDYRA zur Laufzeit Komponenten vor, welche sofort integriert oder gelöscht werden können.

VizBoard [VPM13] benutzt das CRUISe Framework, um Endnutzern die Möglichkeit zu geben, semantische Datensätze mit Hilfe verschiedener InfoVis verstehen zu können. Abbildung 2.3 zeigt die Architektur von VizBoard. Die Daten werden zuerst aus verschiedenen Quellen in eine semantische Repräsentation konvertiert und im Data Repository (DaRe, 2) gespeichert. Das Composition Repository (CoRe, 4) verwaltet die verschiedenen Visualisierungskomponenten und ist für Matching und Ranking von Komponenten zuständig, bevor sie in der Runtime (5) integriert werden. Sowohl das DaRe als auch das CoRe greifen auf die Visualization Ontology (ViSO, 1) zurück, die Wissen über verschiedene Aspekte von Visualisierungen, wie z. B. Struktur der visualisierten Daten, Mapping der Daten auf visuelle Attribute oder Interaktionsmöglichkeiten, enthält [PV13].

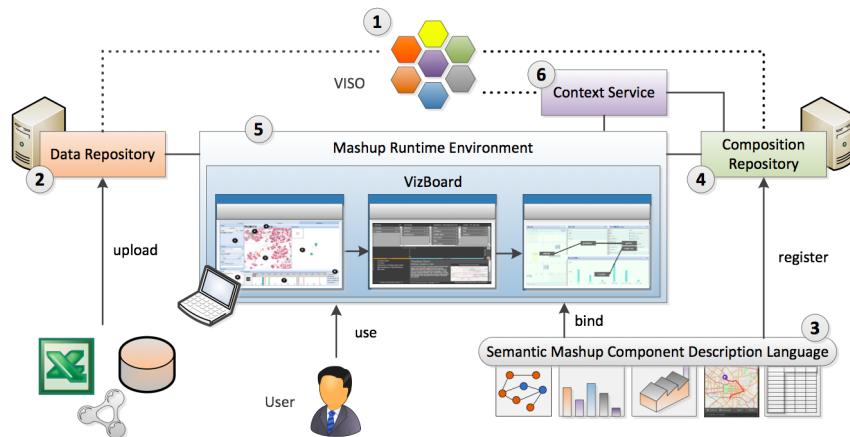


Abbildung 2.3: VizBoard Architektur aus [VPM13]

## Data Repository

Im DaRe werden Daten aus verschiedenen Quellen in einer semantischen Repräsentation gespeichert. Dabei durchläuft das DaRe verschiedene Phasen: Zuerst wird der Datensatz *analysiert* und Metainformationen gesammelt, beispielsweise werden

Subklassen gezählt und wichtige Konzepte identifiziert. Diese werden an den Datensatz *annotiert*. Zur Laufzeit müssen diese Erkenntnisse wieder aus dem Datensatz *extrahiert* werden. Das DaRe stellt eine REST API zur Verfügung, über die ein Datensatz abgerufen werden kann [Pic12].

## Komponentenbeschreibung

Komponenten in CRUISe werden generisch und einheitlich durch Propertys, Events und Operationen sowie Metainformationen beschrieben. Propertys geben Auskunft über den Zustand einer Komponente, also beispielsweise Breite und Höhe oder die Sortierreihenfolge der Elemente. Events weisen andere Komponenten auf interne Zustandsänderungen hin, zum Beispiel wenn sich die Sortierreihenfolge geändert hat. Operationen sind Methoden einer Komponente, die durch Events ausgelöst werden, z. B. `reassignNumbers(order)`, um die Nummerierung der Pins in einer Karte anzupassen. Zu den Metainformationen gehören u. a. der Name einer Komponente oder der Preis.

## Kommunikationsmodell

Die Kommunikation von Komponenten untereinander ist ereignisgesteuert, d. h. eine Komponente veröffentlicht ein Event, dessen Nachricht mit Hilfe des Publish/Subscribe Paradigma an alle Subscriber in diesem Kanal (Link) übertragen wird. Subscriber reagieren auf das Event, indem sie bestimmte Operationen ausführen. Die Übertragung der Nachrichten wird durch sogenannte Links umgesetzt, welche  $n$  Events mit  $m$  Operationen verbinden. Spezielle Links sind Backlinks, die eine Zweiwegekommunikation zwischen Komponenten ermöglichen und PropertyLinks, die Propertys synchron halten. Ein Überblick über die verschiedenen Links ist in Abbildung 2.4 zu sehen.

	<b>Link</b>	<b>BackLink</b>	<b>PropertyLink</b>
<b>Kardinalität</b>	m:n	1:1	m:n
<b>Programmfluss</b>	Daten- und Kontrollfluss	Daten- und Kontrollfluss	Datenfluss
<b>Datenfluss-Richtung</b>	Push	Push und Pull	Impliziter Push
<b>Intention</b>	Publikation ( <i>Fire and Forget</i> )	Publikation mit Rückmeldung ( <i>Request-Response</i> )	Synchronisation
<b>Zeitliche Verarbeitung</b>	Synchron	Asynchron	Synchron

Abbildung 2.4: CRUISe Links aus [Pie12]

## Laufzeitumgebung

Die Laufzeitumgebung (Mashup Runtime Environment, MRE) ist für die Ausführung und Verwaltung der kompositen Anwendung verantwortlich und besteht aus mehreren Modulen (Abbildung 2.5).

Der **Application Manager** initialisiert alle anderen Module und ist für die globale Fehlerbehandlung zuständig. Für den Lebenszyklus der einzelnen Komponenten ist der **Component Manager** zuständig. Die Integration übernimmt allerdings der

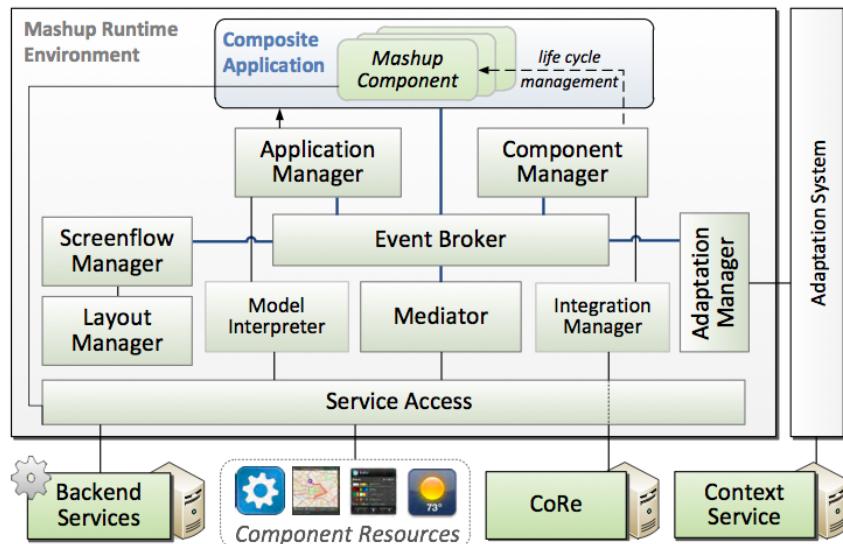


Abbildung 2.5: Mashup Runtime Environment Architektur aus [Pie12]

**Integration Manager.** Im Mashup Modell definierte Sichten und Übergänge zwischen ihnen werden vom **Screenflow Manager** interpretiert. Gerendert werden die Komponenten aber vom **Layout Manager**. Nachrichten zwischen Komponenten der Anwendung und Modulen der MRE werden durch den **Event Broker** übermittelt. Sind die Parameter von Event und Operation syntaktisch nicht äquivalent, aber semantisch aufeinander abbildbar (z. B. zwei gleiche Konzepte aus verschiedenen Namespaces wie dbpedia:city und geonames:city), übernimmt dies der **Mediator**. Die dynamische Anpassung der Anwendung (z. B. von Komponenten, Layout und Kommunikationsmodell) wird gegebenenfalls durch den **Adaptation Manager** durchgeführt. Der **Context Service** [Pie+08] speichert Informationen über den Nutzungskontext, beispielsweise den Aufenthaltsort des Nutzers. Letztlich erlaubt das **Service Access** Modul Zugriff auf Webdienste und Ressourcen im Backend.

## VISO

In der Visualization Ontology (VISO) ist Visualisierungswissen gespeichert. Sie kann dabei helfen, zur Laufzeit die Domäne dargestellter Daten zu identifizieren oder zur User Assistance (Abschnitt 2.3.4) notwendiges Wissen zur Komponentenbeschreibung hinzuzufügen. Unter anderem enthält sie folgende Konzepte und deren Verbindungen untereinander:

- Visualisierte Daten (*viso:data*)
  - Scale of Measurement (nominal, ordinal, quantitativ, unstrukturiert)
  - Struktur der Daten (tabellarisch, Tripel, verlinkt)
  - Art der Variable (abhängig, unabhängig, Dimension etc.)
  - Domäne
- Aktivitäten (*viso:activity*)

- Nutzeraktivitäten (Operationen, Aktionen, Aufgaben)
- Visualisierungspipeline (Editieren, Visual Mapping, Datentransformation etc.)
- Grafikvokabular (`viso:graphic`)
  - Visuelle Attribute (Größe, Farbe, Textur etc.)
  - Koordinaten (kartesisch, Zylinder, Kugel etc.)
  - Art der grafischen Repräsentation (Karten, Chart mit zwei Achsen, animierte InfoVis etc.)
  - Beziehungen zwischen Objekten (Clustering, Labeling, verlinkt etc.)
- System (`viso:system`)
  - Hardware
  - Software
  - Bildschirmauflösung
  - Rundes, eckiges oder unstrukturiertes Display

Dieses Wissen wird in VizBoard auf verschiedene Art und Weise genutzt (Abbildung 2.6). Mit Hilfe der VISO kann Wissen von Visualisierungsexperten formalisiert und im Rankingalgorithmus berücksichtigt werden (2). Gleichermassen wird sie zur Annotation der Daten im Data Repository (Abschnitt 2.3.1) benutzt (3). Visualisierungskomponenten (4) und Nutzer- bzw. Systemkontext (5) werden durch VISO-Konzepte beschrieben.

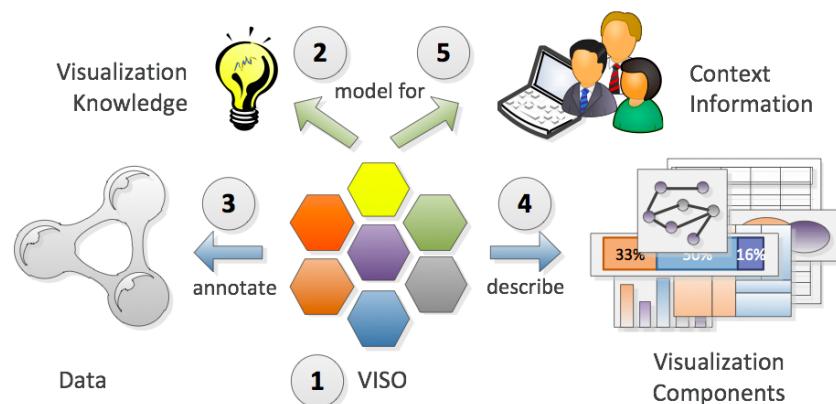


Abbildung 2.6: Nutzung der VISO in VizBoard

### 2.3.2 Semantische Datensätze

Eine Ontologie ist »an explicit specification of a conceptualization« [Gru95] und wird benutzt um domänen spezifisches Wissen abzubilden [CJB99]. Sie besteht aus mehreren Elementen:

- Eine Klasse repräsentiert ein Konzept, eine Entität, ein Ding, beispielsweise ein *Smartphone*.

- Eine Instanz ist ein konkretes Objekt einer Klasse, zum Beispiel das *iPhone mit der Seriennummer XYZ-ABC*.
- Datatype Propertys beschreiben eine Instanz näher, zum Beispiel die *Seriennummer* oder *Bildschirmgröße* des iPhones.
- Object Propertys beschreiben Beziehungen zwischen Klassen und deren Instanzen, beispielsweise eine Person *besitzt* ein Smartphone.
- Außerdem existieren noch Axiome, Regeln, Funktionen und Einschränkungen, welche die Logik einer Ontologie beschreiben.

Um eine Ontologie maschinenlesbar darzustellen, hat das World Wide Web Consortium (W3C) verschiedene Beschreibungssprachen eingeführt. Die bekanntesten sind Resource Description Framework (RDF), RDF Schema (RDFS) und Web Ontology Language (OWL). Mit diesen Sprachen lässt sich unterschiedlich viel Semantik u. a. in Form von Ontologien, Thesauri oder Vokabularen ausdrücken; die Komplexität der Dokumente und damit der Aufwand, sie zu erstellen, verhalten sich aber direkt proportional (siehe Abbildung 2.7).

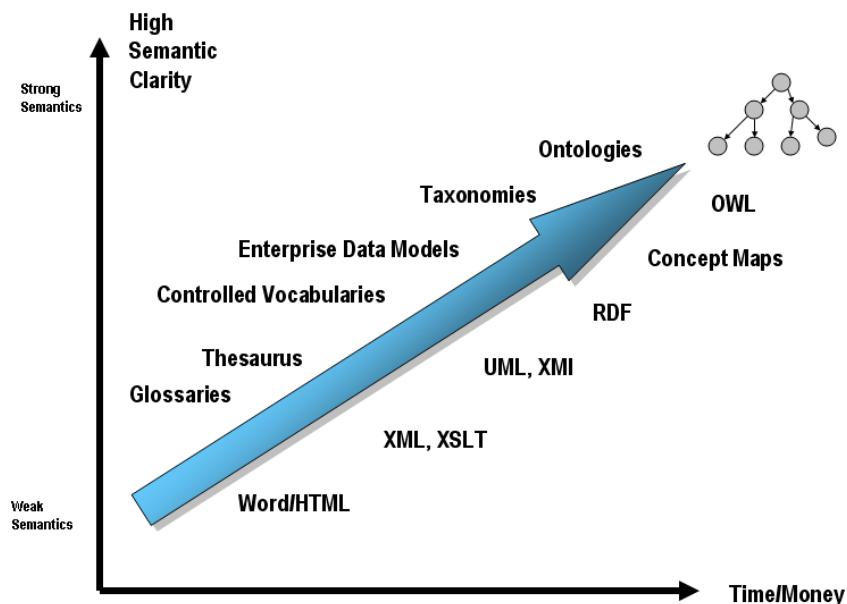


Abbildung 2.7: Semantic Spectrum [Ber07]

RDF ist von den genannten die Sprache mit den wenigsten Features, sie kann nur Tripel der Form (Subjekt, Prädikat, Objekt) darstellen. Die Elemente der Tripel sind Ressourcen (durch URIs gekennzeichnet), welche Objekte beschreiben. So drückt der Tripel (<http://alice.de>, <http://foaf.de/knows>, <http://bob.de>) aus, dass das Element »Alice« mit »Bob« über die Relation »knows« verbunden ist. In RDF existieren noch keine Vererbung, Klassen, Propertys oder Logik.

RDFS ist eine Erweiterung von RDF, nämlich um Klassen und Vererbung, Datatype und Object Propertys. Allerdings ist RDFS damit noch immer nicht so mächtig wie OWL.

OWL erweitert wiederum RDFS um einige Konzepte. So können beispielsweise Axiome definiert werden, Property können als transitiv oder symmetrisch deklariert werden, es existieren Mengenoperationen und Kardinalitäten und Wertebereiche können eingeschränkt werden. OWL existiert in zwei Versionen, da in den meisten Fällen nicht der volle Funktionsumfang benötigt wird: OWL Lite ist für Anwendungen gedacht, die kaum mehr als eine Klassenhierarchie und Attribute benötigen. OWL-DL ist auf die Logik in der Ontologie fokussiert und wird vor allem für Reasoner eingesetzt, um selbstständig Schlüsse innerhalb des vorgegebenen formalen Systems ziehen zu können. OWL 2 wurde 2009 eingeführt, dieser Standard erweitert OWL zum Beispiel um asymmetrische, reflexive und disjunkte Property.

### 2.3.3 Informationsvisualisierung

Card et al. [CMS99] definieren den Begriff »Informationsvisualisierung« wie folgt:

The use of computer-supported, interactive, visual representations of abstract data to amplify cognition.

Beispiele dafür sind Balkendiagramme, Treemaps [Shn92] und Parallel Coordinates [ID91] (Abbildung 2.8). Mangels Interaktivität sind Infografiken [Smi12] von den eben definierten Informationsvisualisierungen ausgeschlossen.

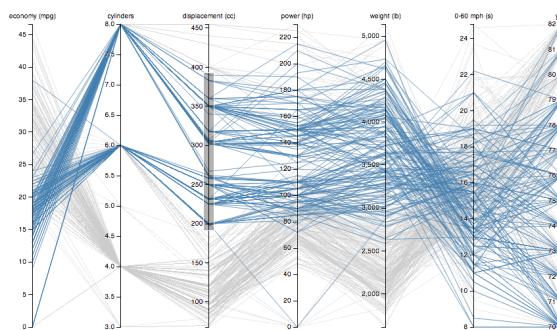


Abbildung 2.8: Parallel Coordinates

Informationsvisualisierungen können besonders bei der Exploration großer Datenmengen hilfreich sein [Koh+11]. Die Wissensaneignung erfolgt dabei iterativ (Abbildung 2.9). Zuerst werden Daten auf eine Visualisierung gemappt, deren Parameter vom Benutzer geändert werden können. Daraus lernt dieser etwas über die Daten und kann danach (»Feedback loop«) von vorne anfangen und eine andere Visualisierung wählen oder die Daten transformieren.

Van Wijk [Wij05] stellt ein ähnliches Modell für den Prozess der Wissensaufnahme über Visualisierungen vor (Abbildung 2.10). Am Anfang stehen die Daten  $D$ , welche anhand einer Visualisierungsspezifikation  $S$  in eine Visualisierung  $V$  transformiert werden. Diese wird vom Benutzer aufgenommen ( $P$ ) und in Wissen ( $K$ ) umgesetzt. Danach startet die Exploration der Daten  $E$  indem die Spezifikation geändert und neues Wissen aufgenommen wird (entspricht der »Feedback loop« aus [Koh+11]).

Im Folgenden wird ein Überblick über verschiedene Informationsvisualisierungen gegeben. Dieser orientiert sich an Keim [Kei02], welcher Informationsvisualisierungen nach dargestellten Daten, Visualisierungs- und Interaktionstechnik klassifizierte. Die Abbildungen stammen, sofern nicht anders angegeben, aus [HBO10].

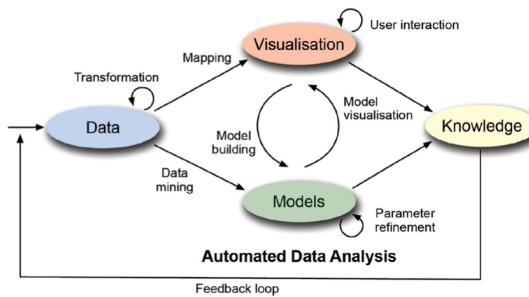


Abbildung 2.9: Visual Analytics Process nach [Koh+11]

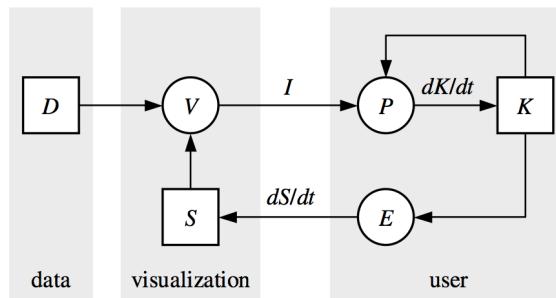


Abbildung 2.10: Generic Model on Visualization nach [Wij05]

## Dargestellte Daten

**Eindimensionale Daten** sind beispielsweise Zeitreihen, also Folgen von Daten (z. B. 1992, 1993, 1995...). Nach Keim können diese aber mit anderen Datenobjekten assoziiert sein. Beispiele für InfoVis eindimensionaler Daten wären demnach ein Index Chart (Abbildung 2.11) oder eine einfache Timeline.

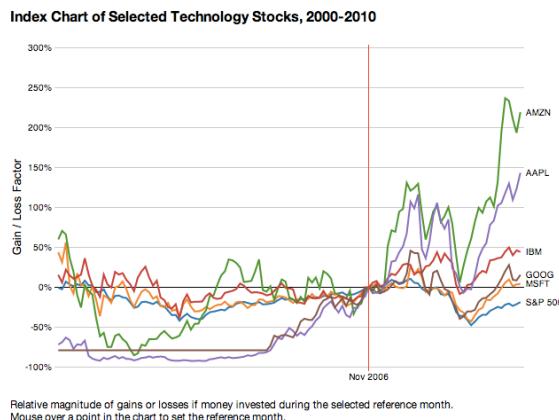


Abbildung 2.11: Index Chart

**Zweidimensionale Daten** haben zwei unterschiedliche Dimensionen, wie zum Beispiel eine Geokoordinate (geografische Länge und Breite). Beispiele für InfoVis dieser Daten sind eben Karten (Abbildung 2.12) oder häufig verwendete zweidimensionale Visualisierungen wie z. B. ein Balkendiagramm.

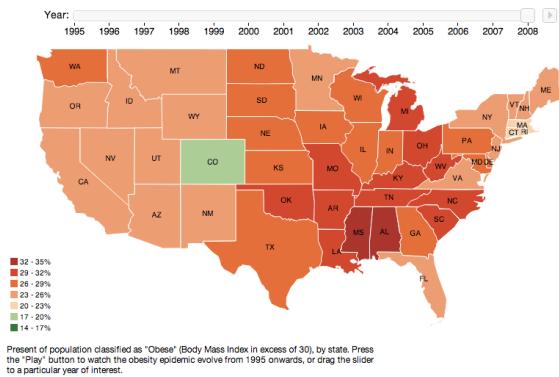


Abbildung 2.12: Karte

**Multidimensionale Daten** haben demnach mehr als zwei unterschiedliche Dimensionen, typischerweise komplexe Objekte wie Autos (Hubraum, Maximalgeschwindigkeit, Leistung, Benzinverbrauch...) oder Digitalkameras (Megapixel, Sensorgröße, maximale Lichtempfindlichkeit, Gewicht...). Um diese Daten darzustellen, werden oft Parallel Coordinates (Abbildung 2.8) oder Scatterplot Matrizen (Abbildung 2.13) verwendet.

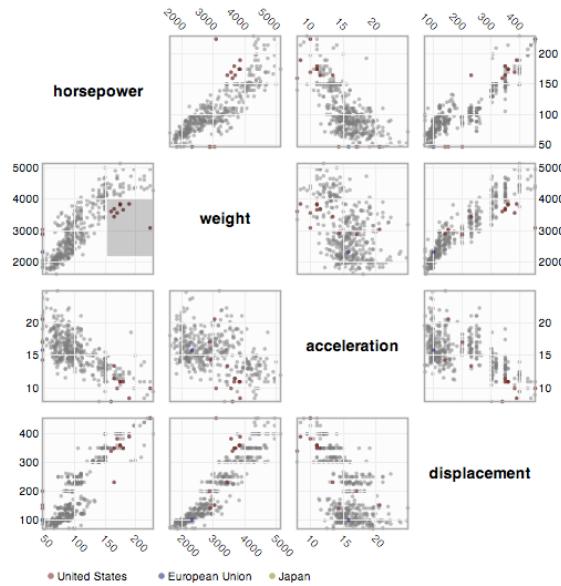


Abbildung 2.13: Scatterplot Matrix

**Text** kann erst nach einer Vorverarbeitungsphase mit Zahlen beschrieben werden (z. B. Wörter zählen), ansonsten schlagen herkömmliche Visualisierungsansätze fehl. Ein im Web verbreitetes Beispiel ist die Tag Cloud (Abbildung 2.14<sup>1</sup>). Je häufiger ein Begriff im Textkorpus vorkommt, desto größer wird er dargestellt.

**Hierarchien und Netzwerke** beschreiben Relationen und Verbindungen zwischen Objekten. Ein Beispiel für InfoVis von Hierarchien ist der klassische Baum (Abbildung 2.15), für Netzwerke ein Node-Link-Diagramm (Abbildung 2.16).

<sup>1</sup>[http://4.bp.blogspot.com/-Wv1cpJ9QqQs/TpbqvKhX3I/AAAAAAAADGc/3PczLY2P0xs/s1600/uni\\_tag\\_wordle.png](http://4.bp.blogspot.com/-Wv1cpJ9QqQs/TpbqvKhX3I/AAAAAAAADGc/3PczLY2P0xs/s1600/uni_tag_wordle.png)



Abbildung 2.14: Tag Cloud

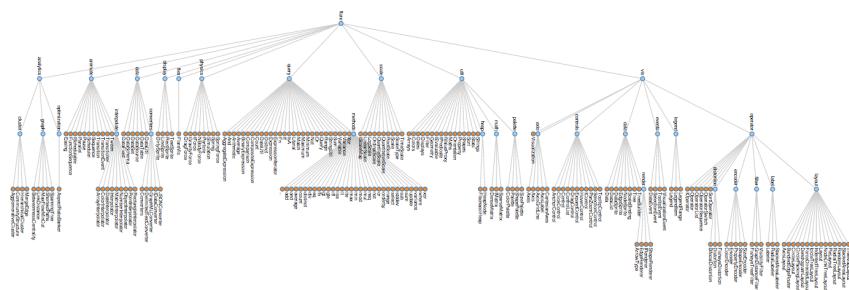


Abbildung 2.15: Baum

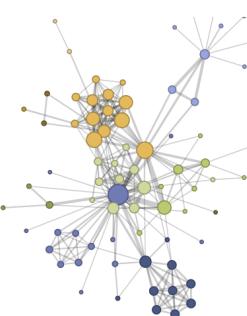


Abbildung 2.16: Node-Link-Diagramm

## Visualisierungstechniken

**Standard 2D/3D** Visualisierungen beinhalten Balkendiagramme, Liniendiagramme, sowie andere zweidimensionale Plots und Karten. Ein Beispiel dafür ist das Index Chart (Abbildung 2.11).

**Multidimensionale** Visualisierungen<sup>2</sup> sind Darstellungen multidimensionaler Datensätze jeder Art. Beispiele sind Parallel Coordinates (Abbildung 2.8) und die Scatterplot Matrix (Abbildung 2.13).

**Symbolische** Visualisierungen setzen auf verschiedene Art und Weise Symbole ein. Das können auf eine Karte projizierte Kuchendiagramme sein (Abbildung 2.17) oder Smileys, die abhängig von den Daten lächeln oder weinen [Che73].

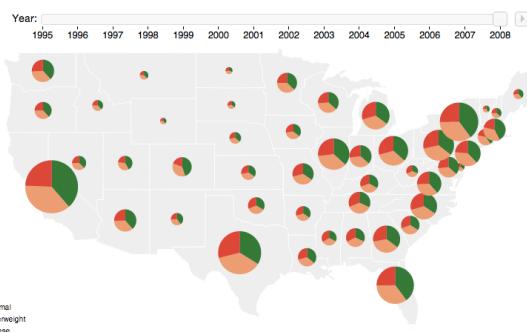


Abbildung 2.17: Symbol Map

**Dense Pixel** Visualisierungen assoziieren jeden Wert einer Dimension mit einem eingefärbten Pixel und platzieren die Pixel einer Dimension nebeneinander. Ein Beispiel dafür ist das Recursive Pattern [KAK95] (Abbildung 2.18).

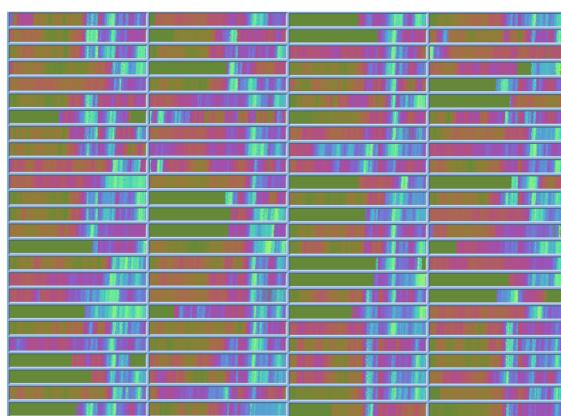


Abbildung 2.18: Recursive Pattern

**Verschachtelte** Visualisierungen repräsentieren Hierarchien, wobei Kindknoten innerhalb ihrer Eltern dargestellt werden. Beispiele dafür sind Treemaps [Shn92] oder Nested Circles (Abbildung 2.19).

<sup>2</sup>Die Bezeichnung stammt von Chi [Chi00] und wird verwendet, da sie logischer erscheint als Keims »geometrically transformed displays«.

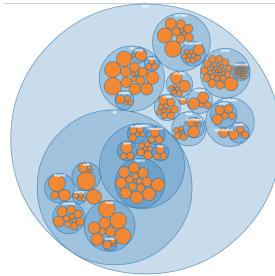


Abbildung 2.19: Nested Circles

## Interaktionstechniken

**Dynamische Projektionen** zeigen dem Benutzer automatisch beispielsweise verschiedene Scatterplots des Datensatzes. Diese Interaktionstechnik eignet sich besonders für multidimensionale Datensätze. Umgesetzt wurde sie zum Beispiel in XGobi [SCB98].

Durch **interaktives Filtern** bestimmt der Benutzer, welche Teilmenge des Datensatzes visualisiert wird. Das kann durch direktes Auswählen (Browsing) oder durch Bestimmen von Eigenschaften der gewünschten Daten (Querying) passieren. Letzteres ist in modernen E-Commerce-Systemen durch Facetten [Yee+03] umgesetzt (Abbildung 2.20).

Abbildung 2.20: Faceted Browsing bei Amazon.de

**Interaktives Zoomen** ermöglicht es dem Benutzer mehr oder weniger Details anzuzeigen. Damit ist nicht nur der computergrafische Vorgang der Skalierung gemeint (wie etwa bei einem Mikroskop), sondern auch schlecht sichtbare Elemente auszublenden (wie z. B. bei Google Maps, siehe Abbildung 2.21).

Durch **Verzerrung** kann ein Bereich der Visualisierung mit hohem Detailgrad angezeigt werden, während der Rest nur wenig detailliert sichtbar ist. Ein bekanntes Beispiel dafür ist das Fisheye (Abbildung 2.22).

**Linking & Brushing** wird vor allem bei mehreren verschiedenen Visualisierungen eingesetzt. Die in einer Visualisierung markierten Daten werden auch in allen anderen Visualisierungen hervorgehoben. Das können zum Beispiel Scatterplots (siehe multidimensionale Visualisierungen in Abschnitt 2.3.3), verschiedene Histogramme (Abbildung 2.23<sup>3</sup>) oder gemischte Visualisierungen sein.

<sup>3</sup><http://square.github.io/crossfilter/>

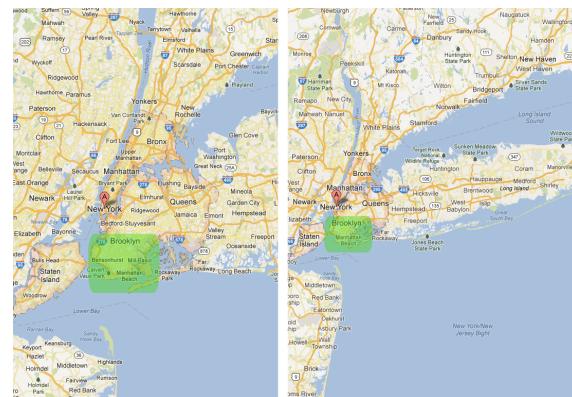


Abbildung 2.21: Interaktiver Zoom bei Google Maps: Im rechten, ausgezoomten Bild fehlt beispielsweise die Interstate 278 (grüne Markierung)

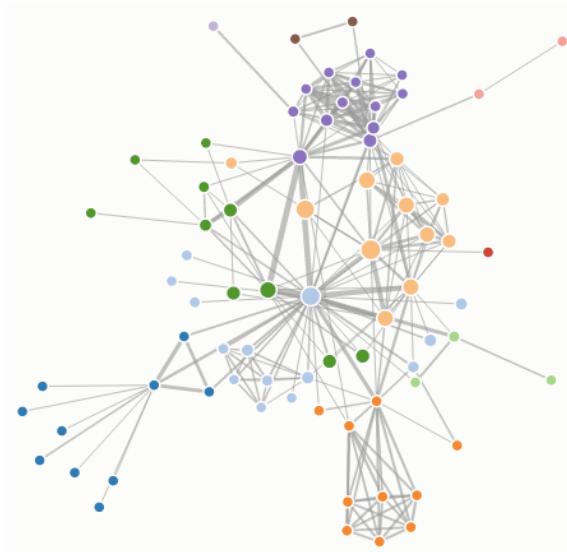


Abbildung 2.22: Fisheye: Das Zentrum der Verzerrung befindet sich ungefähr beim hellblauen Knoten in der Mitte

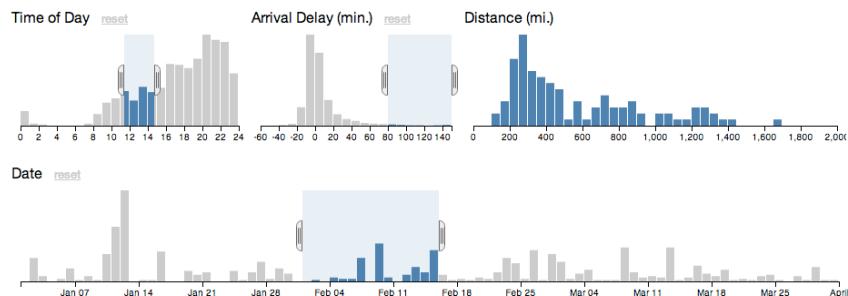


Abbildung 2.23: Linking & Brushing bei Crossfilter.js: Ausgewählt wurden Flüge mit mehr als 80 Minuten Verspätung, die betroffenen Bereiche in anderen Histogrammen wurden automatisch markiert.

In diesem Abschnitt wurde ein Überblick über verschiedene mögliche Informativisualisierungen gegeben, welche bei der Konzeption berücksichtigt werden müssen. Die enthaltenen Daten können beliebig viele Dimensionen besitzen, in Form von Text auftreten oder Netzwerke beschreiben. Das zu erarbeitende Hilfesystem muss also mit jeder Form von Daten umgehen können. Mögliche Visualisierungstechniken beinhalten Standard 2D/3D Visualisierungen (z. B. Balkendiagramme), können aber auch multidimensional sein oder verschachtelt beziehungsweise symbolisch. Für das Hilfesystem bedeutet das, dass von keiner festen Darstellung der Daten ausgegangen werden kann. Die Interaktionstechniken umfassen Filtern, Zoomen und Verzerren der Datendimensionen. Das Hilfesystem kann also beispielsweise keine Annahmen über die verwendete Skala, den sichtbaren Bereich oder die sichtbaren Daten machen.

### 2.3.4 User Assistance

Der Begriff »User Assistance« steht für verschiedene Möglichkeiten, einem Benutzer zu helfen seine Ziele zu erreichen. Gapenne et al. [GLB02] definieren vier Beziehungstypen zwischen Mensch und Technologie, welche auch konkret auf Software übertragen werden können. Einer davon ist Assistance:

The assistance relationship appears as a sub-category of supplementation since it is not a crucial one for the actual and main activity. The function of this type of technology is to qualify and display the state and/or the becoming of the supplementation device which the subject is engaged in.

Assistance sind also zum Beispiel eine Einparkhilfe im Auto oder eine Eieruhr in der Küche: Erfolgreiches Einparken oder Kochen ist auch ohne sie möglich, aber leichter mit ihnen. Supplementation hingegen erweitert die Fähigkeiten des Anwenders, also zum Beispiel eine Handprothese mit der die Pfanne angefasst werden kann.

Rech et al. [RRD07] beschreiben intelligente Assistance in der Softwareentwicklung. Prinzipiell müssen zuerst Daten gesammelt werden, bevor die Assistance generiert und angeboten werden kann (Abbildung 2.24). Folgende Punkte müssen in den einzelnen Schritten beachtet werden.

- Daten sammeln
  - Welche Datenquellen sollen benutzt werden?
  - Wann soll die Datenanalyse stattfinden?
  - Wie können Erkenntnisse aus den Daten gewonnen werden?
- Assistance generieren
  - Für wen ist die Assistance gedacht?
  - Was soll durch Assistance erleichtert werden?
  - Für welchen Prozess/Vorgang ist die Assistance gedacht?
  - Was ist die Zielumgebung der Assistance?
- Assistance anbieten

- Wann soll Assistance angeboten werden?
- Welche Modalität (visuell, akustisch usw.) soll die Assistance anbieten?
- Wo soll die Information angezeigt werden?
- Warum muss dem Benutzer geholfen werden?

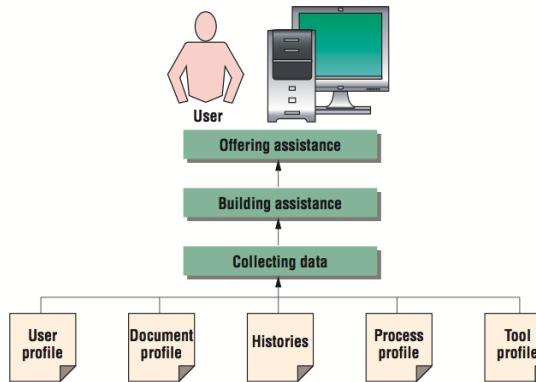


Abbildung 2.24: Intelligente Assistance nach [RRD07]

Besonders die Modalität der Assistance erscheint von großer Bedeutung für eine schnelle und effektive Wissensaufnahme. Wie Information wahrgenommen wird, hat Auswirkungen sowohl auf Verständnis (»Ein Bild sagt mehr als tausend Worte«) als auch auf die benötigte Zeit. Tausend Worte können auf der Suche nach relevanten Informationen immer noch überflogen werden, eine zehnminütige akustische Hilfestellung nicht.

Da kein Medium (Film, Audio, Buch etc.) besser zum Lernen geeignet ist als ein anderes [Cla94; Koz94] und multimodale Erklärungen effektiver als monomodale sind [MM02a], stellt sich die Frage, wie Multimedia-Hilfe konstruiert werden soll. Dazu kann auf die kognitive Multimedia-Lerntheorie zurückgegriffen werden. Sie geht davon aus, dass Menschen grundsätzlich über zwei Kanäle (visuell und verbal) Wissen aufnehmen, entsprechende Repräsentationen bilden und mit vorhandenem Wissen verknüpfen (Abbildung 2.25). Die beiden Aufnahmekanäle sind in ihrer Kapazität beschränkt. Müssen zu viele Informationen verarbeitet werden, kommt es zum »Cognitive Overload« und die Lernfähigkeit wird eingeschränkt. Daraus ergeben sich vier Richtlinien zur Konstruktion der Multimedia-Erklärungen:

- **Gleichzeitigkeit:** Werden akustische und visuelle Mittel eingesetzt, so sollen sie gleichzeitig präsentiert werden [MA91]. So können Lernende ihre visuellen und verbalen Wissensrepräsentationen besser miteinander verknüpfen.
- **Prägnanz:** Unterhaltsame aber irrelevante Details fördern die Erinnerungsfähigkeit an einen Text nicht [GGW89]. Deswegen sollen nur relevante Informationen vermittelt werden.
- **Multimodalität:** Informationen sollten möglichst auf visuelle und verbale Art vermittelt werden, um zu verhindern, dass ein Aufnahmekanal überlastet wird (bspw. durch die Darstellung einer Animation mit On-Screen-Text) [MM99].

- **Keine Redundanz:** Aus dem selben Grund warum zweikanalige Erklärungen vorzuziehen sind, sollte auch Redundanz in einem Aufnahmekanal vermieden werden.

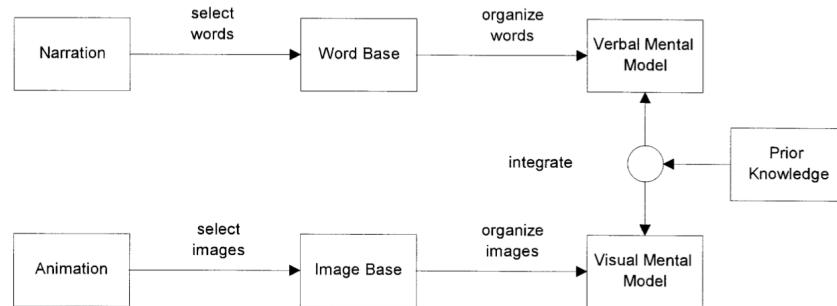


Abbildung 2.25: Kognitive Multimedia-Lerntheorie nach [MM02a]

Wie kann User Assistance im visuellen bzw. verbalen Aufnahmekanal aussehen? Verbale Erklärungen können zur Laufzeit generiert werden [Bau11; Ges12; MCS12], sind jedoch tendenziell zu langatmig formuliert und enthalten viele Wiederholungen. Auf Grund der vorhin erwähnten Designprinzipien »Prägnanz« und »Keine Redundanz« sollte davon wahrscheinlich abgesehen und verbale Erklärungen handgeschrieben oder, falls automatisch generiert, zumindest sehr kurz gehalten werden.

Die naheliegendste Möglichkeit für visuelle User Assistance sind Bilder. Illustrationen/Schemata sind in der Tat hilfreich, solange sowohl die Bestandteile des Systems (z. B. eine Luftpumpe) als auch die einzelnen Schritte des Prozesses (z. B. Luft einziehen, Luft auspumpen) dargestellt werden [MG90]. Auch Comics sind gut geeignet um Sachverhalte zu erklären (Beispiele dafür sind u. a. [McC94; McC08]) und eine ansprechende Variante für User Assistance [Web+12].

Animationen im Sinne von kurzen Videos mit abstrahiertem Inhalt können ebenfalls ein geeigneter Weg sein um Wissen zu vermitteln, sofern die erwähnten Designprinzipien eingehalten werden [MM02b]. Allerdings ist es möglich, dass Benutzer mit wenig Vorwissen weniger davon profitieren also solche mit viel Vorwissen [Kal08]. Wenn Animationen eingesetzt werden, sollte den Benutzern Kontrolle über die Geschwindigkeit der Animation gegeben und wichtige Teile gekennzeichnet werden [WMS11]. Bei Web User Interfaces besteht die Möglichkeit konkrete Elemente der InfoVis für die Animation heranzuziehen. Das würde dem Prinzip der »Semantic Transparency« [KK09] entsprechen, wonach Benutzer möglichst direkt auf den für ihre Ziele relevanten UI-Elementen arbeiten sollen.

## Sensemaking

Im vorhergehenden Abschnitt wurde der Prozess der Wissensvermittlung betrachtet. Um Wissen dauerhaft aufzunehmen, müssen Lernende den Inhalt *verstehen*. Der Begriff »Understanding« (gleichbedeutend mit Sensemaking) wird angewandt, wenn das Gehirn mehr Elemente des Inhalts (z. B. Variablen einer Gleichung) verarbeiten muss als ins Arbeitsgedächtnis passen [SMP98]. *Verstanden* hat der Lernende dann, wenn diese Elemente zu einem sogenannten Schema (mentalnen Modell) verarbeitet

wurden und als ein einziges Element im Arbeitsgedächtnis gehalten werden können (z. B. eine komplexe Zahl statt drei Variablen  $a$ ,  $b$  und  $i$ ).

Klein [KMH06b] beschreibt in seinem Data/Frame-Modell (Abbildung 2.26), wie dieser Prozess funktioniert. »Data« beschreibt dabei die Elemente des Inhalts und »Frame« das mentale Modell oder Schema. Ähnlich dem Visual Analytics Process (Abschnitt 2.3.3) ist Sensemaking ein iterativer Prozess, aber ohne wirklichen Anfang oder Ende. Der Lernende startet mit einem Frame, versucht die Daten einzupassen und nimmt Anpassungen vor bzw. verwirft ein Frame, wenn es nicht weiter durch Daten gestützt wird. Dabei sollte der Lernende möglichst selbstständig zu einer Lösung kommen, nicht zu viele Daten gleichzeitig sehen und während des Prozesses keine Hypothesen aufstellen müssen, die nicht überprüft werden können [KMH06a].

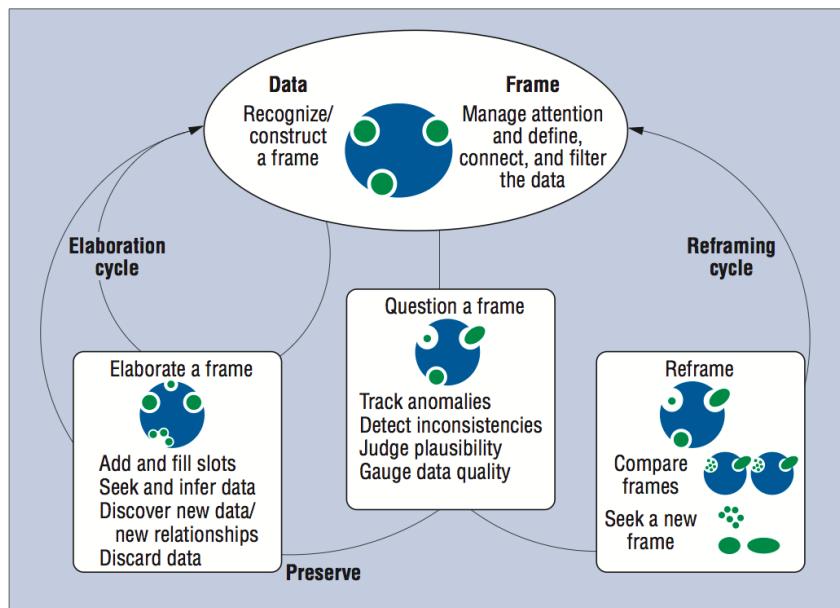


Abbildung 2.26: Data/Frame Modell nach [KMH06a]

Während des Verständnisprozesses ist es außerdem hilfreich, Wissen in irgendeiner Form externalisieren zu können, beispielsweise mit Hilfe einer Mind Map [QF05; Nov07; Uma10]. Andere Menschen können dadurch auf bereits durchgeführte Verständnisprozesse anderer aufbauen, wobei die Struktur aber wichtiger ist als der tatsächliche Inhalt [FCK12].

Nachdem betrachtet wurde, wie Verständnis funktioniert, stellt sich die Frage, *was* vermittelt werden soll um Menschen mit wenig InfoVis-Erfahrung zu helfen. Dazu gehört laut Grammel [Gra12, S. 127] unter anderem das mentale Modell des Benutzers zu verwenden, beispielsweise indem dieselben Bezeichnungen verwendet werden, und zu lehren, wie Visualisierungen verwendet und interpretiert werden. Liu und Stasko [LS10] schlagen folgende Definition für ein mentales Modell im Bereich der Informationsvisualisierung vor, wobei vor allem der erste Punkt über Struktur und Verhalten für ein Hilfesystem wichtig erscheint:

A mental model is a functional analogue representation to an external interactive visualization system with the following characteristics:

- The structural and behavioral properties of external systems are preserved in mental models.
- A mental model can preserve schematic, semantic or item-specific information about the underlying data.
- Given a problem, a mental model of an interactive visualization can be constructed and simulated in working memory for reasoning.

## 2.4 Verwandte Arbeiten

Im Folgenden werden verwandte Arbeiten behandelt. Zuerst befasst sich Abschnitt 2.4.1 mit Mashup Plattformen. Darauf folgen Visualisierungstools und Teilespekte davon, die das Sensemaking der Nutzer verbessern sollen. Danach werden verwandte Arbeiten zu User Assistance Konzepten in Mashups und im Allgemeinen vorgestellt und zuletzt eine Zusammenfassung gegeben.

### 2.4.1 Webbasierte Mashup Plattformen

In diesem Abschnitt werden Mashup Plattformen vorgestellt, mit denen Endnutzer Datenquellen oder UI Komponenten kombinieren können.

Yahoo Pipes<sup>4</sup> ist eine Mashup Plattform von Yahoo, wo der Benutzer verschiedene Webservices mit Hilfe von Operatoren kombinieren kann, indem er sie interaktiv verdrahtet. Pipes erfüllt die funktionale Anforderung *Intro*, da zwei Tutorialvideos angeboten werden und eine kurze Beschreibung zu jedem Element inklusive Link zu einem Beispiel und mehr Informationen vorhanden ist. Die *Kommunikation* ist durch die Drähte sichtbar, allerdings ist die Richtung des Datenflusses nicht sofort eindeutig. Pipes erleichtert die *Bedienung* indem während der Drag & Drop Operation mögliche Anknüpfungspunkte für Drähte blau eingefärbt werden. Allerdings erschwert es die Bedienung gleichermaßen, da kaputte Kompositionen erstellt werden können und Pipes keine aussagekräftigen Fehlermeldungen ausgibt. Für die Arbeit mit VizBoard ist es ebenfalls wichtig, die Kommunikation zwischen Komponenten nachvollziehen zu können. Da bei VizBoard der Datenfluss in beide Richtungen fließen kann, sollte darauf geachtet werden, dessen Richtung klar ersichtlich ist.

DashMash [Cap+11] ist eine Mashup Plattform, die besonders auf Endnutzer ausgerichtet ist. Dazu setzt sie auf das WYSIWYG<sup>5</sup> Prinzip und abstrahiert von Mashup-Interna wie dem Binding zwischen Datenquellen und Komponenten. Es ist nicht möglich, Erkenntnisse aus den Daten mit anderen zu teilen oder selbst Anmerkungen anderer zu lesen. Obwohl keine sichtbaren<sup>6</sup> Hilfestellungen bezüglich *Intro*, *Bedienung* oder *Kommunikation* verfügbar sind, fiel die Nutzerstudie mit 35 Teilnehmern durchweg positiv aus. Der Grund dafür ist wahrscheinlich, dass die Erklärungen am Anfang der Studie so gewählt wurden, dass die Testpersonen ein »korrektes« mentales Modell von DashMash hatten. Für die Hilfe in VizBoard sollte

<sup>4</sup><http://pipes.yahoo.com/pipes/>

<sup>5</sup>What you see is what you get.

<sup>6</sup>Ein Demo-Video ist unter <http://home.dei.polimi.it/cappiell/demo/DemoDashMash.mov> verfügbar.

deswegen auch soweit möglich Technisches abstrahiert oder vor dem Benutzer versteckt werden. Des weiteren sollten Erklärungen dem mentalen Modell des Benutzers entsprechen.

Apache Rave<sup>7</sup> ist eine Mashup Plattform von Apache und Grundlage für OMELETTE [Chu+12]. OMELETTE stellt dem Benutzer verschiedene Widgets zur Verfügung, die er kombinieren kann. Zum Beispiel ein Widget zur Visualisierung des Wassерpegels und eine Karte, die Hochwassermeldungen zeigt. Im Gegensatz zu VizBoard besteht keine Einschränkung auf eine bestimmte Domäne. Es ist möglich Widgets zu kommentieren. Die *Kommentare* werden aber nur im Widget Store angezeigt, da sie eher dem Feedback zum Widget als dem Verständnis des Inhalts dienen. Bezuglich der *Bedienung* oder *Kommunikation* bestehen keine Hilfestellungen. Ein *Intro* wird im Widget Store in Form einer kurzen Beschreibung gegeben, welche der Entwickler verfasst hat. Die Hilfestellungen für den Benutzer fokussieren sich eher auf die Erstellung von Mashups als deren Nutzung [Cho+13]. So wird ein Pattern Recommender eingesetzt, der weitere Widgets zur Integration in die Komposition vorschlägt sowie eine Automatic Composition Engine, die Widgets automatisch verbindet und damit Daten- und Kontrollflüsse ermöglicht.

In diesem Abschnitt wurden Mashup Plattformen vorgestellt, mit deren Hilfe Webservices oder UI Komponenten kombiniert werden können. Die wichtigste Erkenntnis war dabei, dass kaum Hilfestellungen nötig sind, wenn der Benutzer ein passendes mentales Modell der Anwendung hat. Dieses kann mit Hilfe von Erklärungen vermittelt werden. Erklärungen des Hilfesystems sollten deswegen ein konsistentes Modell der Anwendung und einheitliche Begriffe verwenden.

## 2.4.2 Visualisierungstools

Im Folgenden werden Visualisierungstools vorgestellt. Sie bieten mindestens multiple Sichten (ähnlich wie Mashups), sind selbst Mashups (wenn nicht anders angegeben webbasiert) oder besonders populär.

Weave<sup>8</sup> ist eine Webapplikation, mit der Benutzer ein Mashup aus Visualisierungen erstellen können. Mögliche Datenquellen sind strukturierte Daten wie Tabellen und CSV Dateien. Benutzer können aus verschiedenen Visualisierungen wählen und diese an Daten binden. Das User Interface (Abbildung 2.27) ist allerdings sehr komplex und Hilfe gibt es nur in Form einer Online-Dokumentation und einem kurzen Tutorialvideo.

Tableau<sup>9</sup> ist eine Desktopapplikation, mit der aus Tabellen oder Datenbanken Diagramme erstellt werden können. Es ist der kommerzielle Nachfolger von Polaris [SH02] und laut Gartner [Sch+13] unter den Marktführern im Bereich Business Intelligence Platforms. Tableau setzt das Prinzip von »Semantic Transparency« um, indem der Benutzer die zu visualisierenden Daten per Drag & Drop auf dem Diagramm platziert. Die Software muss keine umfassenden Hilfestellungen zur *Bedienung* anbieten, da der Benutzer hauptsächlich Daten an das Diagramm bindet und den Diagrammtyp wechselt. Jede Aktion hat einen sofort sichtbaren Effekt, welcher mit der Undo-Funktion rückgängig gemacht werden kann. Diese drei Eigenschaften

---

<sup>7</sup><http://rave.apache.org/>

<sup>8</sup><http://oicweave.org/>

<sup>9</sup><http://www.tableausoftware.com/>

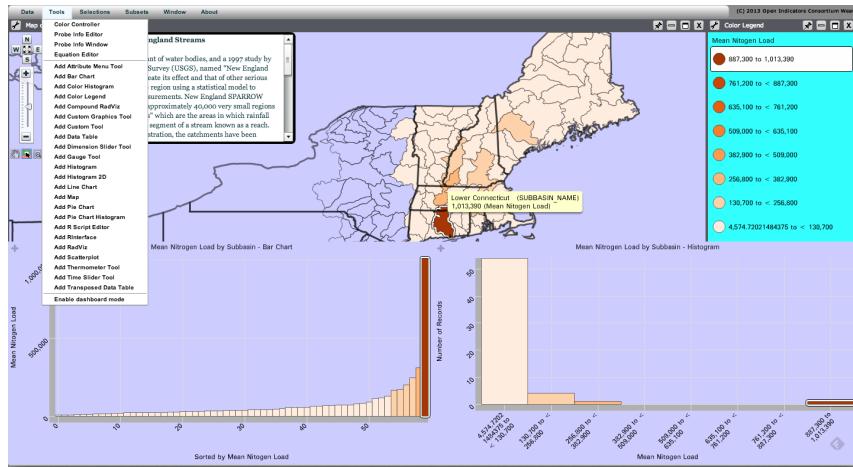


Abbildung 2.27: Weave User Interface

von Tableau (Semantic Transparency, Effekt sofort sichtbar, Undo/Redo) machen es sehr interaktiv, ohne dabei den Benutzer zu frustrieren. Allerdings geht Tableau davon aus, dass der Benutzer die Daten und damit verbundenen Konzepte kennt und bietet diesbezüglich keine Hilfestellungen an.

Choosel [GS10] ist ein Mashup, welches besonders in der Domäne Informationsvisualisierung unbedarfte Nutzer dabei unterstützen soll, Daten zu erforschen und Visualisierungen iterativ zu verfeinern. Dabei setzt es auf direkte Manipulation in Form einer erweiterten Drag & Drop Funktion. Mögliche Drop-Ziele werden während des Vorgangs hervorgehoben und eine Vorschau des Drop-Ergebnisses wird angeboten. Wie Tableau setzt Choosel voraus, dass der Benutzer die verwendeten Daten kennt. Im Gegensatz zu VizBoard sind bei Choosel die verfügbaren Komponenten festgelegt und stammen vom selben Entwickler. Falls im Hilfesystem für VizBoard Drag & Drop angewendet wird, ist Choosel ein gutes Vorbild.

MoSaiC [SSZ11] ist ein Tool mit dem Benutzer online kollaborativ Dokumente erstellen können. Wie bei VizBoard ist die Domäne damit festgelegt. Das Tool besteht aus verschiedenen Sichten für die Rollen der Beitragenden (Collaborators) und Organisatoren (Coordinators) und ist damit komposit, es können aber keine zusätzlichen Views aufgerufen werden. Inwieweit MoSaiC den Benutzern Hilfestellungen gibt, ist nicht bekannt, da der Autor keine Screenshots oder Videos finden konnte.

## Sensemaking

Dieser Abschnitt befasst sich mit Teifunktionen von Visualisierungstools, die das Sensemaking der Benutzer verbessern sollen. Hauptsächlich wird dafür eine Kommentar- bzw. Annotationsfunktion eingesetzt.

Groth und Streefkerk [GS06] präsentieren ein generisches Annotationssystem für Informationsvisualisierungen am Beispiel einer 3D Visualisierung von Molekülen. Grundlage des Systems sind Nutzerinteraktionen wie »rotieren«, »zoomen« etc., welche geloggt und als Graph abgespeichert werden. An den Knoten des Graphen – Zustände der InfoVis – können beliebig viele *Kommentare* abgelegt werden. Es ist allerdings nicht möglich, innerhalb dieser Sichten noch einmal Elemente zu referenzieren (z. B. ein bestimmtes Atom), weswegen das System am besten für InfoVis mit

uneingeschränkten Navigationsmöglichkeiten geeignet ist. Zum Beispiel einer Karte, wo mit Hilfe der Interaktionen Pan & Zoom theoretisch ein beliebiger Punkt auf der Welt den Viewport ausfüllen kann und so eindeutig ist, worauf sich der Kommentar bezieht.

Heer et al. [HVW07] entwickelten mit sense.us einen Prototyp zur asynchronen Kollaboration mit interaktiven Visualisierungen. Benutzer können *Kommentare* zu Visualisierungen hinzufügen und sie mit grafischen Annotationen (z. B. geometrische Formen, Pfeile, Text) versehen. Die Kommentare wurden dabei aber nicht an die Daten selbst, sondern an die Visualisierung hinzugefügt. Es war möglich, Kommentare in anderen Visualisierungen zu referenzieren. Die am meisten verwendeten Annotationstypen waren Pfeile und Text. Das alles sollte beim Entwurf der Kommentarfunktion des Hilfesystems berücksichtigt werden.

Ebenfalls erwähnenswert ist ManyEyes [Vie+07]. Benutzer können Datensätze hochladen und eine aus mehreren vorgegebenen Visualisierungen wählen. Danach können sie ähnlich wie bei sense.us die Visualisierungen *kommentieren*. Wie bei sense.us können bestimmte Elemente oder Bereiche einer Visualisierung mit Kommentaren versehen werden, diese werden aber nicht visualisierungsübergreifend wieder verwendet.

Elias und Bezerianos [EB12] entwickelten ein Annotations- bzw. *Kommentarsystem* für Business Intelligence Dashboards. Dafür interviewten sie Domänenexperten und erarbeiteten sieben funktionale Anforderungen an ein annotationsgestütztes Dashboard. Dazu gehört, dass Kommentare mehrere Datenpunkte und Charts referenzieren können. Sie beschreiben außerdem ihr Handling von Annotationen und Kommentaren bei dynamischem Kontext sowie von unterschiedlichen Granularitätsstufen (Jahr/Monat/Tag). Diese Erkenntnisse können für VizBoard ebenfalls nützlich sein.

Shrinivasan et al. [SW08] entwickelten ein Framework zur Informationsvisualisierung, welches den analytischen Denkprozess unterstützen soll. Es besteht aus drei verschiedenen Views: Den InfoVis selbst, einem »Knowledge View« und einem »Navigation View«. Im Knowledge View kann der Benutzer seine Hypothesen und Analysen externalisieren, verwalten und überprüfen, sowie mit Zuständen der InfoVis verlinken. Der Navigation View erlaubt es ihm, zu einem beliebigen Punkt im Laufe der Interaktionen zurück zu gehen. In VizBoard könnte eine Variante des Knowledge Views dazu verwendet werden, um eine nutzergenerierte, strukturiertere Zusammenfassung eines Datensatzes zu erstellen. Mit dessen Hilfe könnte der Verständnisprozess zukünftiger Benutzer unterstützt werden. Ein Navigation View könnte zusätzlich die *Bedienung* in VizBoard verbessern.

In diesem Abschnitt wurden verschiedene Visualisierungstools und Ansätze, um das Sensemaking in ihnen zu verbessern, vorgestellt. Dabei zeigte sich am Beispiel Tableau, dass wenig Hilfestellungen nötig sind, sofern der Benutzer Systemzustände als Effekt eigener Interaktionen einfach identifizieren kann und die Anwendung offensichtliche Rücksetzmöglichkeiten bietet. Im Fall von VizBoard ist die Situation etwas anders, da mehrere Views gleichzeitig sichtbar sind (bei Tableau ist es immer nur ein Diagramm) und der Systemzustand sich unabhängig von einer Nutzerinteraktion verändern kann (durch Komponenten, die Daten externer Webservices laden). Die Rücksetzmöglichkeit ist aber auch Teil von Shneidermans Mantra [Shn96] und der ISO Norm 9241-110 »Grundsätze der Dialoggestaltung«, weswegen sie auf alle

Fälle umgesetzt werden sollte.

Um den Verständnisprozess des Benutzers zu unterstützen, verwenden die meisten Systeme grafische Annotationen in Kombination mit Kommentaren. Im Fall von [EB12] konnten die Kommentare den Benutzern helfen, neue Diagrammtypen zu verstehen. Probanden in sense.us fanden sowohl eigene als auch fremde Kommentare hilfreich. In Tableau ist es wie in [EB12] möglich, Kommentare an Datenpunkten zu erstellen. Die Nützlichkeit von Kommentaren scheint damit außer Frage zu stehen und sie sollten auch in VizBoard implementiert werden. Dabei ist die Arbeit von Elias et al. ein guter Startpunkt, wobei aber zu beachten ist, dass die Visualisierungskomponenten bei VizBoard von fremden Entwicklern stammen und »Black Boxes« sind. Von keiner der Arbeiten aufgegriffen wurde die Möglichkeit, dass dem Benutzer Teile der Daten oder der darin verwendeten Konzepte unklar sind.

### 2.4.3 User Assistance

Dieser Abschnitt befasst sich mit verschiedenen Ansätzen zur User Assistance, welche die Anforderungen Verständnis, Vollständigkeit oder Unaufdringlichkeit betrifft. Dazu zählen notwendige Funktionen des Hilfesystems, mögliche Fragestellungen des Nutzers oder Interaktionskonzepte. Zunächst werden Ansätze vorgestellt, die behandeln, was dem Benutzer erklärt werden muss. Darauf folgen Konzepte zur Präsentation der Erklärung und zuletzt zur Reduktion von notwendigen Erklärungen (beispielsweise durch kluge Interaktionstechniken).

Lim und Dey [LD09] kategorisieren mögliche Fragestellungen von Nutzern in kontextadaptiven Anwendungen. Dabei ändert die Anwendung abhängig vom Nutzer- bzw. Nutzungskontext ihren Zustand (z. B. schaltet Smartphone auf lautlos) und der Benutzer will wissen, warum (z. B. weil er Abends in der Oper ist). Grundlage für Lims Ausführungen ist die Annahme, dass der Nutzer nur eingeschränkt Einfluss auf die Arbeitsweise des Systems hat, etwa durch Hinzufügen eigener Regeln. Das ist im letzten Schritt des VizBoard Workflows nicht gegeben, da der Benutzer die Visualisierungskomponenten selbst ausgewählt hat. Die Fragestellungen könnten aber hilfreich sein, wenn es um die Komponenten selbst geht (»Warum werden gerade diese Daten angezeigt?«) und so die *Verständlichkeit* und *Vollständigkeit* erhöhen.

Abgesehen von allgemeinen Fragestellungen wäre es wichtig zu wissen, ob bestimmte Mashupkonzepte besondere Beachtung finden müssen. Cao et al. [Cao+10] untersuchten mit Hilfe der Thinking Aloud Methode [SBS94] das Verhalten von Endnutzern, wenn sie Mashups erstellen, sowie deren Debuggingstrategien, wenn etwas nicht funktioniert. Dazu benutzten sie das inzwischen eingestellte Microsoft Popfly. In ihrer Studie hatten alle Probanden Probleme mit dem Konzept des Datenflusses, also welche Daten wann zwischen welchen Komponenten ausgetauscht werden. In VizBoard sollte überlegt werden, den Daten- bzw. Kommunikationsfluss explizit anzuseigen und damit die *Verständlichkeit* zu verbessern.

In einem Mashup kommunizieren die beteiligten Komponenten miteinander. Weder ist dieser Vorgang für den Benutzer offensichtlich noch sollte er das Kommunikationsparadigma des Mashups lernen müssen. Chudnovskyy et al. [Chu+13] kategorisieren Probleme und Lösungsansätze, mit deren Hilfe Benutzer die Kommunikation zwischen Komponenten besser wahrnehmen können. Darunter finden sich Self-Descriptive Design, welches visuelle Hinweise und Annotationen verwendet, sowie

die Surprise-Explain-Reward Strategie (Überraschung, Erklärung, Belohnung), die zeitgleich mit einem ungewöhnlichen Vorgang eine Erklärung dazu anbietet. Diese beiden Lösungsansätze könnten gut mit einer expliziten Darstellung des Kommunikationsflusses kombiniert werden.

VizBoard ist ein Mashup von Informationsvisualisierungen, weswegen häufige Hindernisse für Endnutzer in diesem Bereich identifiziert werden müssen. Grammel [Gra12] verfasste seine Doktorarbeit zum Thema, wie InfoVis-Anfänger Visualisierungen konstruieren. Um sie dabei zu unterstützen und ihr *Verständnis* zu verbessern, schlägt er unter anderem vor, ihr mentales Modell zu verwenden (z. B. mit ihnen bekannten Begriffen), kontextuelle Hilfe anzubieten und es ihnen zu erleichtern, mehr über InfoVis zu lernen. Was genau ein mentales Modell bei InfoVis ausmacht, untersuchten Liu et al. [LS10]. Demzufolge ist es besonders wichtig, die Struktur und das Verhalten der kompositen InfoVis zu erklären.

Erklärungen können nicht nur Bilder oder Videos sein, sondern auch textbasiert. Einige Arbeiten, wie z. B. [Ges12; MCS12], befassen sich mit der Generierung von textuellen Erklärungen. In beiden Arbeiten ist eine formale Repräsentation des Erklärungsgegenstands notwendig, beispielsweise als Baum von Vorbedingungen und »ist Teil von« Relationen oder einer Ontologie. In der Evaluation von Matheson et al. merkten einige Probanden an, dass deren Erklärungen zu langatmig seien, was gegenläufig zur Anforderung *Verständlichkeit* ist.

Um kontextuelle Hilfe mit Bildern einfach umsetzen zu können, entwickelten Yeh et al. [Yeh+11] ein Tool für Entwickler, Lehrer oder den technischen Support. Mit dieser Software sollen sie einfach Screenshots einer Anwendung erstellen und Kommentare sowie grafische Annotationen hinzufügen können. Eine Nutzerstudie wurde nicht durchgeführt, dennoch erscheint dieser Ansatz geeignet, um einfache Ursache-Wirkung Relationen zu veranschaulichen. Um in VizBoard die *Einheitlichkeit* der Hilfe zu gewährleisten, müsste die Generierung aber automatisiert werden.

Ein ähnlicher Ansatz von Hilfe mit Bildern ist der Einsatz von Comics. Webb et al. [Web+12] überprüften in einer Nutzerstudie, wie User Assistance in Form eines Comics gegenüber einer PowerPoint-Präsentation angenommen wird. Comics lagen in allen Fragestellungen (einfache Nutzung, attraktiv, nützlich, verständlich) vorne, am Besten wurden Comics angenommen, die eine Metapher beinhalteten. Da von Komponentenentwicklern nicht erwartet werden kann, einen Comic zu erstellen, müsste dieser generiert werden. Damit befassten sich unter anderem schon [KSS96; SRL06; CTC09; WTC12] in unterschiedlichen Kontexten, es ist also möglich.

Um auf Webseiten kontextuelle Hilfe anbieten zu können, entwickelten Chilana et al. LemonAid [CKW12]. Dabei wählen Benutzer ein beliebiges UI Element aus und bekommen dann eine Liste von Fragen und Antworten dazu angezeigt. Beispielsweise könnte ein HTML `input` Element ausgewählt werden, in das die Telefonnummer eingetragen werden muss. LemonAid erklärt, warum die Telefonnummer unverzichtbar ist, indem es auf die entsprechende FAQ Seite verweist. Da VizBoard webbasiert ist, kann dasselbe Interaktionspattern auch dort angewendet werden. Visualisierungskomponenten können UI Elemente aber dynamisch erzeugen, darauf müsste Rücksicht genommen werden.

Erklärungen zur Bedienung sind nötig, wenn der Benutzer nicht weiß, wie er eine Aktion ausführen kann (»Gulf of Execution« nach Norman [Nor86]). Der Bedarf an Erklärungen kann somit reduziert werden, wenn es die Anwendung nicht zu diesem

Gulf of Execution kommen lässt. Zu diesem Zweck wurden einige Richtlinien zur User Interface Gestaltung entwickelt. Dazu gehören unter anderem die ISO9241-110, Sheidermans Mantra [Shn96] oder die Heuristiken nach Nielsen [Nie94]. Forsell und Johansson [FJ10] führten eine Nutzerstudie durch, um aus diesen verschiedenen Richtlinien bzw. Heuristiken zur Heuristischen Evaluation [Nie94] diejenigen herauszufiltern, welche bei Informationsvisualisierungen hilfreich sind. Darunter finden sich Support für Undo/Redo, visuelle Konsistenz und den mentalen Aufwand des Nutzers gering zu halten. So kann die *Verständlichkeit* und *Unaufdringlichkeit* des Hilfesystems von VizBoard erhöht werden.

Mit Designrichtlinien im weitesten Sinne beschäftigten sich auch Kuttal et al. [KSR13]. Ein Nachteil von Yahoo Pipes war, dass kaputte Mashups entstehen können. Also entwickelten sie ein System zur automatischen Fehlererkennung in Yahoo Pipes, welches auch gleichzeitig die Fehlermeldungen verständlicher formulierte. In der Nutzerstudie zeigte sich, dass Benutzer mit dem System erfolgreicher Mashups erstellen konnten als ohne. Die Autoren stellen in ihrer Arbeit auch Designrichtlinien vor und raten dazu, keine technische Begriffe zu verwenden und Undo-Möglichkeiten sowie kontextuelle Hilfe anzubieten. Das deckt sich mit den Empfehlungen bisher betrachteter verwandter Arbeiten.

Um die Undo-Funktionalität in Tableau zu verbessern, entwickelten Heer et al. [Hee+08] eine grafische History für Tableau. Sie zeigt Screenshots des Visualisierungszustands, die chronologisch wie eine Timeline angeordnet sind. Ein Klick auf einen Screenshot setzt die Visualisierung in den angezeigten Zustand zurück. Dieses Konzept könnte zur *Verständlichkeit* von VizBoard beitragen, wenn es umsetzbar ist.

In diesem Abschnitt wurden verschiedene Ansätze zur User Assistance im Allgemeinen und bei Mashups sowie InfoVis im Speziellen betrachtet. Zu der Frage, was erklärt werden muss, konnte als Antwort die Struktur und das Verhalten der Anwendung identifiziert werden, da diese Teil des mentalen Modells des Benutzers sind. Es wurde außerdem gezeigt, dass textuelle Erklärungen langsam und deswegen visuelle Hilfestellungen besser geeignet sein können. Dazu können Screenshots oder Comics verwendet werden. Die Herausforderung bei VizBoard wäre in diesem Fall, dass sie automatisch generiert werden müssten. Um die Interaktion mit VizBoard besser zu gestalten, kann auf Designrichtlinien für Informationsvisualisierungen zurückgegriffen werden. Diese beinhalten unter anderem visuelle Konsistenz, Undo/Redo, zusätzliche Informationen anzeigen oder den mentalen Aufwand des Benutzers zu reduzieren. Bei VizBoard sind viele der Richtlinien abhängig von den Komponentenentwicklern, aber einige können auch durch das Hilfesystem umgesetzt werden.

## 2.5 Zusammenfassung

In diesem Abschnitt wird eine Zusammenfassung des Kapitels gegeben, welches aus dem Szenario (Abschnitt 2.1), der Anforderungsanalyse (Abschnitt 2.2), den Grundlagen (Abschnitt 2.3) und den verwandten Arbeiten (Abschnitt 2.4) besteht.

Im Szenario wurde eine mögliche Situation anschaulich gemacht und Problemstellungen aufgezeigt. Dazu gehört unter anderem, dass der Benutzer nicht weiß, wie er

Komponenten bedienen kann oder wie sie miteinander kommunizieren. Eine weitere Hilfestellung ist, dass in den Daten enthaltene Konzepte erklärt werden sollen.

Auf Basis des Szenarios wurde eine Anforderungsanalyse durchgeführt. Funktionale Anforderungen beinhalten die zu leistenden Hilfestellungen, unter anderem wie erwähnt zur Bedienung, Kommunikation oder Verlinkung. Als nichtfunktionale Anforderungen wurden unter anderem die Korrektheit, Vollständigkeit, Wiederverwendbarkeit und Verständlichkeit der Hilfestellung selbst und die Unaufdringlichkeit der Präsentation identifiziert.

Danach wurde in den Grundlagen ein Überblick über die CRUISe Mashup Plattform gegeben, auf der EDYRA und VizBoard aufzubauen. Es wurde aufgeführt, dass Propertys den Zustand einer Komponente vollständig beschreiben und Komponenten über Links zwischen Events und Operationen miteinander Nachrichten austauschen. Darauf folge eine kurze Einführung in semantische Daten. Dort wurden die wichtigsten Konzepte (Klassen, Instanzen und Propertys) erläutert sowie gängige Beschreibungssprachen (RDF, RDFS, OWL) aufgezeigt. Im Abschnitt über Informationsvisualisierungen wurde der Begriff als »the use of computer-supported, interactive, visual representations of abstract data to amplify cognition« definiert. Mit Hilfe einer Klassifikation anhand der dargestellten Daten, Visualisierungs- und Interaktionstechnik einer InfoVis konnte ein Überblick über selbige gegeben werden. Zuletzt wurden in diesem Abschnitt die Begriffe »User Assistance« und »Sensemaking« erklärt. User Assistance ist eine Hilfestellung für den Benutzer, der sein Ziel aber auch ohne sie erfüllen kann. Sensemaking meint den menschlichen Verständnisprozess. Es wurde beispielsweise festgestellt, dass das Medium selbst keinen Einfluss auf die Lernfähigkeit einer Person hat und möglichst gleichzeitig multimodal erklärt werden sollte. In der Domäne InfoVis müssen besonders das Verhalten und die Struktur einer Visualisierung erklärt werden, um das mentale Modell des Benutzers anzusprechen.

Nach den Grundlagen folgte ein Überblick über verwandte Arbeiten. Dieser wurde in webbasierte Mashup Plattformen, Visualisierungstools und Konzepte zur User Assistance eingeteilt. Nach der Betrachtung von Mashups konnte am Beispiel von DashDash festgestellt werden, dass kaum Hilfestellungen nötig sind, wenn der Benutzer ein passendes mentales Modell der Anwendung hat. Das lässt darauf schließen, dass Erklärungen in VizBoard besonders Rücksicht auf jenes Modell nehmen sollten. Die Visualisierungstools setzten zur Unterstützung des Benutzers auf Undo/Redo, wobei aber im Unterschied zu VizBoard die Anwendung von denselben Entwicklern geschrieben wurde wie die Visualisierungen und somit mehr Kontrolle darüber möglich war. Ein Teilabschnitt über Visualisierungstools befasste sich mit Konzepten, welche das Sensemaking von Daten verbessern sollen. Das populärste davon war Benutzern zu erlauben, Kommentare und Annotationen zu erstellen. Diese wurden aber entweder nicht an den Daten selbst gespeichert und waren kaum wiederverwendbar (sense.us [HVW07], ManyEyes [Vie+07]) oder die Entwickler des Kommentarsystems schrieben auch die Visualisierungen (Business Intelligence Dashboard [EB12]). Dann konnten sie Annahmen über die vorhandenen Visualisierungen treffen und die Darstellung und Interaktion der Kommentare entsprechend anpassen. Im Abschnitt über User Assistance waren Undo/Redo, visuelle Konsistenz und das mentale Modell des Benutzers zu verwenden, wiederkehrende Vorschläge. Es wurde außerdem gezeigt, dass Hilfestellungen mit Hilfe von Bildern gegeben werden

können, beispielsweise in Form von Screenshots oder Comics. Letztere konnten in unterschiedlichen Kontexten (Chats [KSS96], Storytelling [CTC09; WTC12] usw.) bereits automatisch generiert werden, weswegen das bei VizBoard kein Hindernis sein sollte. Beachtenswert ist, dass kaum eine der vorgestellten Arbeiten berücksichtigte, dass in den Daten enthaltene Konzepte (wie z. B. BIP, Yen usw.) dem Benutzer unbekannt sein könnten.

# 3 Konzeption

Basierend auf dem Szenario (Abschnitt 2.1), den Grundlagen (Abschnitt 2.3) und den Erkenntnissen aus den verwandten Arbeiten (Abschnitt 2.4) wird ein Konzept zur User Assistance für das webbasierte, composite Informationsvisualisierungssystem VizBoard erarbeitet. Die einzelnen Hilfestellungen können dabei in allgemeine, in jedem CRUISe-basierendem Mashup Einsetzbare (Abschnitte 3.2, 3.3, 3.4, 3.5, 3.8, 3.9) und domänen spezifische, besonders für Informationsvisualisierungen Nützliche (Abschnitte 3.6, 3.7), eingeteilt werden. Die Ziele richten sich nach der Anforderungsanalyse (Abschnitt 2.2).

## 3.1 Einführung

In diesem Abschnitt werden für die Konzeption notwendige Vorberichtigungen getätigt. So wird zunächst ein Rollenmodell eingeführt (Abschnitt 3.1.1), damit Aufgaben korrekt verteilt werden. Danach werden Designrichtlinien für das User Interface des Hilfesystems entworfen (Abschnitt 3.1.2). Zuletzt wird auf die formative Nutzerstudie eingegangen, welche nach dem ersten Entwurf des Konzepts durchgeführt wurde (Abschnitt 3.1.3). Deren Ergebnisse wurden im zweiten Entwurf, welcher in den folgenden Kapiteln vorgestellt wird, bereits berücksichtigt.

### 3.1.1 Rollenmodell

Bei der Konzeption spielt eine große Rolle, wer für welche Tätigkeit verantwortlich ist. Um diese Fragen klären zu können, wird an dieser Stelle ein Rollenmodell für VizBoard eingeführt (Abbildung 3.1).

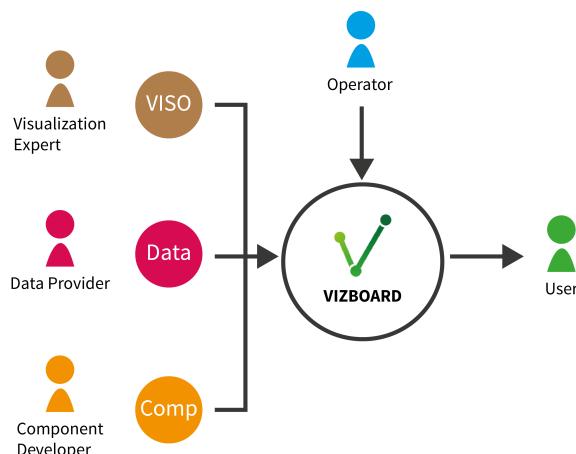


Abbildung 3.1: Rollenmodell von VizBoard

Das System VizBoard muss von jemandem betrieben werden. Der **Operator** bezahlt die Serverkosten und ist für Wartung und Instandhaltung von VizBoard verantwortlich. Für ihn ist wichtig, dass die anderen Parteien zufrieden sind, sodass sein System auch genutzt wird.

VizBoard greift auf die VISO zurück, eine Ontologie von Wissen über Informationsvisualisierung. Der **Visualization Expert** besitzt Expertenwissen in der Domäne der InfoVis und formalisiert sein es in der VISO, um VizBoard zu verbessern.

Ein wichtiger Teil von VizBoard sind die vorhandenen Daten. Der **Data Provider** stellt Datensätze in unterschiedlichen Formaten (z.B. APIs, RDF, OWL oder Excel Daten) zur Verfügung, die von Administratoren – also letztlich dem Betreiber – in VizBoard hochgeladen werden können.

Ebenso wichtig wie die Daten sind die Komponenten, die sie visualisieren. Sie werden von **Component Developers** geschrieben und bei VizBoard registriert. Komponentenentwickler können programmieren und kennen sich teilweise auch mit InfoVis aus.

Der **User** verwendet VizBoard, um Wissen in semantischen Datensätzen aufnehmen zu können. Er weiß weder über InfoVis noch über semantische Daten Bescheid und verfügt auch nicht über Programmierkenntnisse.

### 3.1.2 User Interface

Im Laufe der Konzeption werden zum besseren Verständnis einige User Interface Mockups entworfen. Diese basieren auf der Grundlage des Szenarios (Abschnitt 2.1). Die dort erwähnte Komposition enthält eine Karte, eine Treemap, eine Tabelle und ein Balkendiagramm (Abbildung 3.2). In den verwandten Arbeiten (Abschnitt 2.4) war die visuelle Konsistenz eine Designrichtlinie, welche das Bedürfnis nach Hilfestellungen verringern kann. Aus diesem Grund wird im folgenden ein Styleguide für das Hilfesystem beschrieben.



Abbildung 3.2: Grundlage der UI Mockups

Art

Das User Interface für die ausgewählte Hilfefunktion wird links oder rechts der Komponente in einer Sidebar dargestellt (Abbildung 3.3, 1). Gegenüber einem Popover

(2) hat sie den Vorteil, dass sie sowohl links als auch rechts funktioniert. Ein Popover kann nur über der Titelleiste angezeigt werden, ansonsten werden große Teile der Komponente verdeckt (3). Die Sidebar kann in der Größe variieren (4), da für die Kommentare vermutlich mehr Platz benötigt wird als beispielsweise für die Feedbackfunktion.

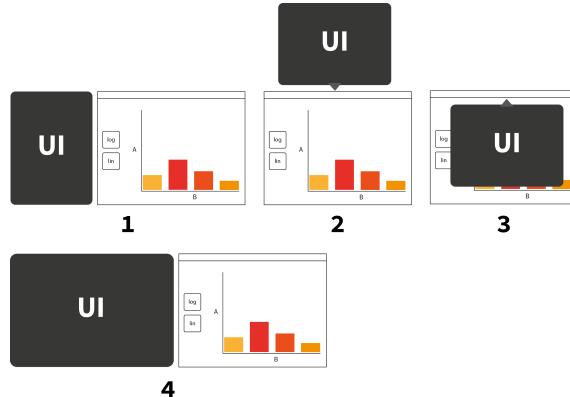


Abbildung 3.3: Verschiedene Möglichkeiten zur Platzierung der Hilfe UI

## Optik

Visuelle Konsistenz ist wichtig, damit der Benutzer Elemente gleichen Typs erkennen kann (Abschnitt 2.4). Deswegen hat der Bereich, in der Hilfefunktionen angezeigt werden, abgerundete Ecken. So hebt sich die UI für Hilfe von der normalen UI der Komponente, welche spitze Ecken aufweist, ab. Aus demselben Grund wird für die Hilfe UI ein dunkler Hintergrund gewählt, da der Hintergrund von VizBoard hell gehalten ist. Um etwas hervorzuheben, wird prinzipiell die Farbe Orange benutzt, da dies im bestehenden VizBoard Farbschema so festgelegt wurde. Eine Überlagerung der Komponente mit einem leichten Grau dient der Blickführung zu einem nicht überlagerten Element. Ist die Komponente hingegen mit einem tiefen Grau überlagert, signalisiert die dunkle Fläche eine »No Click Area«, wo keine Interaktionen ausgeführt werden können.

Offensichtlich sind die gewählten visuellen Effekte nicht mehr so nützlich, wenn eine Komponente einen dunklen Hintergrund aufweist. Schwarz mit Dunkelgrau zu überlagern macht für das menschliche Auge nur wenig Unterschied. Deswegen werden in diesem Fall dieselben Effekte mit Weiß und Hellgrau verwendet.

### 3.1.3 Nutzerstudie

Das entworfene Konzept wurde mit einer formativen Nutzerstudie evaluiert und danach entsprechend überarbeitet. Die wichtigste zu beantwortende Frage war dabei, ob die Kommunikationshilfe statisch (d.h. nach Klick auf einen Button) oder dynamisch (d.h. sobald eine Nachricht zwischen Komponenten ausgetauscht wurde) aufgerufen werden soll. Dazu wurde ein Paper Mockup [VSK96] erstellt (Abbildung 3.4). Er besteht aus einem Balkendiagramm, einer Karte und einer Tabelle. Als Datengrundlage wurden NBA Spieler gewählt. Die Balken des Balkendiagramms konnten ausgewählt werden, woraufhin sich die Tabelle und die Karte anpassten.

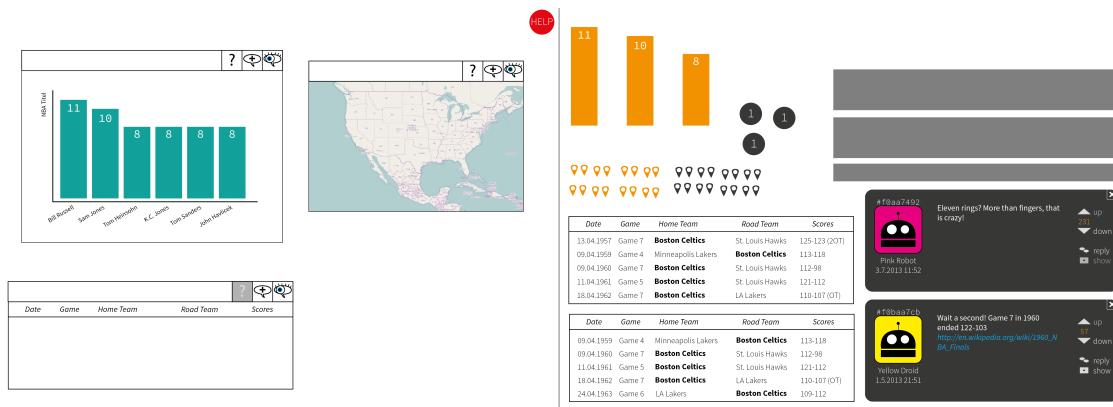


Abbildung 3.4: Die ersten zwei Seiten des Paper Mockups

## Teilnehmer

Die Nutzerstudie wurde mit fünf Teilnehmern zwischen 23 und 25 Jahren durchgeführt (Tabelle 3.1). Dabei waren 60 % der Personen männlich. Bei der Auswahl der Probanden wurde darauf geachtet, dass beide Geschlechter und auch andere Domänen außer Informatik vertreten sind.

Teilnehmer	Geschlecht	Alter	Domäne
Teilnehmer 1	M	25	Informatik
Teilnehmer 2	W	23	Zahnmedizin
Teilnehmer 3	M	23	Informatik
Teilnehmer 4	M	25	Informatik
Teilnehmer 5	W	25	Zahnmedizin

Tabelle 3.1: Teilnehmer der Nutzerstudie

## Methode

Zuerst wurde den Teilnehmern eine einführende Erklärung gegeben, wie sie später die Einführung in eine Komponente (Abschnitt 3.2) übernehmen soll. Dabei wurde der Typ des Diagramms erklärt (wenn notwendig) und die enthaltenen Daten erwähnt. Über die Kommunikation zwischen Komponenten wurde keine Aussage gemacht, einige Teilnehmer konnten allerdings einfach schließen, dass diese in irgendeiner Form stattfinden musste.

Nach der Einführung sollten die Testpersonen die Anwendung benutzen, wobei zwei mögliche Fragestellungen erwähnt wurden, denen sie nachgehen konnten. MouseOver-Interaktionen wurden explizit beschrieben, da sie im Paper Mockup nicht sichtbar sind. Die Probanden wurden gebeten, während der Nutzung ihre Gedanken laut mitzuteilen (»Thinking Aloud« Methode [SBS94]). Im Laufe des Tests wurden die Teilnehmer beispielsweise gefragt, warum sie bestimmte Dinge annahmen, ob die gezeigten Informationen ausreichen, welche Interaktionsmöglichkeiten sie in einer Situation sahen, welche Veränderungen sie erwarteten oder ob eine alternative Interaktion beziehungsweise ein verändertes User Interface besser geeignet wäre. Wenn sie eine Funktion nicht von selbst wählten (etwa einen Kommentar zu

verfassen), wurden sie darauf hingewiesen, sodass jeder Teilnehmer alle Funktionen benutzte.



Abbildung 3.5: Eine Teilnehmerin der Nutzerstudie

Um die Hauptfrage, ob Kommunikationshilfe (Abschnitt 3.5) statisch oder dynamisch präsentiert werden soll, zu klären, wurde die dynamische Variante drei der fünf Teilnehmer gezeigt. Die anderen zwei testeten die statische Kommunikationshilfe. Allen Teilnehmern wurde am Ende der Studie die andere Variante erklärt. Sie wurden dann gefragt, ob ihnen die Alternative mehr geholfen hätte oder besser verständlich gewesen wäre.

### **Einschränkungen**

Die gewählte Methode weist gewisse Einschränkungen auf, denen man sich bei der Auswertung bewusst sein sollte. Einerseits werden Teilnehmer bei »Thinking Aloud« doppelt belastet. Sie müssen sowohl ein unbekanntes User Interface benutzen, was mentalen Aufwand darstellt, als auch gleichzeitig ihre Gedanken formulieren. Das kann zu Inkonsistenzen in ihren Äußerungen führen.

Andererseits war der Mockup sehr klein, er beinhaltete nur drei Komponenten und zwei Kommunikationskanäle. Gerade letztere konnten die Teilnehmer auch ohne Hilfe verstehen, da sich der »Bildschirm« nur langsam und schrittweise änderte. Außerdem bezogen sich die Äußerungen und Vorschläge nur auf die vorgestellten Komponenten, ohne Alternativen zu berücksichtigen (beispielsweise Scatterplots, Parallel Coordinates etc.).

Zuletzt liegt es in der Natur des Paper Mockups, dass Interaktionen wie MouseOver, scrollen oder zoomen nur mit viel Aufwand umgesetzt werden können. Da dies im beschriebenen Mockup nicht der Fall war, konnte die Wirksamkeit dieser Interaktionen nicht überprüft werden.

### **Ergebnisse**

Die Bedienungshilfe (Abschnitt 3.3) wurde als ein Comic mit vier Panels vorgestellt, welche die Fragen »Welche Komponente?«, »Welche Elemente?«, »Welche Operation?« und »Was für ein Ergebnis?« beantworten sollten. Zwei der Teilnehmer konnten nicht sofort erkennen, dass die vier Bilder einen Comic und dieser wiederum eine Anleitung darstellte. Das sollte zusätzlich mit Text beschrieben werden. Außerdem

könnte ein ansprechenderes Layout der Panels dazu führen, dass sie mehr als Comic und weniger als vier unabhängige Bilder wahrgenommen werden. Einer der Teilnehmer merkte zusätzlich an, dass nicht ganz klar ist, ob der durch die Pfeile der statischen Kommunikationshilfe (Abschnitt 3.5) angedeutete Nachrichtenaustausch immer, nacheinander oder gleichzeitig stattfindet. Nachdem das für das Hilfesystem auch nicht eindeutig ist, kann in dieser Hinsicht leider nicht viel getan werden. Vier der fünf Teilnehmer waren sich einig, dass die letzten drei Panels des Comics auch ausreichen würden, da die Information in den ersten beiden redundant ist: Durch die Auswahl des Pfeils ist klar, um welche Komponenten es geht.

Prinzipiell waren zwei Varianten der Kommunikationshilfe (Abschnitt 3.5) denkbar: Einmal würde sie zeitgleich mit der Bedienungshilfe auf Anfrage des Benutzers angezeigt (statisch) und einmal angeboten, sobald ein Nachrichtenaustausch zwischen Komponenten stattfindet (dynamisch). Es gab keinen eindeutigen Sieger bezüglich der Frage, ob sie statisch oder dynamisch angeboten werden soll (Tabelle 3.2). Jeder Teilnehmer schätzte die jeweilige Variante der Hilfestellung als notwendig bzw. hilfreich ein (+) oder nicht (-), wobei ein Teilnehmer sich nicht festlegen wollte (0). Es stellte sich heraus, dass die Probanden sich uneins waren, welche Form die Hilfestellung haben sollte: Je ein Teilnehmer fand beide gut oder beide schlecht, zwei bevorzugten die statische Variante und einer die dynamische. Es ist zu beachten, dass diese Anmerkungen nach einer sehr übersichtlichen Situation im Mashup gemacht wurden. Wäre die Nutzerstudie mit mehr Komponenten durchgeführt worden, böte sich wahrscheinlich ein anderes Bild. Deswegen sollten in einer weiteren Nutzerstudie Grenzwerte für die Anzahl an Komponenten und Kommunikationskanälen bestimmt werden, bei denen Benutzer die Kommunikation nicht mehr ohne weiteres nachvollziehen können. Das Hilfesystem kann dann beim Start der kompositen Anwendung entscheiden, ob zusätzlich die dynamische Kommunikationshilfe eingesetzt werden muss.

Teilnehmer	Dynamisch	Statisch
Teilnehmer 1	+	+
Teilnehmer 2	-	+
Teilnehmer 3	-	+
Teilnehmer 4	+	0
Teilnehmer 5	-	-
Gesamt	-1	+2

Tabelle 3.2: Wertung der Kommunikationshilfe

Auffallend war außerdem, dass Teilnehmer, wenn sie über notwendige Ausführungsschritte sprachen, ein anderes Vokabular als die Kommunikationshilfe (Abschnitt 3.5) benutzten. Anstatt in Operationen (»hier auf einen Balken klicken«) sprachen sie in Aktionen (»hier einen Spieler auswählen«). Das lässt darauf schließen, dass das Hilfesystem ebenfalls in Aktionen sprechen sollte, wenn es die Zusammenhänge zwischen Komponenten erklärt, da es eher dem mentalen Modell der Benutzer entspricht (Abschnitt 2.3).

Ein Teilnehmer schlug vor, die verschiedenen Hilfestellungen zur Kommunikation zusätzlich in der ursprünglich für die Bedienung vorgesehenen Sidebar aufzulisten. Dieser Ansatz ist sowohl einfach als auch für Benutzer offensichtlich. Vermutlich

könnte diese Liste, welche dann alle Anleitungen zu Aktionen in der Komponente, alle Effekte von ausgehenden Nachrichten und alle Reaktionen auf eingehende Nachrichten enthielte, schnell sehr lang werden. Es ist zu befürchten, dass Benutzer in diesem Fall die Liste wegen zu viel dargestellter Information gar nicht mehr beachten würden. Deshalb wird dieser Vorschlag vorerst nicht umgesetzt.

Den Teilnehmern war nicht allein aufgrund der UI klar, dass Kommentare (Abschnitt 3.7) Annotationen enthalten können und wie diese hinzugefügt oder angezeigt werden. Die Kommentare sollten deswegen am Ende des Intros (Abschnitt 3.2) zusätzlich erklärt werden. Dasselbe Problem war bei der Bewertungsfunktion von Kommentaren zu beobachten. Die abstrakten, von StackOverflow<sup>1</sup> inspirierten Icons wurden auch von Informatikern teilweise mit »scrollen« anstatt »bewerten« assoziiert. Deswegen sollte noch einmal explizit mit Text beschrieben werden, dass sie eine Bewertungsfunktion darstellen und welche Semantik sich hinter ihnen verbirgt (»agree/disagree«). Falls die Funktionalität dann immer noch nicht verständlich ist, können die abstrakten Icons durch Daumen hoch/runter ersetzt werden. Eine weitere Erkenntnis dieses Sachverhalts ist, dass Scrollmöglichkeiten immer explizit dargestellt werden sollten. Das kann entweder in Form einer Scrollbar oder wie im Metro User Interface von Windows Phone 7 mit angeschnittenen Interaktionselementen passieren.

Von zwei Teilnehmern kam der Vorschlag, die Anzahl der Kommentare (Abschnitt 3.7) in einer Visualisierung ständig darzustellen und nicht über einen Button in der Titelleiste zu togeln. Hier ist aber zu beachten, dass die Komponenten im Paper Mockup sehr einfach gehalten waren und sehr eingeschränktes Interaktionspotential boten. Bei einem großen Scatterplot oder Parallel Coordinates ist diese Variante nicht mehr praktikabel. Aufgrund des Ziels der *Einheitlichkeit* sollten auch keine Ausnahmen gemacht werden.

Weitere Vorschläge, die hier den Rahmen sprengen würden und in weiteren Arbeiten behandelt werden können, beinhalten zusätzlich einen Mauszeiger im Comic zu animieren oder Vorschläge für mögliche Auswahlrechtecke bei den Annotationen anzubieten.

## 3.2 Intro in die Funktionalität einer Komponente

Aus dem Szenario (Abschnitt 2.1) und den verwandten Arbeiten (Abschnitt 2.4) wurde deutlich, dass dem Endnutzer zuerst ein Überblick über die verschiedenen ausgewählten Komponenten gegeben werden muss, sodass ihm die Struktur der kompositen InfoVis klar wird. Sie ist zusammen mit dem Verhalten eines Systems ein wichtiger Teil des mentalen Modells. Der Überblick beinhaltet folgende Punkte:

1. Art der Visualisierung, z. B. »Balkendiagramm«
2. Beschreibung der Visualisierung, z. B. »Ein Diagramm, das durch auf der x-Achse senkrecht stehende, nicht aneinander grenzende Rechtecke die Häufigkeitsverteilung einer diskreten Variable veranschaulicht.«
3. Informationen über enthaltene Daten, z. B. BIP auf der y-Achse und Länder auf der x-Achse.

<sup>1</sup><http://stackoverflow.com>

4. Links zu Tutorials, Videos und Erklärungen
5. Vor- und Nachteile der ausgewählten Visualisierung, z. B. »übersichtlich, wenn wenige Farben vorhanden sind«

Eine Möglichkeit – so ist die Beschreibung einer Komponente bereits umgesetzt – ist den Komponentenentwickler selbst einen Text dazu schreiben zu lassen. Er fügt diesen in die Komponentenbeschreibung ein, von wo das Hilfesystem ihn dann auslesen kann. Allerdings kann auf diese Weise kein *einheitliches* Vokabular bezüglich Informationsvisualisierungen sichergestellt werden. Ein einheitliches Vokabular ist aber wichtig, da unterschiedliche Bezeichnungen derselben Konzepte kontraproduktiv für das Verständnis sind. Deswegen werden die Informationen direkt zu den entsprechenden Konzepten in der VISO hinzugefügt. Von dieser zentralen Stelle kann das Hilfesystem sie dann auslesen und anzeigen, sodass die verwendeten Vokabeln konsistent bleiben. Außerdem kann eine einheitliche Beschreibung die Forderungen nach *Vollständigkeit* und *Korrektheit* erfüllen.

Nach Grammel [Gra12] können Links zu Tutorials und Erklärungen hilfreich sein, Endnutzern Visualisierungen zu erklären. Da externe Quellen aber nicht durch das Hilfesystem kontrolliert werden, kann es nicht sicherstellen, dass überall ein einheitliches Vokabular benutzt wird. Wie bereits erwähnt ist ein konsistentes Vokabular wichtig für effektive Hilfestellungen, weswegen dieser Punkt in dieser Form nicht umgesetzt werden kann. Was allerdings möglich ist, sind vom Operator erstellte Tutorialvideos in Form von vertonten Animationen, die Aufbau und Vor- bzw. Nachteile grafischer Repräsentationen erklären. Das sind zwar viele Videos, aber die Anzahl der Repräsentationen ist endlich und nicht alle sind gleich wichtig<sup>2</sup>, weswegen der nötige Aufwand gerechtfertigt scheint.

Grammel schreibt ebenfalls, dass InfoVis-Novizen die Vor- und Nachteile von verschiedenen Visualisierungstypen und visuellen Mappings lernen sollten [Gra12, S. 125]. Es wäre möglich, allgemeine Eigenschaften von grafischen Repräsentationen in der VISO zu beschreiben. Diese sind abhängig von den dargestellten Daten und den visuellen Mappings. Die Eigenschaft »Übersichtlichkeit« ist bei einem Balkendiagramm mit drei Balken gegeben (Vorteil), bei 15 Balken nicht (Nachteil). Ähnlich verhält es sich mit der Farbkodierung: Ein Mensch kann bis zu acht Farben in einer Visualisierung gut unterscheiden (Vorteil) [Maz09], aber mehr nicht (Nachteil). Das Hilfesystem müsste dem Benutzer demnach sowohl die unterschiedlichen Vor- und Nachteile als auch deren Bedingungen präsentieren, was sehr viel Information für eine Einführung darstellt. Im schlechtesten Fall kann der Benutzer auch keinen Bezug zur konkreten Visualisierung herstellen, weil keine der aufgeführten Bedingungen erfüllt ist. Er kann die aufgestellte Hypothese nicht am Beispiel überprüfen, was laut Grundlagen schlecht für seinen Verständnisprozess ist (Abschnitt 2.3.4). Hinzu kommt, dass diese Erklärung einen Schritt früher im VizBoard Workflow konzeptionell mehr Sinn macht. Dort wählt der Benutzer – von VizBoard unterstützt – die visuellen Mappings aus.

Aus den eben aufgeführten Gründen zeigt das Intro für eine Komponente folgende Informationen:

---

<sup>2</sup>Ein Balkendiagramm ist gängiger und deswegen leichter verständlich als eine Scatterplot Matrix.

1. Art der Visualisierung
2. Beschreibung
3. Enthaltene Daten
4. Tutorialvideo, wenn vorhanden

Davon ist Punkt 3 spezifisch für die Domäne der Informationsvisualisierung, da CRUISe Komponenten keine Daten visualisieren müssen. Die anderen Punkte sind für alle auf CRUISe basierenden Mashups hilfreich. Über die Interaktion mit der Visualisierung wird hier noch keine Aussage gemacht, da sie sehr umfangreich sein kann und den Rahmen des Intros sprengen würde. In Abschnitt 3.3 wird eine Hilfestellung zur Bedienung einer Komponente erarbeitet.

In der Nutzerstudie (Abschnitt 3.1.3) war die Kommentarfunktion nicht ersichtlich. Die Teilnehmer hatten Probleme sie aufzurufen, zu benutzen und die möglichen Funktionen zu erkennen. Deswegen sollte das Intro einer Komponente zusätzlich grundlegende Informationen über Kommentare (Abschnitt 3.7) und einen Hinweis auf die statische Meta-Hilfe (Abschnitt 3.9) enthalten.

### 3.2.1 User Interface und Interaktion

Nachdem der Benutzer bei der Visualisierung angekommen ist und die Komponenten sieht, wird der gesamte Viewport abgedunkelt und eine Sidebar neben der ersten Komponente (von links oben nach rechts unten) eingeblendet (Abbildung 3.6). Dort sind die im vorigen Abschnitt erläuterten Informationen zu finden: Die Art der Visualisierung (»Karte«), darunter eine Beschreibung einer Karte, Informationen zu den dargestellten Daten und zuletzt ein Tutorialvideo, wenn vorhanden. So werden die Fragen »Was ist das für eine Visualisierung?«, »Wie soll ich sie interpretieren?« und »Was zeigt sie für Daten?« beantwortet. Mit den Pfeilen links und rechts kann zwischen den Beschreibungen für verschiedene Komponenten gewechselt werden. Über die Meta-Hilfe (Abschnitt 3.9) kann das Intro wieder aufgerufen werden, wenn es geschlossen wurde.

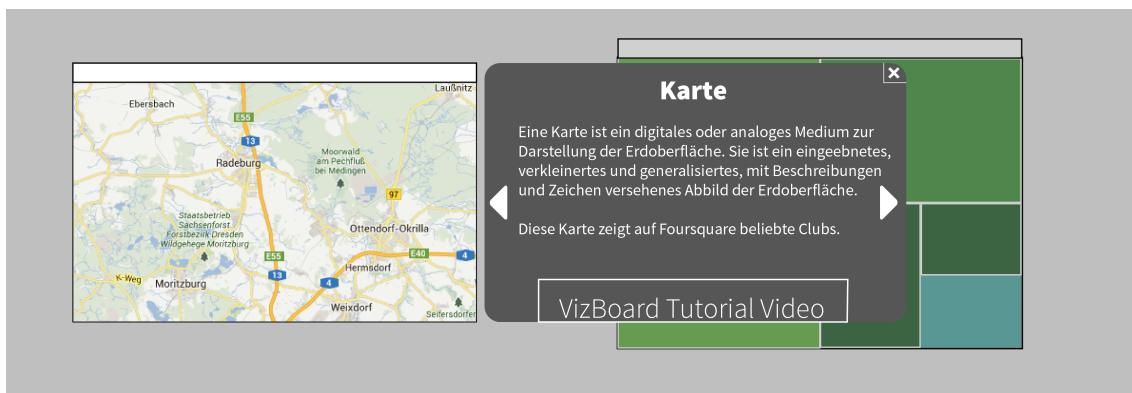


Abbildung 3.6: UI Mockup: Intro

### 3.2.2 Backend

Die VISO (Abschnitt 2.3.1) enthält bereits ein Vokabular für Informationsvisualisierungen. Deswegen ist es naheliegend, die Beschreibung der Visualisierung in die VISO hinzuzufügen. Die Art der grafischen Repräsentation muss der Komponentenentwickler bereits jetzt in der Komponentenbeschreibung hinzufügen. Das bedeutet das Hilfesystem kann die entsprechenden Daten auslesen. Die in der Visualisierung verwendeten Daten entsprechen dem Domain Assignment, welches der Komponente von der Laufzeitumgebung mitgeschickt wird. Das Domain Assignment ist das Mapping vom während der Integration erstellten generischen Datenschema auf Konzepte, die in einer externen Wissensbasis wie der DBpedia beschrieben werden [Voi+12].

Damit das Hilfesystem aber die URIs der aktuell visualisierten Datatype Properties bekommt, muss das CoRe erweitert werden. Aktuell gehen die URIs bei der Transformation in das generische Datenschema verloren, weil sie umbenannt werden. Um dem Komponentenentwickler Aufwand zu ersparen sollte deswegen das CoRe dahingehend erweitert werden, dass das ursprüngliche Mapping der MRE mitgeliefert wird.

## 3.3 Bedienung einer Komponente

In den verwandten Arbeiten (Abschnitt 2.4) wurde festgestellt, dass das Hilfesystem den Benutzer über das Verhalten der InfoVis aufklären muss. Da die Komponenten »Black Boxes« sind und von verschiedenen Entwicklern stammen, kann der Benutzer weder von verfügbaren möglichen Aktionen wie z. B. zoomen noch von einheitlicher Gestaltung der Interaktionen ausgehen. Beispielsweise wann eine Interaktion durch `MouseOver` anstatt `Click` ausgeführt wird. Zusätzlich sind die verschiedenen Komponenten wahrscheinlich visuell inkonsistent, einmal gelerntes kann nur schwer wiederverwendet werden. Deswegen ist eine Hilfestellung zur Bedienung notwendig.

Die einfachste Umsetzung der Hilfe zur Bedienung ist dem Komponentenentwickler eine Möglichkeit zu geben, Text und Bilder zu diesem Thema in die Komponentenbeschreibung einzufügen. Das hat allerdings den Nachteil, dass das Hilfesystem keinen Einfluss auf die Wortwahl, Gestaltung der Bilder oder Ausführlichkeit der Erklärung hat (Ziele der *Einheitlichkeit*, *Vollständigkeit*, *Korrektheit*, *Verständlichkeit*).

Hilfestellungen zur Bedienung würden vom Benutzer wahrscheinlich am besten verstanden, wenn sie tatsächlich vorhandene, konkrete Elemente der Komponente verwendete (»Semantic Transparency« [KK09]). Um zu demonstrieren, dass nach einem Klick die Tabelle sortiert sein wird, könnten die Zeilen derselben beispielhaft ihre Position wechseln. Assistance in dieser Form kann auf zwei Arten umgesetzt werden, die beide ihre Nachteile haben.

Einerseits könnten Aktionen mit Hilfe einer festgelegten Grammatik beschrieben werden. Beispielsweise würde die Aktion »sortieren« beinhalten, dass Elemente in Abhängigkeit eines Attributwerts ihre Position in der Visualisierung ändern. Dann könnte das Hilfesystem selbst die Tabellenzeilen demonstrativ sortieren, da es nun weiß, was die Aktion »sortieren« bedeutet. Der große Nachteil daran ist, dass es vermutlich immer einen Komponentenentwickler geben wird, der eine unvorgesehene Aktion in seiner Komponente umsetzen will und sie nicht beschreiben kann. Die

zweite Lösung, eine semantisch transparente Assistance umzusetzen, ist den Komponentenentwickler für jede Aktion eine Methode in seiner Komponente hinzufügen zu lassen, die diese demonstriert. Zur Laufzeit würde die Methode einfach vom Hilfesystem aufgerufen. Das hat allerdings den Nachteil, dass dem Komponentenentwickler erheblicher Mehraufwand aufgebürdet wird und keine Einheitlichkeit der Assistance garantiert werden kann. Aus diesen Gründen ist eine andere Lösung notwendig.

Wie in den Grundlagen (Abschnitt 2.3.4) gezeigt, können Comics eine effektive und attraktive Lösung für User Assistance sein. Allerdings stellt es für den Entwickler erheblichen Mehraufwand dar, für jede mögliche Aktion verschiedene Comicpanels zu erstellen. Davon abgesehen können (und sollen) die gestalterischen Fähigkeiten dazu nicht von ihm erwartet werden (*Minimalität*) und das Problem der Kontrolle über den Inhalt ist auch nicht gelöst (*Korrektheit*, *Vollständigkeit*, *Verständlichkeit*).

Aus diesem Grund werden die Comicpanels vom Hilfesystem automatisch erstellt. Es bleibt die Frage der Präsentation: Aus den Grundlagen geht hervor, dass eine multimodale Ausgabe (akustisch-verbal und visuell) zur Lerneffektivität beiträgt. Darauf wird aber aus zwei Gründen verzichtet. Einerseits sollen beide Kanäle gleichzeitig ausgegeben werden. Das ist problematisch, weil vier Panels darzustellen wesentlich schneller möglich ist als eine verbale Erklärung für den Vorgang zu sprechen. Die Darstellung der Panels müsste sich der Audioausgabe anpassen und würde wesentlich langsamer vonstatten gehen. Eine nichtfunktionale Anforderung an die Hilfestellung ist aber, dass der Benutzer die Hilfe schnell aufnehmen und verarbeiten können soll. Zweitens wird automatische Soundausgabe gemeinhin als schlechte Usability angesehen. Hauptsächlich weil der Nutzen von Screenreadern dadurch eingeschränkt wird, aber auch weil der Benutzer automatische Wiedergabe nicht kontrollieren kann und die Audiodaten möglicherweise unerwünscht Bandbreite beanspruchen.

### 3.3.1 API und Komponentenbeschreibung

Die VISO enthält im Namensraum `viso:activity` Konzepte der hierarchischen Aktivitätstheorie nach Kuutti [Kuu96]:

- **Aktivitäten** stellen langfristige Ziele des Benutzers dar, die nicht unmittelbar zu Ergebnissen führen (müssen). In Bezug auf das Szenario (Abschnitt 2.1) wäre die Aktivität von Anna »mehr über die geografische Verteilung verschiedener Genvariationen lernen«.
- **Aktionen** sind Teilziele der Aktivität. Im Szenario sind das »VizBoard aufrufen«, »Komponenten auswählen«, »Tabelle sortieren« etc.
- Aktionen bestehen wiederum aus einer Kette von **Operationen**, also beispielsweise Klicks oder Drags.

Für das Hilfesystem sind Operationen und Aktionen relevant, da Aktivitäten nicht beeinflusst werden können. Aktionen einer Komponente sind beispielsweise zoomen, suchen oder filtern; Operationen sind etwa Klicks, Drags oder Multitouch-Gesten.

Die CRUISe Komponentenbeschreibung gibt bereits Auskunft über die verfügbaren Aktionen in einer Komponente (Capabilities). Diese können entweder mit

(UI Capability) oder ohne Nutzerinteraktion (System Capability) ausgeführt werden. Für die Bedienungshilfe sind erster Linie UI Capabilities interessant. Allerdings wird sich bei der Hilfe zur Kommunikation (Abschnitt 3.5) zeigen, dass System Capabilities ebenfalls behandelt werden müssen. Im Folgenden werden zunächst UI Capabilities beschrieben.

## UI Capabilities

In der Komponentenbeschreibung fehlen Informationen darüber, welche Operationen auf welchen Elementen eine Aktion ausführen. Um dem beizukommen, werden Operationen aus der VISO (»welche Operationen«) zusammen mit CSS Selektoren (»welche Elemente«) in der Komponentenbeschreibung definiert. Aktionen können oft auf verschiedene Weise ausgeführt werden, haben also so gesehen eine Menge von äquivalenten Operationen. Wenn beispielsweise eine Tabelle sowohl durch einen Klick auf den Spaltenkopf als auch über ein separates Menü sortiert werden kann, werden diese Aktionen als äquivalent zusammengefasst. Sequentielle Operationen werden ebenfalls zusammengefasst, wobei die Reihenfolge im Markup die notwendige Ausführungsreihenfolge darstellt. So können auch zusammengesetzte Operationen, wie beispielsweise für ein HTML `select` Element nötig, abgebildet werden. Parallel Operationen werden nach demselben Prinzip zusammengefasst, allerdings ist hier die Reihenfolge irrelevant, da sie gleichzeitig ausgeführt werden müssen. Auf diese Weise können zusammengesetzte Operationen der Art STRG+T abgebildet werden.

Außerdem kann der Komponentenentwickler eine Wartezeit definieren, die bei der Generierung der Panels beachtet werden soll. Zum Beispiel ist das sinnvoll, wenn längere Animationen zu erwarten sind. Der Entwickler könnte UI Elemente innerhalb von fünf Sekunden anstatt 500 Millisekunden einblenden wollen. Zusätzlich zur Wartezeit ist es möglich, dass der Entwickler Zusatzinformationen in Stichworten angibt. Zum Beispiel könnte bei einem Balkendiagramm die Skala »verändert« (Aktion) werden. Dabei ist es wichtig zu wissen, welche Skala verwendet werden wird, z. B. linear oder logarithmisch. Der Inhalt dieser Zusatzinformationen kann statischer Freitext oder eine Referenz auf eine Datenvariable im Schema der Komponente sein, deren Label zur Laufzeit mit Hilfe des Mappings zwischen Daten und Schema der Komponente angezeigt würde. Nach demselben Prinzip sollten auch Zusatzinformationen über die VISO Operation angegeben werden können. Vor allem bei Tastaturkürzel (z. B. j/k für vor/zurück) macht es Sinn, da so nicht für jede Taste eine VISO Operation erstellt werden muss. Die erwähnten Zusätze zur Komponentenbeschreibung sollten allerdings optional sein, falls der Komponentenentwickler keine Hilfe für bestimmte Aktionen anbieten will.

## System Capabilities

System Capabilities unterscheiden sich von UI Capabilities dadurch, dass sie nicht durch Nutzerinteraktionen ausgeführt werden können. Sie werden an CRUISe Operationen annotiert, welche wiederum als Reaktion auf Events ausgeführt werden. Deswegen ist es nicht sinnvoll, hier VISO Operationen anzugeben. Stattdessen sollten in diesem Fall Beispieldaten aufgeführt werden, anhand derer die Operation

im Kontext des Testdatensatzes ordnungsgemäß ausgeführt werden kann. Andere Annotationen wie z. B. die Wartezeit können beibehalten werden.

## Zusammenfassung

Im folgenden werden die in diesem Abschnitt vorgenommenen Änderungen an der Komponentenbeschreibung noch einmal zusammengefasst:

- Zu den Aktionen werden VISO Operationen angegeben, welche diese ausführen. Diese können atomar, sequentiell oder parallel sein.
- Zu den VISO Operationen werden CSS Selektoren angegeben, um die relevanten UI Elemente zur Laufzeit finden zu können.
- Die VISO Operationen können Zusatzinformationen beinhalten, um sie genauer und verständlicher zu definieren. Ein Beispiel ist die Taste zur Operation »tippen«.
- Zu den Aktionen wird eine Anzahl von Sekunden angegeben, welche die Zeit bis zur finalen Änderung des User Interfaces angeben (beispielsweise wenn alle Animationen beendet sind).
- Zu den Aktionen können Zusatzinformationen in Form Freitext oder Referenzen auf Datenvariablen angegeben werden, um sie näher zu beschreiben.
- Falls die Aktion eine System Capability ist, werden Beispieldaten angegeben, mit denen eine CRUISe Operation ordnungsgemäß ausgeführt werden kann.

### 3.3.2 Generierung der Assistance

Die User Assistance zur Bedienung soll ein Comic werden, der per Definition wiederum aus mehreren Panels besteht, die Bilder und/oder Text enthalten [McC94]. Die Bilder der Panels können nicht a priori gezeichnet werden, da der Inhalt der Komponenten unbekannt ist (»Black Box« Prinzip). Aus diesem Grund müssen sie automatisch generiert werden. Die einfachste Möglichkeit dafür sind Screenshots der Komponente.

Dazu wird ein sogenannter »Headless Browser« eingesetzt. Das ist ein programmierbarer Browser ohne grafische Benutzeroberfläche, der z. B. Screenshots einer Seite abspeichern oder Testskripte ausführen kann. Beispiele sind unter anderem PhantomJS<sup>3</sup>, HTMLUnit<sup>4</sup> oder Ghost<sup>5</sup>.

Bei der Generierung der Panels ist zuerst das Component Repository beteiligt. Sobald eine Komponente hinzugefügt oder upgedatet wurde, parst das CoRe die entsprechende Komponentenbeschreibung und schickt eine Anfrage an den Hilfeservice. Dieser kann ins CoRe integriert sein, sollte dem »Separation of Concerns« Designprinzip folgend aber davon getrennt sein, da es schon für Matching, Ranking und Verwaltung der Komponenten zuständig ist. Der Hilfeservice rendert die

---

<sup>3</sup><http://phantomjs.org/>

<sup>4</sup><http://htmlunit.sourceforge.net/>

<sup>5</sup><http://jeanphix.me/Ghost.py/>

Komponente mit Hilfe des Headless Browsers in einer CRUISe Testumgebung. Sie besteht aus folgenden Komponenten:

- Laufzeitumgebung: Sie ist unter anderem für die Initialisierung und das Layout der Komponenten nötig.
- Data Repository: Die Komponente muss Daten darstellen, welche aus dem DaRe kommen können (aber nicht müssen).
- Component Repository: Das CoRe enthält alle verfügbaren Komponenten und stellt sie dem Hilfeservice zur Verfügung.

Es wurde eben erklärt, dass das DaRe die (Test-)Daten für eine Komponente liefern kann. Der Komponentenentwickler gibt in der Komponentenbeschreibung den gewünschten Datensatz und die zu visualisierenden Daten in Form einer SPARQL Query an, welche die Laufzeitumgebung bei der Generierung der Assistance der Komponente zur Verfügung stellt. Das kann ein »echter« Datensatz oder ein vom Operator zur Verfügung gestellter Testdatensatz sein. Es kann allerdings passieren, dass das DaRe keine passenden Daten enthält, weil die Komponente sehr spezielle Daten visualisiert (z. B. 3-Sterne-Hotels in Andorra). Dann sollte der Komponentenentwickler zusammen mit der Komponente Testdaten hochladen können oder eine URL angeben, unter der sie zu finden sind. Das ist zwar etwas Mehraufwand, aber es kann angenommen werden, dass im Laufe der Entwicklung einer Komponente sowieso Testdaten verwendet wurden. Sie müssen in den meisten Fällen also nicht mehr erstellt, sondern nur noch zur Verfügung gestellt werden. Die Bedienungshilfe kann für alle auf CRUISe basierenden Mashups hilfreich sein. Allerdings kann nicht davon ausgegangen werden, dass diese über ein Data Repository verfügen. Dann muss der Komponentenentwickler in jedem Fall Testdaten zur Generierung der Hilfe zur Verfügung stellen.

Nachdem die Komponente von der Laufzeitumgebung vollständig initialisiert und integriert wurde, führt der Hilfeservice folgende Schritte aus:

1. Für jede verfügbare UI Capability:
  - a) Für alle äquivalenten Operationen:
    - i. Für jede Operation:
      - Finde die Bounding Box der betroffenen Elemente und speichere sie
      - Hebe betroffene Elemente hervor
      - Erstelle einen Screenshot der Komponente
      - Setze Zustand der betroffenen Elemente zurück
  - b) Führe die Aktion aus
  - c) Warte die definierte Anzahl an Sekunden
  - d) Erstelle einen Screenshot vom Ergebnis der Aktion

Falls die Operation in Schritt 2.a.i sequentiell sein sollte, muss der beschriebene Algorithmus die Bounding Box der Elemente jeder Teiloperation finden, genauso bei parallelen Operationen. Für System Capabilities wird derselbe Algorithmus verwendet, allerdings entfällt der Teil mit VISO Operationen. Stattdessen wird einfach die CRUISe Operation ausgeführt.

1. Für jede verfügbare System Capability:
  - a) Führe die CRUISe Operation mit Beispieldaten aus
  - b) Warte die definierte Anzahl an Sekunden
  - c) Erstelle einen Screenshot vom Ergebnis der Aktion

Nachdem alle nötigen Screenshots der Komponente erstellt wurden, muss das Hilfesystem eine Kategorisierung in »helle Komponente« und »dunkle Komponente« vornehmen. Das ist nötig, damit das Hilfesystem zur Laufzeit weiß, ob es dunkle oder helle visuelle Effekte einblenden soll. Eine naive Möglichkeit dazu wäre mit Hilfe einer Bildverarbeitungsbibliothek wie OpenCV<sup>6</sup> die Screenshots zu binärisieren, also in Bilder aus schwarzen und weißen Pixeln umzuwandeln. Enthält die Mehrzahl der Bilder mehr schwarze als weiße Pixel, wird die Komponente als »dunkel« eingestuft, ansonsten als »hell«.

Die Bilder der Comics können auf unterschiedliche Weise ans Hilfesystem ausgeliefert werden. Eine Möglichkeit ist die Bilder nach Generierung in der Komponentenbeschreibung zu definieren, so wie es bereits mit einem Screenshot der Komponente gemacht wird. Ebenso könnte ein Webservice die Bilder über eine REST API anbieten, von wo das Hilfesystem sie zur Laufzeit lädt. Dieser Webservice kann eigenständig oder im CoRe integriert sein.

## Einschränkungen

Die vorgestellte Lösung hat auch Nachteile. Zunächst ist sie vom Document Object Model (DOM) abhängig, was eine automatische Hilfegenerierung für Komponenten ausschließt, die auf Flash, Silverlight oder ähnlichen Technologien basieren. Der Trend geht aber ohnehin zu HTML/SVG-basierten Visualisierungen (z. B. mit D3 [BOH11] oder Raphaël<sup>7</sup>): Flash ist laut Adobe hauptsächlich für Videoauslieferung und Spiele gedacht [Ado13] und Microsoft zieht HTML5 Silverlight vor [Fol10]. Deswegen sollte dieser Nachteil nicht ins Gewicht fallen.

Eine weitere Einschränkung ist, dass zeitabhängige Interaktionen wie zum Beispiel »MouseOver für 3 Sekunden« nicht modelliert werden können. Diese sind im Web aber ohnehin eher unüblich und werden als schlechte Usability betrachtet, weil sie nicht sichtbar und deswegen schwer zu entdecken sind.

### 3.3.3 User Interface und Interaktion

Die Hilfestellung zur Bedienung kann über einen ?-Button in der Titelleiste aufgerufen werden, den der Benutzer vielleicht von nativen Desktop-Applikationen kennt

<sup>6</sup><http://opencv.willowgarage.com/wiki/>

<sup>7</sup><http://raphaeljs.com/>

(Abbildung 3.7). Der Viewport der Komponente wird abgedunkelt und nur die Elemente, für die auch Hilfestellungen existieren, bleiben sichtbar. Neben der Komponente erscheint eine Sidebar, welche die verfügbaren Aktionen noch einmal auflistet. Die Assistance wird entweder nach einem Klick auf ein sichtbares UI Element oder ein Element der Liste aufgerufen. So kann der Benutzer sowohl die Frage »Was macht dieser Button?« als auch »Wie kann ich Aktion XY ausführen?« beantworten. Eine zweite Möglichkeit, wie die Assistance aufgerufen werden kann, ist über einen ?-Button am Interaktionselement selbst (Abschnitt 3.6).

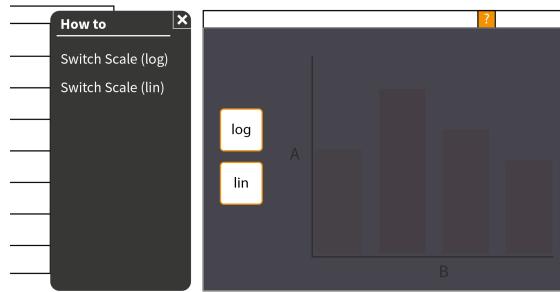


Abbildung 3.7: UI Mockup: Bedienung (Schritt 1)

Danach wird die kürzeste Operationsfolge für die ausgewählte Aktion erklärt. Es werden schrittweise mindestens drei Panels in der Komponente angezeigt (Abbildung 3.8). Der Inhalt der Panels ist in Abbildung 3.9 zu sehen. Das erste Panel startet mit der ganzen Komponente, zoomt dann aber auf das erste relevante UI Element. Panel 2 startet wiederum mit dem Inhalt von Panel 1, überblendet dann aber eine notwendige Operation. Das dritte Panel startet mit dem Inhalt aus Panel 1, zoomt aber auf eine Gesamtansicht wie anfangs hinaus, sodass der Benutzer den Unterschied sehen kann. Sollte eine Aktion aus mehreren zusammenhängenden Operationen bestehen (sequentielle VISO Operation), wiederholen sich die Panels 1 und 2 dementsprechend. Falls mehrere gleichwertige Operationen (parallele VISO Operation) eine Aktion ausführen, so können die Panels 1 und 2 gescrollt werden.

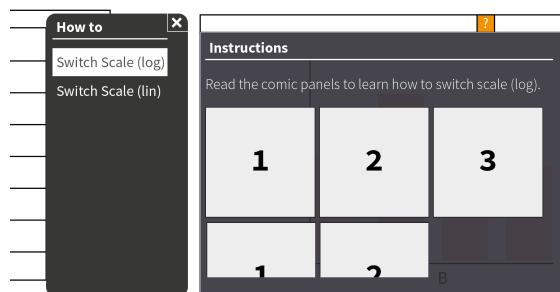


Abbildung 3.8: UI Mockup: Bedienung (Schritt 2)

Die Panels haben eine festgelegte Mindestgröße, um lesbar zu bleiben. Heer et al. [Hee+08] verwenden in ihrer History, die ebenfalls Screenshots enthält, eine Seitenlänge von 120 Pixel. Falls die Komponente nicht breit genug sein sollte, um die nötige Anzahl von 120 Pixel großen Panels in einer Zeile anzuzeigen, wird die Be-

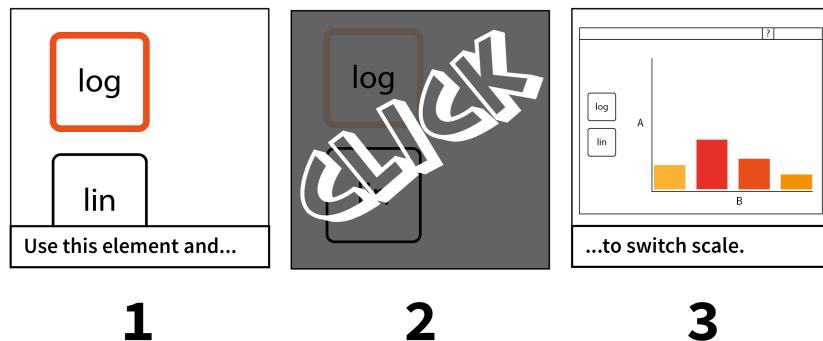


Abbildung 3.9: UI Mockup: Comic

dienungshilfe in einem Fenster angezeigt. So können parallele Operationen immer gescrollt werden.

Die Texte in den Panels werden zur Laufzeit hinzugefügt, da so beispielsweise auf die Sprache des Nutzers reagiert werden kann. Ansonsten müsste der Hilfeservice für jede unterstützte Sprache ein eigenes Bild erzeugen. Die dafür notwendigen Informationen sind:

- Panel 1: Welche Elemente beteiligt sind.
- Panel 2: Name der Operation.
- Panel 3: Name der Aktion.

Diese werden aber schon im Schritt davor (Abbildung 3.8) benötigt, müssen also sowieso bekannt sein.

### 3.4 Feedback zu einer Komponente

Wie bei jeder Software können bei den Komponenten in VizBoard Fehler auftreten. Externe APIs, die sie benutzen, wie zum Beispiel von Facebook oder Twitter können sich ändern. Visualisierte Daten können Fehler in der Komponente verursachen, beispielsweise durch nicht überprüfte `null` Werte oder wenn sie ein unerwartetes Format haben. Ebenso ist es möglich, dass VizBoard selbst Bugs enthält. Schlimmstenfalls fallen diese Fehler erst dem Endnutzer auf, welcher sie dann melden können soll.

Für diejenigen, die sich mit dem Bug Report befassen müssen – also Komponentenentwickler bzw. Operator – sind möglichst genaue Angaben sehr wichtig. Welcher Browser wurde benutzt? Welche Version? Welches Betriebssystem? Was genau hat nicht funktioniert? Ist der Fehler reproduzierbar? Wenn ja, wie? Diese und andere Fragen müssen beantwortet werden, um letzten Endes die Fehlerquelle zu finden. Liegt es an der Komponente (z. B. wegen veralteten API Anfragen), an VizBoard (z. B. ein Bug im Event Broker) oder keinen der beiden (z. B. fehlerhafte Daten von der externen API)? Aus Sicht des Endnutzers »funktioniert es nicht«, da er keine Kenntnisse der Programmierung besitzt und deswegen keine sinnvollen Hypothesen aufstellen, überprüfen und Schlussfolgerungen ziehen kann. Deswegen sollte

der Endnutzer einen möglichst ausführlichen Text verfassen, um das Problem zu beschreiben, und die Möglichkeit haben, die Komponente prinzipiell als gut oder schlecht zu bewerten. Letztere Funktion existiert bereits in VizBoard und wird in die Feedback UI integriert. Die anderen Informationen, wie eben beispielsweise die Browerversion, sollten automatisch gesammelt und mitgeschickt werden.

### 3.4.1 User Interface und Interaktion

Der Nutzer klickt auf einen Button in der Titelleiste, beschreibt sein Problem und sendet ab (Abbildung 3.10). Optional kann er außerdem eine Bewertung abgeben [VFM13].



Abbildung 3.10: UI Mockup: Feedback-Funktion

### 3.4.2 Backend

Informationen über den Browser befinden sich im DOM Objekt `window.navigator` und beinhalten unter anderem:

- User Agent String, welcher Betriebssystem, Browser und deren Versionen enthält
- Sprache des Browsers
- Ob Cookies aktiviert sind
- Installierte Plugins, wie Java oder Flash

Diese und andere notwendige Informationen können mit Javascript einfach ausgelesen werden. Noch mehr Informationen über den Nutzungskontext können über den Context Service (Abschnitt 2.3.1) bezogen werden. Dieser beinhaltet u. a. den Ort oder das Geschlecht des Nutzers.

Das Feedback des Nutzers wird ans Rating Repository gesendet und gespeichert. Die gesammelten Informationen aus dem Context Service über Nutzer, Browser und Anwendungskontext werden im RaRe referenziert. Das RaRe könnte aus der vom Nutzer abgegebenen Bewertung und Textklassifikation [Seb02] in »positiv« oder »negativ« bestimmen, ob ein Ticket in einem Support System (wie z. B. OSTicket<sup>8</sup>) des VizBoard-Betreibers erstellt werden soll. Die positiven Kommentare könnten beispielsweise für Testimonials auf der Startseite von VizBoard verwendet werden.

<sup>8</sup><http://osticket.com/>

## 3.5 Kommunikation zwischen Komponenten

Im kompositen Infomationsvisualisierungssystem VizBoard kommunizieren Komponenten miteinander, indem sie Nachrichten austauschen (Abschnitt 2.3.1). Dieser Vorgang ist für den Benutzer nicht sichtbar und kann für Verwirrung sorgen, wenn andere Komponenten auf Interaktionen reagieren, die dort nicht getätigten wurden. Dem wird beigekommen, indem

1. die Kommunikation zwischen Komponenten sichtbar gemacht wird und
2. die Abhängigkeiten zwischen Komponenten mit Comics erklärt werden, genau wie bei der Bedienung (Abschnitt 3.3).

### 3.5.1 User Interface und Interaktion

Wie im vorigen Abschnitt erläutert, wird zuerst der Nachrichtenaustausch zwischen Komponenten sichtbar gemacht (Abbildung 3.11). Dazu wird ein Pfeil vom Sender der Nachricht zum Empfänger animiert und gezeichnet, der in der Mitte ein Fragezeichen enthält. Das Fragezeichen wurde gewählt, da es konsistent mit dem Rest der Hilfe ist und ein Brief in der Nutzerstudie Assoziationen mit »Email« oder »in sozialen Netzwerken teilen« weckte. Sollten mehrere Empfänger vorhanden sein, existiert für jeden davon ein Fragezeichen. Die Kommunikationshilfe wird gleichzeitig mit der Bedienungshilfe (Abschnitt 3.3) aufgerufen, solange wenig genug Komponenten im Mashup vorhanden sind (Abschnitt 3.1.3). Ansonsten wird zusätzlich bei jeder Nachricht ein entsprechender Pfeil eingeblendet und nach kurzer Zeit (ca. 200–500 ms) wieder ausgeblendet.

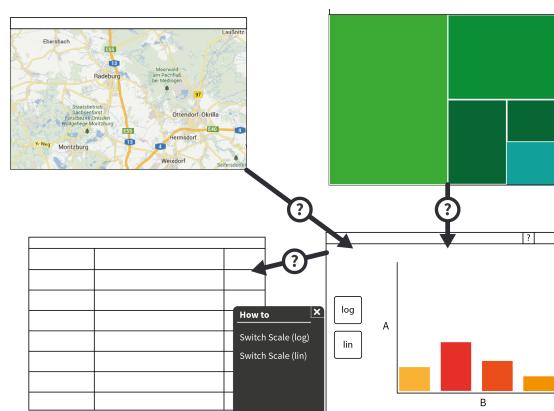


Abbildung 3.11: UI Mockup: Kommunikationshilfe

Wenn eine Aktion ausgewählt wird, wird die Anzahl der gezeigten Pfeile entsprechend eingegrenzt (Abbildung 3.12).

Die Abhängigkeiten zwischen den beiden Komponenten werden erklärt, nachdem auf das Fragezeichen geklickt wurde (Abbildung 3.13). Ein Fenster wird geöffnet, welches fast den gleichen Comic wie in Abschnitt 3.3 enthält. Der einzige Unterschied ist, dass anstatt einer VISO Operation eine VISO Aktion genannt wird. Auf diese Weise wird die Frage beantwortet, wie sich die angezeigten Komponenten beeinflussen und was Aktionen in anderen Komponenten bewirken.

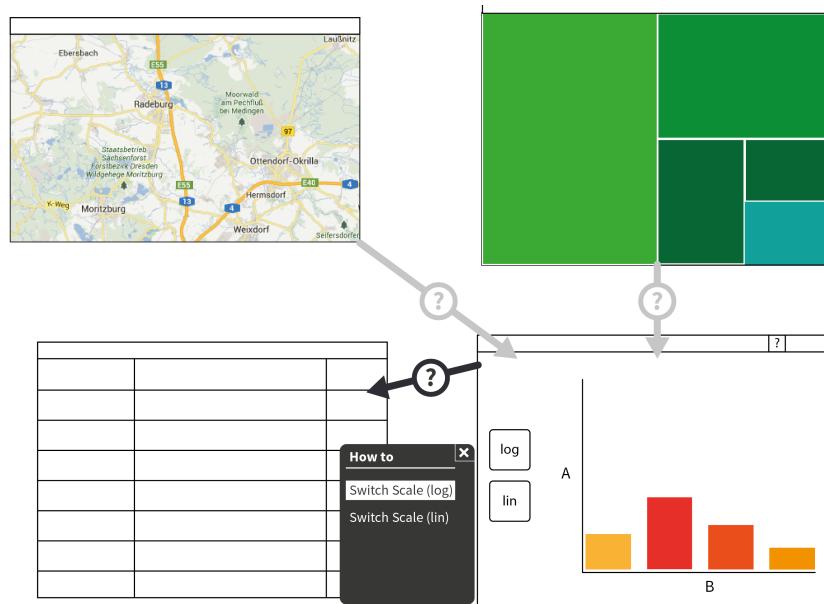


Abbildung 3.12: UI Mockup: Kommunikationshilfe mit eingeschränkten Optionen

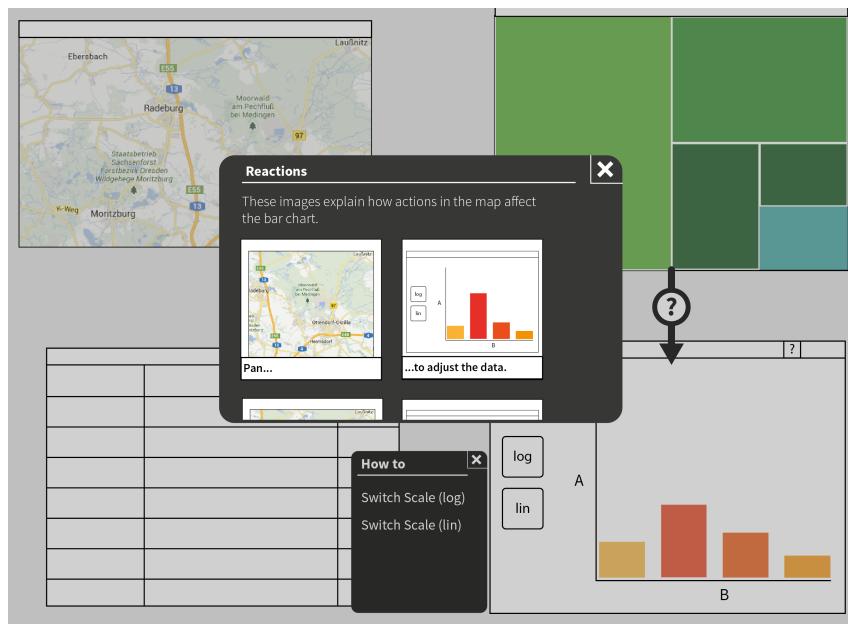


Abbildung 3.13: UI Mockup: Comic erklärt Kommunikation. Verändern des Kartenausschnittes (Aktion: Panning, Operation: Drag) bewirkt ein Update im Balkendiagramm.

### 3.5.2 Backend

Die Kommunikation zwischen Komponenten kann über den Event Broker (Abschnitt 2.3.1) nachvollzogen werden, da dieser zentraler Austauschpunkt für Nachrichten ist. Das Hilfesystem zeichnet Pfeile, wenn es vom Event Broker über Nachrichten informiert wird und die dynamische Kommunikationshilfe aktiviert ist.

Für die Generierung des Comics muss das Konzept der User Assistance für Bedienung (Abschnitt 3.3.1) nur leicht erweitert werden. Da Komponenten mit Operationen auf Nachrichten reagieren (Abschnitt 2.3.1), muss für das Ergebnis einer Operation ebenfalls ein Screenshot erstellt werden. Diese sind jetzt schon in der Komponentenbeschreibung vorhanden und können während der Panel-Generierung einfach ausgeführt werden. Sollten die Operationen Parameter erwarten, muss der Komponentenentwickler Beispieldaten in die Komponentenbeschreibung einfügen.

Zu beachten ist allerdings, dass es Operationen geben kann, welche das User Interface der Komponente nicht verändern. Das Konzept der Hilfestellung mit Comics fußt darauf, dass ein sichtbarer Unterschied zwischen den Zuständen vor und nach einer Interaktion bzw. Operation besteht. Ist jener nicht vorhanden, ergibt der Comic keinen Sinn mehr. Deswegen muss dem Komponentenentwickler eine Möglichkeit gegeben werden, Operationen zu identifizieren, für die keine Hilfe generiert werden soll. Das kann am einfachsten durch ein optionales Attribut in der Komponentenbeschreibung umgesetzt werden.

## 3.6 Verlinkung von unbekannten Konzepten

Ein Punkt, der in den verwandten Arbeiten (Abschnitt 2.4) kaum behandelt wurde, ist dass der Benutzer mehr Informationen über die visualisierten Daten benötigen könnte. Er könnte auf vollständig (»Was ist ein Säurezeiger?<sup>9</sup>«) oder teilweise (»Wie ist das BIP definiert?<sup>10</sup>«) unbekannte Begriffe treffen. Um das Verständnis der Daten zu fördern, müssen diese erklärt werden, weswegen sie mit einer externen Wissensbasis verlinkt sein sollen. Zu erklärende Begriffe werden vermutlich hauptsächlich in Legenden von Visualisierungen zu finden sein, können aber auch in Freitexten oder in Interaktionselementen vorkommen.

### 3.6.1 Markup und Backend

Um zu erklärende Begriffe für das Hilfesystem auffindbar zu machen, zeichnet der Komponentenentwickler sie mit einem dafür vorgesehenen Datenattribut aus: Eines für Legenden, eines für Freitext. Diese können entweder in der Komponentenbeschreibung angegeben und so selbst gewählt werden, oder sie werden vom Vizboard-Betreiber festgelegt. Hier wird die letzte Variante umgesetzt, weil sie dem Komponentenentwickler etwas Aufwand erspart und »Kollisionen« mit bereits vorhandenen Attributen erstens unwahrscheinlich sind und zweitens einfach behoben werden können (beispielsweise mit dem Präfix `my-`). Der zu erklärende Begriff ist dann der

<sup>9</sup>Eine Pflanze, die nur auf Böden mit einem bestimmten ph-Wert wachsen kann.

<sup>10</sup>Der Gesamtwert aller Waren und Dienstleistungen, die innerhalb eines Jahres in einer Volkswirtschaft hergestellt wurden und dem Endverbrauch dienen.

Textinhalt des Elements mit dem entsprechenden Datenattribut. Bei Interaktionselementen sind die CSS Selektoren für die Bedienungshilfe (Abschnitt 3.3) schon vorhanden und das Konzept wird in Form einer URI in die Komponentenbeschreibung eingefügt.

Auf diese Weise können die erklärungsbedürftigen Konzepte über CSS Selektoren gefunden werden. Allerdings stellt sich die Frage, wie das Hilfesystem die Beschreibung lesen kann. Die Konzepte sind im Falle von Freitext dynamisch, bei Legenden statisch, aber abhängig vom Datensatz und bei Interaktionselementen statisch in Abhängigkeit der Komponente. Es erscheint sinnvoll eine einheitliche Zugriffsschicht für Konzeptbeschreibungen zu etablieren. Dabei bietet sich im Fall von VizBoard das DaRe an, welches als einheitliche Zugriffsschicht auf Daten konzipiert ist. Bei anderen auf CRUISe aufbauenden Mashups sollte das CoRe diese Funktion übernehmen.

Das DaRe findet das Domain Assignment eigenständig, nachdem ein Datensatz hochgeladen wurde. Danach sollte eine Beschreibung aus der externen Wissensbasis abgerufen und gespeichert werden. Das hat den Nachteil, dass die Informationen zum Zeitpunkt der Anzeige möglicherweise nicht mehr aktuell sind. Da sich aber Beschreibungen zu Konzepten wie »logarithmische Skala« kaum ändern sollten und als Beispiel die DBpedia nur ein- bis zweimal pro Jahr ein Update bekommt<sup>11</sup>, sollte das kein Problem sein. Im Gegenzug ist das Hilfesystem unabhängig von der DBpedia verfügbar und schneller, da im Vergleich zum dynamischen Abruf der Beschreibung ein Network Round Trip eingespart wird.

Im Fall von Interaktionselementen parst das CoRe die Komponentenbeschreibung, nachdem eine Komponente hochgeladen wurde. Dabei extrahiert es die notwendigen Konzepte und schickt diese ans DaRe, welches die Beschreibungen dazu abruft und speichert.

### 3.6.2 User Interface und Interaktion

Bei Legenden und Freitext wird hinter die entsprechend markierten DOM Elemente (siehe Abschnitt 3.6.1) ein Fragezeichen eingefügt, um eine Interaktionsmöglichkeit anzudeuten. Dieses kann angeklickt werden, woraufhin eine Sidebar mit einer kurzen Erklärung des Konzepts angezeigt wird (Abbildung 3.14), bei Interaktionselementen wird zusätzlich die Bedienungshilfe (Abschnitt 3.3) angezeigt. Die Sidebar enthält auch einen Link, der zu einer Erklärung in einer externen Wissensbasis führt. Zur Erklärung des Konzepts gehört auf jeden Fall eine kurze Beschreibung, wie sie im ersten Absatz von Wikipedia-Artikeln gegeben ist und idealerweise ein Bild, falls vorhanden. Ist keine Beschreibung in der Sprache des Benutzers vorhanden, wird sie auf englisch angezeigt.

Im Fall von Interaktionselementen wird bei MouseOver derselbe Hilfebutton eingeblendet (Abbildung 3.15). Das hat den Grund, dass unbekannt ist, wie die Elemente angeordnet sind. Eine ständige Anzeige des Hilfebuttons könnte andere Elemente verdecken. Nachdem der Benutzer den Button gedrückt hat, werden falls vorhanden sowohl die Bedienungshilfe (Abschnitt 3.3) als auch die ausgehende Kommunikation (Abschnitt 3.5) bei der betreffenden Aktion eingeblendet.

<sup>11</sup><http://blog.dbpedia.org/category/dataset-releases/> zeigt ein bis zwei Artikel im Jahr, abgerufen am 01.07.2013.

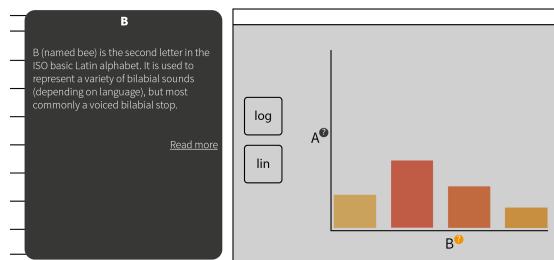


Abbildung 3.14: UI Mockup: Verlinkung von Legenden



Abbildung 3.15: UI Mockup: Verlinkung von Interaktionselementen

Zwar wäre es wünschenswert, sowohl Legenden als auch Interaktionselemente mit derselben UI kennzeichnen zu können. Das ist leider nicht ohne weiteres möglich. Würden Legenden und Freitext die UI der Interaktionselemente verwenden, wäre die Hilfefunktion nicht sichtbar. Wäre es andersherum, müsste der DOM auf nicht-triviale Weise manipuliert werden (Einfügen von eigenen Elternknoten usw.), was unerwünschte Auswirkungen auf CSS Selektoren des Entwicklers haben könnte.

## 3.7 Kommentare in Visualisierungen

Kommentare waren in den verwandten Arbeiten (Abschnitt 2.4) eine populäre Variante, um das Verständnis der Daten zu fördern. Mit Hilfe von Kommentaren können Endnutzer andere auf beobachtete Fakten der Daten wie Ausreißer oder Trends hinweisen und so zum Verständnis beitragen. Der Begriff »Kommentartext« referenziert hier den Text, welcher von Benutzern geschrieben wird, und »Annotation« die Markierung in der Visualisierung. Ein »Kommentar« ist die Kombination aus beliebigen Annotationen und einem Kommentartext.

Prinzipiell gibt es zwei Varianten, wie Kommentare in einer InfoVis-Komponente umgesetzt werden können. In der ersten kümmert sich die Komponente darum, wie und wann sie Kommentare lädt und anzeigt. Der Vorteil daran ist, dass die Handhabung von Kommentaren speziell auf jede Komponente angepasst wird. Allerdings kann so kein *einheitliches* Look & Feel gewährleistet werden. Zwar könnten Designrichtlinien Aussehen und Interaktionen vorschreiben, sie sind aber zwecklos wenn der Komponentenentwickler sie nicht einhält. Außerdem erhöht diese Variante den Entwicklungsaufwand der Komponente stark (gegenläufig zum Ziel der *Minimalität*).

In der zweiten Variante ist es Aufgabe des Hilfesystems, Kommentare zu verwalten und anzuzeigen. Die vorhin genannten Nachteile werden hier zu Vorteilen: Einheitliches Look & Feel wird sichergestellt und der Komponentenentwickler hat keinen oder kaum zusätzlichen Entwicklungsaufwand. Allerdings muss das User Interface

für Kommentare sich verschiedenen Interaktionen der Komponente wie z. B. scrollen, Tab wechseln, zoomen etc. anpassen, ohne darüber informiert zu werden oder über die konkrete Darstellung der Daten Bescheid zu wissen (»Black Box« Prinzip von Komponenten). Im folgenden wird erläutert, wie diese Nachteile umgangen werden können.

### 3.7.1 Features

In diesem Abschnitt wird diskutiert, welche Funktionen das Kommentarsystem zur Verfügung stellen soll.

#### URL einfügen

Es ist wahrscheinlich, dass Benutzer zur Erklärung der Daten bzw. Visualisierung auf externe Quellen verweisen müssen. Da VizBoard ohnehin webbasiert ist, eignen sich Links in Form von URLs dafür sehr gut.

#### Kommentar referenzieren

Um Diskussionen und Konversationen zur Visualisierung zu fördern, müssen Kommentare einander referenzieren können [HVW07]. So wird es Benutzern erleichtert, zusammen eine Erklärung für Ausreißer oder Trends zu finden.

#### Voting

Erfahrungsgemäß tragen nicht alle Kommentare gleich viel zur Diskussion bei. Man denke dabei an Hinweise auf Tippfehler oder das berüchtigte »Erster!«. Deswegen ist es notwendig, dass Benutzer Kommentare anderer bewerten können.

#### Ranking

Da Diskussionen und Konversationen nicht durch das User Interface eingeschränkt werden sollen, ist es nötig, Kommentare nach Datum sortieren zu können. So kann der Gesprächsverlauf nachvollzogen werden. Wenn Benutzer aber nicht an der Konversation teilnehmen, sondern schnell eine Antwort auf ihre Fragen haben wollen, bietet es sich an, nach Bewertung sortieren zu können.

#### Kommentar editieren

Den eigenen Kommentar editieren zu können, ist aus verschiedenen Gründen wichtig, z. B. wenn der Benutzer Tippfehler ausbessern oder ein paar Absätze zur Lesbarkeit einfügen will. Wichtig ist für andere, dass dies transparent geschieht und die Diskussion nachvollziehbar bleibt. Es gibt verschiedene Lösungsmöglichkeiten:

- Der Kommentar kann nicht editiert werden. So bleibt alles perfekt nachvollziehbar, aber der Benutzer kann nichts ausbessern.
- Der Kommentar kann nur für  $x$  Minuten editiert werden, danach ist er gesperrt. Diese Variante erlaubt immerhin noch Tippfehler auszubessern, aber der Benutzer kann beispielsweise nicht nach einer Stunde eine Quelle einfügen.

- Der Kommentar kann beliebig editiert werden. Diese Variante ist aus Nutzersicht am Besten, aber sie muss anderen gegenüber transparent gemacht werden. Es muss sichtbar sein, dass ein Kommentar editiert wurde und wann. Im Idealfall würde auch die Originalversion eines referenzierten Kommentars mit abgespeichert, sodass diese später eingeblendet werden kann.

### Kommentar löschen

Gründe, um einen Kommentar zu löschen, beinhalten unter anderem Beschimpfungen oder mangelnde Aktualität, etwa wenn eine zitierte Studie widerlegt wurde. Wie beim Editieren von Kommentaren ist es hier notwendig, dass der Gesprächsverlauf nachvollziehbar bleibt. Die Lösungsansätze beinhalten:

- Kommentare können nicht gelöscht werden. Das könnte Benutzer davon abhalten, Kommentare zu schreiben, was auch nicht Sinn der Sache ist.
- Kommentare können gelöscht werden, solange keine Referenz darauf besteht. So kann ein Diskussionsverlauf konsistent bleiben, allerdings ist dieser Weg aus Nutzersicht schwer verständlich.
- Kommentare können beliebig gelöscht werden. Diese Variante weckt keine Bedenken, den eigenen Kommentar nicht mehr löschen zu können. Es muss aber etwas mit den Referenzen auf gelöschte Kommentare geschehen, da ansonsten die Nachvollziehbarkeit der Diskussion leidet. Beispielsweise könnte die referenzierte Originalversion abgespeichert und eingeblendet werden.

### Visualisierung annotieren

Um eine gemeinsame Grundlage für eine Diskussion zu schaffen ist es notwendig, Bereiche der Visualisierung markieren zu können, über die man in einem Kommentar spricht. Diese Bereiche können entweder Flächen, die Datenpunkte enthalten oder nicht (Bereichskommentare), oder Datenpunkte selbst sein (Punktkommentare, Abbildung 3.16).

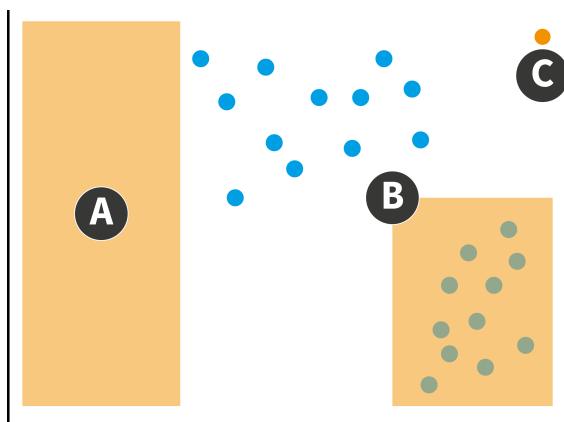


Abbildung 3.16: Ein Scatterplot mit eingezeichneten Annotationsmöglichkeiten: Leere Bereiche (A), nicht-leere Bereiche (B) und einzelne Datenpunkte (C).

### 3.7.2 API und Komponentenbeschreibung

Um *Wiederverwendbarkeit* über Komponenten hinweg zu fördern, sollten Kommentare sich möglichst auf die visualisierten Daten selbst beziehen (datenbezogene Kommentare) und nicht auf einen Bereich der Visualisierung (ortsbezogene Kommentare).

Komponenten, welche die Kommentarfunktion anbieten wollen, dürfen aus zwei Gründen keine externen Daten anzeigen (also beispielsweise die neuesten Tweets eines Twitter-Accounts): Erstens könnte der Kontext eines Kommentars sich ändern, etwa wenn Tweets gelöscht wurden. Die externen Daten unterliegen nicht der Kontrolle des Hilfesystems. Zweitens sind die externen Daten nicht im DaRe vorhanden, weswegen sie nicht über Komponenten hinweg wiederverwendet werden können. Dabei ist es unerheblich, ob die Komponente selbst externe Daten lädt oder externe Daten einer anderen Komponente anzeigt.

Um den Fall von eigenen externen Daten für das Hilfesystem beliebiger CRUI-Se Mashups erkennbar zu machen, kann die Komponentenbeschreibung erweitert werden. Für den Fall von fremden externen Daten muss das Hilfesystem nach erfolgreicher Integration aller Visualisierungskomponenten selbst überprüfen, welche davon externe Daten laden und mit welchen diese über Links (Abschnitt 2.3.1) verbunden sind. Im Fall von VizBoard wird festgelegt, dass Daten nur aus dem DaRe kommen.

#### Datenbezogene Bereichskommentare

Damit Bereichskommentare Daten referenzieren können, müssen drei Voraussetzungen gegeben sein:

1. Die grafische Repräsentation mindestens einer Datenvariable muss die Position, Breite oder Höhe als visuelles Mapping benutzen, damit eine Transformation zwischen kartesischen Koordinaten und Daten durchgeführt werden kann.
2. Die Komponente muss eine Operation bereitstellen, welche kartesische Koordinaten ( $x_1, y_1$ ) in Datenbereiche (2000 \$, 14. Mai 1998) transformiert sowie deren Umkehrung. Komponenten sind »Black Boxes« und es kann deswegen keine Annahme über die verwendete Skala (linear oder logarithmisch) oder andere Interna getroffen werden. Aus diesem Grund ist es nicht möglich, aus Auswahlkoordinaten ( $x_1, y_1, x_2, y_2$ ) direkt auf Daten zu schließen.
3. Die verwendeten Daten müssen im Data Repository gespeichert sein, da ansonsten keine Daten referenziert werden können. Aus diesem Grund darf weder die Komponente selbst Datenverarbeitung in Form von Aggregationen o. ä. durchführen, noch dürfen die Daten selbst bereits aggregiert in der Komponente ankommen.

Erfüllt eine Komponente diese Bedingungen nicht vollständig, können nur ortsbezogene Bereichskommentare eingesetzt werden. Wenn datenbezogene Bereichskommentare angezeigt und von anderen Benutzern nachvollzogen werden sollen, muss

das Hilfesystem die Komponente in den damals aktiven Zustand zurücksetzen können, da ansonsten die referenzierten Bereiche andere Daten enthalten können. Diese Tatsache führt zu einer weiteren Anforderung an die Komponente:

4. Die Komponente muss der Laufzeitumgebung einen Weg zur Verfügung stellen, mit dem Zustände (wie z. B. Zentrum und Zoomlevel einer Karte) gespeichert und vollständig wiederhergestellt werden können.

Diesbezüglich bietet sich ein Memento-Pattern [Gam+94] an:

Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later.

Um Mementos umzusetzen, wird auf Propertys zurückgegriffen. Eine Komponente muss ihren Zustand als Propertys öffentlich machen und zwar so, dass durch Ändern von Propertys kein inkonsistenter Zustand erreicht werden kann. Sie definiert das Format ihres Mementos, also welche Propertys es ausmachen, in der Komponentenbeschreibung. Das Hilfesystem kann dann über die Methoden `getProperty(p)` und `setProperty(p, v)`, welche schon in der API der Komponente vorhanden sind, das Memento abrufen bzw. wiederherstellen. Eventuell registrierte PropertyLinks (Abschnitt 2.3.1) würden aktiviert und andere Komponenten ebenfalls verändert. Nachdem Events für eine Komponente zurückgehalten werden müssen, da sich das User Interface sonst ändern könnte, während der Benutzer Kommentare liest, haben PropertyLinks keinen Effekt.

## Ortsbezogene Bereichskommentare

Datenbezogene Kommentare haben den Vorteil, dass sie auch in anderen Komponenten als der ursprünglich benutzten angezeigt werden können. Sie haben die Form eines Rechtecks, da so der Implementierungsaufwand für den Komponentenentwickler minimiert wird, obwohl prinzipiell beliebige geometrische Formen möglich sind. Andere Formen von Kommentaren sind zum Beispiel Pfeile oder Text, welche in der Nutzerstudie von [HVW07] am populärsten waren. Diese werden am besten als ortsbegrenzte Kommentare modelliert, da bei beiden Formen nicht eindeutig ist, welche Daten sie referenzieren. Siehe das Beispiel in Abbildung 3.17: Bezieht sich der Pfeil auf den Pixel am Ende der Pfeilspitze oder auf den roten Balken? Welche Daten soll der Text referenzieren?

## Punktkommentare

Mit Datenpunkten verhält sich die Situation etwas einfacher als bei Bereichen. Es kann davon ausgegangen werden, dass Datenpunkte und deren DOM Elemente in einer 1 : 1 (z. B. Balkendiagramm) oder 1 : n (z. B. Scatterplot-Matrix) Relation stehen, da die Komponente sonst aggregieren würde. Im vorigen Abschnitt wurde für datenbezogene Kommentare ausgeschlossen, dass Komponenten Datenverarbeitung durchführen. Wenn der Komponentenentwickler im Markup der Komponente sichtbar machen würde, zu welchem Datenpunkt ein Element gehört, könnte das Hilfesystem mit Hilfe von CSS Selektoren diese finden und beispielsweise auswählbar

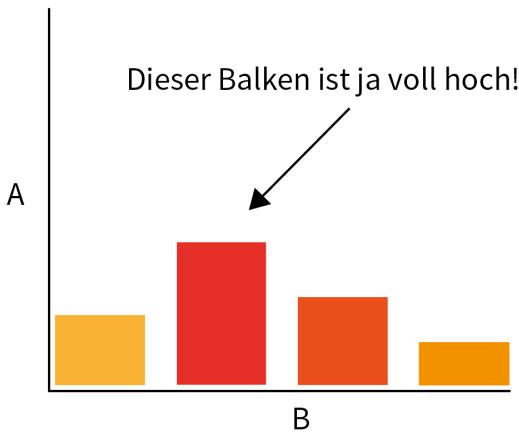


Abbildung 3.17: Ein ortsbezogener Kommentar mit Pfeil und Text

machen, wenn ein Kommentar verfasst werden soll. Wird der Kommentar abgeschickt, muss die Komponente dem Hilfesystem zuerst mitteilen, welche Werte die visualisierten Attribute eines Datenpunkts haben. Das kann nicht automatisch aus dem DOM Element berechnet werden, weil der Maßstab unbekannt ist: Entspricht eine Fläche von  $100 \times 100$  Pixel in der Treemap 10 Quadratmetern oder 1000 Hektar? Der Maßstab kann auch nicht in der Komponentenbeschreibung angegeben werden, weil er von der Größe der Komponente abhängig ist. Bei einer Breite von 500 Pixel wäre er z. B. 1 : 100, bei 250 Pixel Breite aber schon 1 : 200. Die URI des Datenpunkts sowie die Werte der Datatype Propertys werden im Kommentar gespeichert, sodass sie in anderen Visualisierungen wiederverwendet werden können.

Zuvor muss allerdings noch geklärt werden, auf welche Weise der Komponentenentwickler die URIs im Markup sichtbar machen soll. Prinzipiell kommen dafür alle Möglichkeiten in Frage, die mit Hilfe von CSS Selektoren gefunden werden können. Dazu gehören HTML Markup Elemente wie `id` oder Datenattribute `data-*`, aber auch Microformats<sup>12</sup> oder RDFa [W3C12]. Microformats werden ausgeschlossen, da bei diesen das Vokabular vorgegeben ist, während bei RDFa nur die Syntax definiert wird. Eine URI ist kein gültiger Wert für die `id` eines DOM Elements, deswegen wird auch sie ausgeschlossen. RDFa hat gegenüber Datenattributen den Vorteil, dass so auch zusätzliche Informationen über den Datenpunkt (beispielsweise das Label) ins Markup integriert werden können. Auf diese Weise kann das Hilfesystem unter Umständen den einen oder anderen Aufruf des DaRes einsparen und stattdessen HTML parsen, was zur Geschwindigkeit der Anwendung beiträgt.

Beim Laden von Punktkommentaren zählt das Hilfesystem erst die Anzahl von Kommentaren pro Datenpunkt und stellt diese Anzahl über dem visualisierten Datenpunkt dar. Das ist möglich, weil über die URI in Kombination mit CSS Selektoren die entsprechenden DOM Elemente identifiziert werden können.

### 3.7.3 User Interface und Interaktion

In der Titelleiste gibt es zwei Buttons mit einem Sprechblasen-Icon, einmal ist eine Zahl darin (Kommentare ansehen) und einmal ein Pluszeichen (Kommentar hin-

<sup>12</sup><http://microformats.org/>

zufügen). Eine ständig sichtbare textuelle Beschreibung wäre offensichtlicher, kann aber aufgrund des geringen Platzangebots nur mit einem Tooltip umgesetzt werden.

Wenn der Benutzer einen Kommentar verfassen will, erscheint eine Sidebar mit einer Textbox, Controls um die Annotationsform zu ändern (Auswahl, Rechteck, Pfeil, Text) und einem Button zum Bestätigen (Abbildung 3.18). Der Benutzer kann nun ein oder mehrere Rechtecke in der Visualisierung oder Datenpunkte markieren (linker Balken in Abbildung 3.18). Das Hilfesystem setzt die Komponente in den Ausgangszustand zurück, sobald der Benutzer bestätigt oder abgebrochen hat.

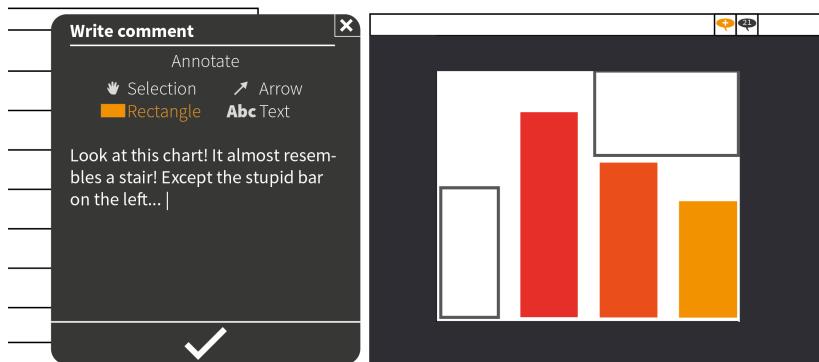


Abbildung 3.18: UI Mockup: Kommentar hinzufügen

Möchte der Benutzer Kommentare anderer lesen, wechselt der Inhalt der Sidebar zu einer Liste aller zu diesen Daten verfassten Kommentare (Abbildung 3.19). Ein Kommentar zeigt folgende Elemente.

- Linke Spalte
  - ID des Kommentars: Diese ist zur Referenzierung notwendig.
  - Avatar des Benutzers und Benutzername: Diese sollen das Vertrauen der Benutzer untereinander erhöhen und so die Qualität der Kommentare verbessern [CHW06]. Gespeichert und abgerufen werden sie im UserService von CRUISe.
  - Datum der Erstellung oder letzten Editierung, was jünger ist. Wurde editiert, ist das Datum kursiv geschrieben.
  - Editieren/Löschen-Buttons, falls der Kommentar selbst verfasst wurde.
  - Antworten: Eine zweite Möglichkeit als die ID des Kommentars einzutippen. Die UI wechselt zu »Kommentar verfassen« und inkludiert die ID des betreffenden Kommentars.
  - Annotationen anzeigen: Zeigt die Annotationen des Kommentars in der Visualisierung.
  - Voting: Ein Benutzer kann einen fremden Kommentar genau einmal bewerten. Die Anzahl der Stimmen wird ebenfalls angezeigt. Anstatt »upvote/downvote« wird hier die Wortkombination »agree/disagree« verwendet, da bei der Nutzerstudie (Abschnitt 3.1.3) ersteres selten mit »bewerten« und manchmal mit »scrollen« assoziiert wurde.

- Rechte Spalte

- Kommentartext mit Referenzen (durch #hash gekennzeichnet) und URLs (unterstrichen).

Referenzen auf andere Kommentare können angeklickt werden. Dann wird der gewählte Kommentar in der Sidebar angezeigt und die Titelleiste enthält einen Zurück-Button, mit dem der vorherige Zustand wiederhergestellt werden kann.

Die Datenpunkte in der Visualisierung werden mit einer Anzeige, wie viele Kommentare dazu vorhanden sind, erweitert (Abbildung 3.19). Sie sind oben rechts platziert, um so an iOS Badges zu erinnern und damit Assoziationen zu einem Hinweis zu wecken.

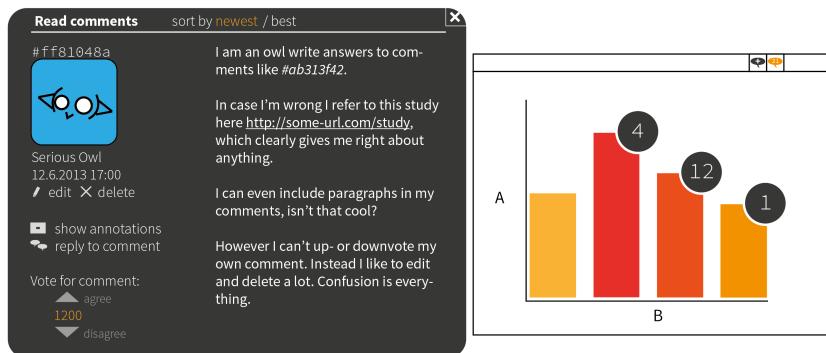


Abbildung 3.19: UI Mockup: Anzeige aller Kommentare

Diese Badges können angeklickt werden. Die Sidebar enthält dann alle Kommentare, die sich auf den gewählten Datenpunkt beziehen (Abbildung 3.20).



Abbildung 3.20: UI Mockup: Alle Kommentare zum ausgewählten Datenpunkt

Annotationen eines Kommentars müssen durch einen Klick auf den Button »show annotations« angezeigt werden. Der Viewport der Visualisierung wird dann abgedunkelt und nur die Annotationen des ausgewählten Kommentars bleiben sichtbar (Abbildung 3.21).

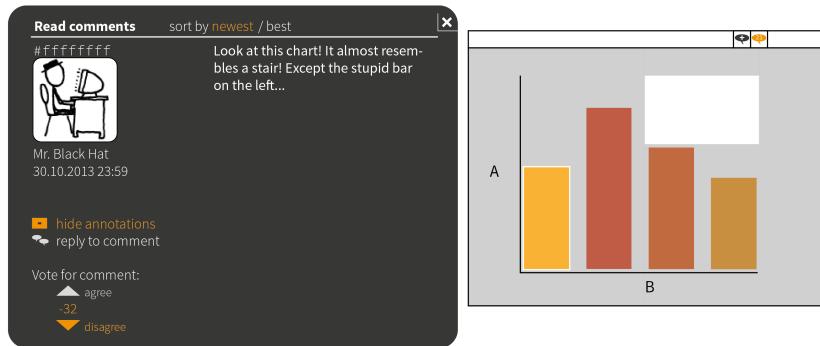


Abbildung 3.21: UI Mockup: Ausgewählter Kommentar mit referenziertem Bereich und Datenpunkten

### 3.7.4 Backend

Den Ausführungen in den Abschnitten 3.7.1 und 3.7.2 nach enthält ein abgespeichertes Kommentar im DaRe folgende Daten:

- ID des Kommentars, um darauf verweisen zu können.
- ID der Komponente, in der der Kommentar verfasst wurde. Das ist notwendig um ortsbezogene Kommentare wiederherstellen zu können. Bei datenbezogenen Kommentaren kann eventuell ein Verweis auf die ursprüngliche Komponente erstellt werden.
- ID des Datensatzes, der visualisiert wurde
- ID des Verfassers
- URIs der Datatype Propertys, die visualisiert wurden
- Memento der Komponente zum Zeitpunkt des Kommentars
- *Für jede Version des Kommentars*
  - Versionsnummer
  - Zeitpunkt
  - Kommentartext
  - *Wenn ortsbezogene Elemente vorhanden*
    - \* Für jedes Element (in relativen Einheiten)
      - Wenn Pfeil:  $(x_1, y_1, x_2, y_2)$
      - Wenn Text: Text,  $(x, y)$
      - Wenn Rechteck:  $(x_1, y_1, x_2, y_2)$
  - *Für jedes datenbezogene Element*
    - \* Grenzwerte der visualisierten Datatype Propertys. Ist eine Datatype Property nominalen Typs, besitzt also keine inhärente Ordnung (z. B. Länder), muss eine Menge von gültigen Werten angegeben werden.

- *Für jeden Datenpunkt*
  - \* URI
  - \* Werte der visualisierten Datatype Propertys

Hat ein Benutzer Bereiche und Datenpunkte referenziert, sowie einen Kommentar geschrieben, sammelt das Hilfesystem die eben beschriebenen Daten der Komponente und sendet sie ans Data Repository. Das Speichern und Laden von Kommentaren liegt in dessen Verantwortung.

Sollen Kommentare angezeigt werden, schickt das Hilfesystem eine entsprechende Anfrage ans DaRe und empfängt die Daten aller relevanten Kommentare. Für ortsbegrenzte Bereichskommentare ist nichts weiter zu tun, sie können direkt in dieser Komponente gerendert werden. Um datenbezogene Bereichskommentare anzuzeigen, muss das Hilfesystem zuerst die Datengrenzen in Koordinaten in der Visualisierung transformieren lassen. Bei Punktkommentaren ist dies nicht nötig, weil das Hilfesystem über CSS Selektoren herausfinden kann, wo sie sich befinden sollen.

Um zu gewährleisten, dass der Benutzer störungsfrei Kommentare lesen kann, muss das Hilfesystem den Event Broker (Abschnitt 2.3.1) veranlassen können, anfallende Events nicht weiterzuleiten (»suspend«) und in einer FIFO Warteschlange zwischenzulagern. Ansonsten könnte der Fall eintreten, dass die Komponente aktualisiert wird, während der Benutzer Kommentare zu den Daten liest oder eine andere Hilfe in Anspruch nimmt. Diese Funktion ist in Grundzügen bereits für das Handling von PropertyLinks vorhanden [Pie12, S. 161]. Nachdem die Kommentar UI geschlossen wurde, kann der Event Broker die gespeicherten Events weiterleiten (»resume«). Sie werden einfach in Reihenfolge weitergegeben. Das hat den Nachteil, dass sich das User Interface mehrmals schnell ändern könnte, bis alle pausierten Events abgearbeitet sind. Eine andere Möglichkeit wäre die Pausierungsfunktion auf Komponentenebene umzusetzen. Das würde allerdings Mehraufwand sowohl für den Komponentenentwickler, der die entsprechenden Methoden implementieren müste, als auch für das Hilfesystem, welches jede Komponente einzeln ansprechen muss, bedeuten.

## Weitere Nutzung

Kommentare können auch Metainformationen über die Daten oder Komponente enthalten [CYR09]. Es wäre denkbar, mittels automatischer Textklassifikation [Seb02] diese Kommentare zu identifizieren und als positiv oder negativ zu kategorisieren. Daraus ließe sich wiederum eine Bewertung für die Kombination (Datensatz, Komponente) berechnen<sup>13</sup>. Der errechnete Wert könnte in Zukunft beim Ranking einer Komponente für einen ausgewählten Datensatz berücksichtigt werden. Wenn der Aufwand dafür gerechtfertigt werden kann, ist es auch möglich, eine zuvor definierte Ontologie aus Kommentaren zu extrahieren [Ala+03]. Diese könnte zum Beispiel das Qualitätsmanagement von Komponenten automatisieren. Denkbar wäre, dass aus Kommentaren automatisch extrahiert wird, worüber sich die Benutzer beschweren. Ab einem Schwellwert kann dann der Komponentenentwickler informiert werden.

<sup>13</sup>Am einfachsten ginge das indem bei Null gestartet wird und für jeden positiven Kommentar der Zähler um eins erhöht und für jeden Negativen um eins verringert wird.

## 3.8 History

In den verwandten Arbeiten (Abschnitt 2.4) wurde mehrmals empfohlen, dem Benutzer Undo/Redo von getätigten Aktionen zu ermöglichen. So können unerwünschte Ergebnisse einfach rückgängig gemacht werden. Heer et al. [Hee+08] unterscheiden dabei zwei Modelle einer History: State und Action. Bei einer State History sind die Knoten Applikationszustände und die Kanten Aktionen, bei einer Action History ist es umgekehrt. Undo in einer Action History bedeutet in Reihenfolge inverse Aktionen auszuführen, in einer State History ist es die Wiederherstellung eines gespeicherten Zustands.

In diesem Abschnitt wird untersucht, wie in einem CRUISe Mashup, in dem die Komponenten lose gekoppelt sind, ein kompositionsübergreifendes Undo/Redo umgesetzt werden kann. Die Granularitätsstufe entspricht dabei Nutzerinteraktionen beziehungsweise zusammenhängenden anstatt einzelnen Events oder Operationen. Das sollte sich wie in den verwandten Arbeiten von Grammel [Gra12] vorgeschlagen mit dem mentalen Modell des Benutzers decken, da für ihn Events und Operationen unsichtbar sind. Als Beispiel wird die Komposition in Abbildung 3.22 verwendet.

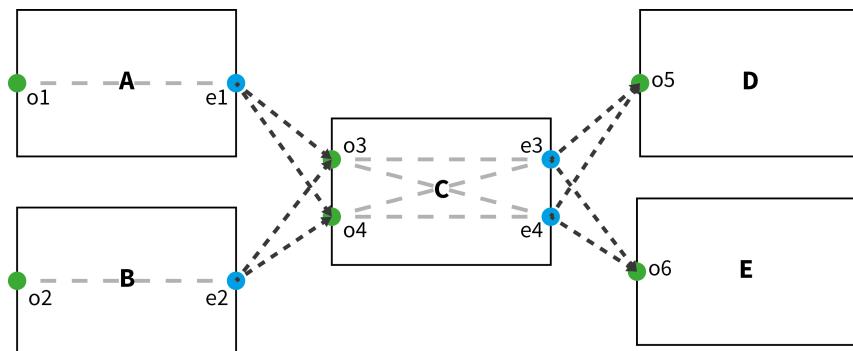


Abbildung 3.22: Beispielhafte Komposition

In Abschnitt 3.7.2 wurde ein Memento-Pattern für Komponenten eingeführt. Damit kann das Hilfesystem Zustände von Komponenten abrufen, speichern und wiederherstellen. Im Folgenden wird davon ausgegangen, dass diese Mementos ausreichen, um eine Komponente von einem konsistenten Zustand in einen anderen konsistenten Zustand zu überführen (und beispielsweise keine Operationen ausgeführt werden müssen). Das deckt sich mit dem Paradigma von CRUISe, wonach die Menge von Propertys den Zustand einer Komponente vollständig beschreibt. Gegebenenfalls muss die Komponente selbst ihre Konsistenz überprüfen, beispielsweise wenn eine Liste von Flügen in einen späteren Zustand wiederhergestellt werden soll und einige inzwischen ausgebucht sind. Es stellt sich nun die Frage, wann die Mementos der Komponenten abgerufen werden sollen. Prinzipiell kann es vor oder nach einem Event (welches Operationen aufruft, die potentiell den Zustand der Komponente ändern) passieren.

**Vor** einem Event die Mementos abzurufen ist nicht möglich, da ein Event erst registriert wird, wenn die Nachricht im Event Broker ankommt. Zu diesem Zeitpunkt kann eine Komponente ihren Zustand bereits geändert haben. Das Hilfesystem könnte eigene Event Handler auf die Operationen der actionsrelevanten UI Elemente registrieren und so bemerken, wann ein Event ausgelöst wird. Allerdings

kann das Hilfesystem erst Event Handler registrieren, wenn die Komponente initialisiert wurde. Zu diesem Zeitpunkt wurde der DOM geladen und Event Handler des Komponentenentwicklers registriert. Die Event Handler des Hilfesystems würden zwangsläufig nach den Handlern des Entwicklers ausgeführt. Wenn es soweit ist, könnte die Komponente schon ihren Zustand verändert haben und das Memento enthält nicht die gewünschten Attributwerte.

Eine mögliche Lösung wäre den Lebenszyklus der Komponenten zu modifizieren. Es könnte ein Zustand *Rendered* zwischen *Integrated* und *Active* eingefügt werden (Abbildung 3.23), bei dem der DOM der Komponente vollständig vorhanden sein, aber Event Handler noch nicht registriert sein sollen. Zu diesem Zeitpunkt könnte das Hilfesystem seine Handler vor jenen des Komponentenentwicklers registrieren, die dann auch als erstes ausgeführt würden. Problematisch ist allerdings, wenn eine Komponente zur Laufzeit DOM Elemente erstellt und Event Handler registriert. Dieses Verhalten müsste man verbieten, um sicherzustellen dass Handler des Hilfesystems immer zuerst ausgeführt werden. Das ist aber nicht wünschenswert, weil diese Vorgabe möglicherweise große Anpassungen des Komponentencodes nötig macht und oft auch gar nicht umsetzbar ist.

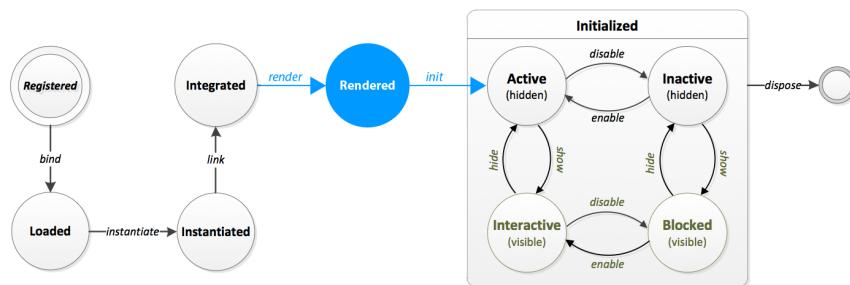


Abbildung 3.23: Modifizierter CRUISe Komponentenlebenszyklus

Es bleibt die Möglichkeit **nach** einem Event Mementos abzurufen. Zunächst ist es schwierig festzustellen, wann »nach« einem Event überhaupt ist. Events können in dem Sinne kaskadieren, als dass sie Operationen aufrufen, die weitere Events feuern, die wiederum Operationen ausführen, die selbst Events feuern und so weiter. Aus dem Kommunikationsmodell kann a priori nicht eindeutig bestimmt werden, welche Events nach einem Event  $e_0$  zu erwarten sind. Das ist eine Folge des »Black Box« Prinzips der Komponenten. Im folgenden Abschnitt wird ein Konzept erarbeitet, mit dem das Hilfesystem kaskadierende Events identifizieren kann.

### 3.8.1 Backend

Ein Ansatz zur Lösung des Problems ist die aufrufende Methode in die Eventnachricht zu schreiben. Das muss der Komponentenentwickler übernehmen. So kann das Hilfesystem bzw. der Event Broker zur Laufzeit Eventkaskaden erkennen, da das Ziel einer Nachricht (CRUISe Operation) denselben Wert enthält wie die aufrufende Methode einer anderen Nachricht. Zusätzlich zur aufrufenden Methode muss noch ein Timestamp, eine eindeutige Event ID und die Instanz ID der Komponente angegeben in einer Nachricht enthalten sein, was aber der Event Broker übernehmen kann.

Der Ablauf des Algorithmus wird anhand der Situation in Abbildung 3.24 erklärt. Sie zeigt einen beispielhaften Ablauf, wo  $e_3$  und  $e_4$  gefeuert wurden und  $o_3$  bzw.  $o_4$  aufrufen, weil eine ursprünglich eine Aktion in Komponente A ausgeführt wurde.

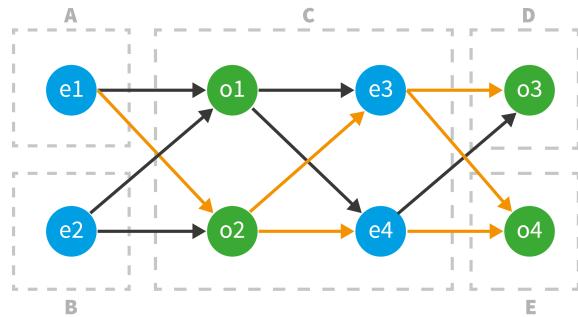


Abbildung 3.24: Möglicher Ablauf der Events in der beispielhaften Komposition

Der Algorithmus funktioniert wie folgt:

1. Komponente A feuert Event  $e_1$ .
2. Das Hilfesystem registriert die Nachricht  $e_1$  und sucht nach einer Collection mit einer Nachricht  $e_i$ , die der Vorgänger von  $e_1$  sein könnte. Da keine vorhanden ist, fügt das Hilfesystem  $e_1$  in eine neu erstellte Collection hinzu und legt ein Timeout von  $t$  Millisekunden fest. Läuft es ab, fährt der Algorithmus bei Schritt 7 fort. Außerdem wird  $e_1$  bis dorthin der Methode  $o_2$  von C zugeordnet.
3. Innerhalb des Timeouts  $t$  feuert Komponente C Event  $e_3$ .
4. Das Hilfesystem registriert die Nachricht  $e_3$  und überprüft, ob es eine Collection mit einem Vorgänger dafür gibt. Das ist der Fall, also liest das Hilfesystem aus der Nachricht, welche Methode sie geschickt hat ( $o_2$ ). Danach wird eine Collection gesucht, welche eine der  $o_2$  zugeordneten Nachrichten enthält. Findet das Hilfesystem eine, fügt es  $e_3$  dieser hinzu und setzt den Timer auf 0 zurück. In diesem Beispiel ist das der Fall. Würde das Hilfesystem keine Collection finden, führe es fort wie in Schritt 2.
5. Innerhalb des Timeouts  $t$  feuert Komponente C Event  $e_4$ .
6.  $e_4$  wird gleich wie  $e_3$  in Schritt 4 behandelt.
7. Das Timeout  $t$  läuft ab.
8. Das Hilfesystem markiert die Collection als geschlossen und entfernt die Assoziation zwischen enthaltenen Nachrichten sowie den Methoden der Komponenten. Da die zustandsverändernde Operation von Komponente D erst aufgerufen werden muss, legt es ein Timeout  $t_D$  fest, nach dessen Ablauf die Mementos der Komponenten abgerufen werden können.

Ein noch nicht betrachteter Punkt sind asynchrone Operationen. Die Reihenfolge der Nachrichten spielt im vorgestellten Algorithmus keine Rolle, weswegen asynchrone Operationen wie synchrone behandelt werden, solange sie innerhalb des Timeouts

$t$  ein Event feuern. Das festgelegte Timeout ist aber keine gute Designentscheidung, wenn es um Zugriffe auf externe APIs geht. Es kann nicht davon ausgegangen werden, dass sie innerhalb desTimeouts beantwortet werden. Aus diesem Grund wird bei der History – wie bei den Kommentaren (Abschnitt 3.7) auch – ausgeschlossen, dass Komponenten externe Daten laden können.

Für das Timeout  $t_D$ , welches auf die Ausführung der Operation der letzten beteiligten Komponente wartet, kann der Wert aus der Bedienungshilfe (Abschnitt 3.3) herangezogen werden, wenn vorhanden. Dieser teilt dem Hilfeservice, der die Comic Panels generiert, mit, wann die Zustandsveränderung der Komponente abgeschlossen ist. Ansonsten wird ein durch Tests bestimmter Standardwert verwendet.

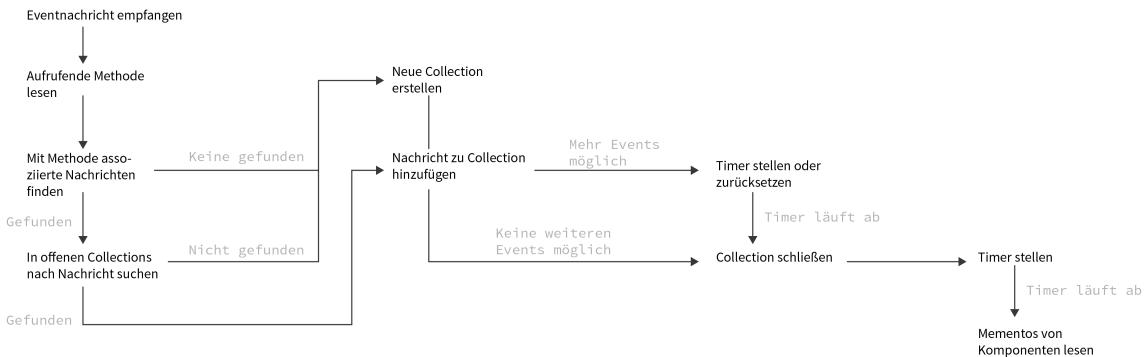


Abbildung 3.25: Ablaufdiagramm des History Algorithmus

Außerdem ist zu beachten, dass für die korrekte Arbeitsweise des vorgestellten Algorithmus jedes Event eine Nachricht senden **muss**, sei es auch nur »Methode ausgeführt«. Ansonsten könnte eine Komponente ihren Zustand verändern, ohne dass es der Event Broker und damit das Hilfesystem merkt. Die Undo-Reihenfolge wäre dann nicht mehr korrekt. Das impliziert weiterhin, dass Zustandsveränderungen ein Event feuern müssen. Diese Forderung ist schon umgesetzt, da zum Beispiel `setProperty(p, v)` eine Operation ist, die `propertyChanged` Events feuert und die Menge an `propertys` den Zustand einer Komponente vollständig beschreiben.

Bevor konkrete visuelle Umsetzungen der History präsentiert werden, muss diskutiert werden, wie mit von Komponenten (anstatt vom Benutzer) ausgelösten Aktionen umgegangen wird. Beispielsweise wäre eine Uhrzeit-Komponente denkbar, die jede volle Minute ein `minutePassed` Event auslöst. Klassische Implementierungen einer History löschen den Redo-Stack, wenn nach einem Undo eine neue, nicht in der History vorhandene Aktion ausgeführt wird. Im Falle der Uhrzeit-Komponente wäre das äußerst ungünstig, da sie spätestens nach einer Minute unerwünschterweise den Redo-Stack löschen würde. Derartige Komponenten von der History auszuschließen ist jedoch keine Option, da sie aufgrund des Ziels der *Universalität* alle Komponenten unterstützen muss. Events nach einem Undo zu pausieren ist ebenfalls nicht möglich, da dann überhaupt keine neuen Aktionen mehr ausgeführt werden können. Deswegen sollten neue, durch Komponenten ausgelöste Aktionen nach einem Undo weiter auf den Redo-Stack gelegt werden. Dieser kann gelöscht werden, sobald der Benutzer eine Aktion ausführt. Der im vorhergehenden Abschnitt vorgestellte Algorithmus kann eine Nutzeraktion aber nicht in allen Fällen von einer Komponentenaktion unterscheiden. Zwar gibt es in der SMCDL ein Attribut `trigger`, welches angibt, ob ein Event durch Nutzerinteraktion, infolge einer Operation oder interner

Logik ausgelöst wird [Pie12, S. 98]. Dieses ist jedoch optional und außerdem muss es nicht immer festgelegt sein: Es ist möglich, dass ein Event sowohl durch eine Operation als auch durch Nutzerinteraktion ausgelöst wird.

Um dem beizukommen, kann das Hilfesystem eigene Eventhandler auf aktionsrelevanten UI Elementen registrieren. Diese sind in Folge der Erweiterungen durch die Bedienungshilfe (Abschnitt 3.3) bekannt. Abbildung 3.26 verdeutlicht die Funktionsweise: Zuerst wird eine Nachricht  $e_1$  an den Event Broker gesendet (1). Dieser fügt sie einer Collection hinzu (2). Danach wird der Eventhandler des Hilfesystems aufgerufen (3). Das Hilfesystem weiß, zu welcher Komponente, Aktion und Operation der Eventhandler gehört (Abschnitt 3.3.1) und kann in den Collections des Event Brokers nach entsprechenden Nachrichten suchen. Von diesen wird die aufrufende Methode gelöscht und sie werden in eine neue Collection eingefügt (4). Danach kann das Hilfesystem den Redo-Stack löschen, falls nötig.

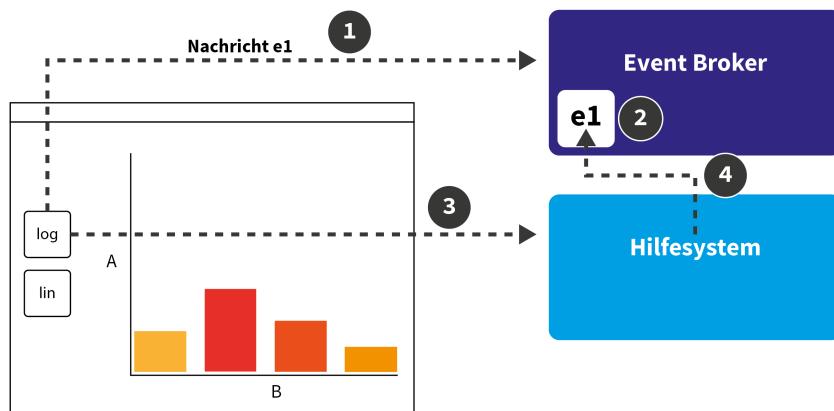


Abbildung 3.26: Erkennung von Nutzerinteraktionen

Diese Vorgehensweise funktioniert allerdings nicht, wenn der Entwickler ein Event auslöst, indem er programmatisch ein UI Element betätigt, beispielsweise durch jQuerys `click()` Methode. Eine Möglichkeit, dies zu verhindern, wäre mit Hilfe statischer Codeanalyse während der Registrierung einer Komponente.

### 3.8.2 User Interface und Interaktion

Der im vorangehenden Abschnitt vorgestellte Algorithmus, welcher zu gegebener Zeit Mementos der Komponenten abruft, speichert und wiederherstellt, macht die hier behandelte History zu einer State History. Um die State History einfach bedienen zu können, sollten die verschiedenen States erkennbar dargestellt werden.

#### Umsetzung einer webbasierten State History

Prinzipiell kann der in Abschnitt 3.3.2 vorgestellte Algorithmus zur Generierung von Screenshots auch dynamisch aufgerufen werden. Alle notwendigen Informationen sind bekannt: Der geladene Datensatz, die Mappings, die betroffenen Komponenten, die ausgeführten Aktionen und Operationen. Das Hilfesystem könnte beim Start der kompositen Ansicht den Generierungsservice mit denselben Komponenten und Daten initialisieren. Nach jeder registrierten Operation würde es diese Daten an

den Generierungsservice schicken, der einen Screenshot der Applikation in diesem Zustand zurück gibt. Jedoch sollte es nie länger als zwei Sekunden dauern, bis ein Screenshot in der History verfügbar ist. Von der Tatsache abgesehen, dass dieser Vorgang zeitkritisch ist, ist er leider auch nicht umsetzbar: Wegen des »Black Box« Prinzips der Komponenten können Zustandsveränderungen beliebig lange dauern. Deren obere Grenze ist zwar bekannt (Abschnitt 3.3.1), aber der Generierungsservice müsste die Zeit trotzdem abwarten, bevor ein Screenshot erstellt werden kann. Diese Zeit muss von den veranschlagten zwei Sekunden abgezogen werden, um die verfügbare Rechen- und Transferzeit zu erhalten. Es erscheint unwahrscheinlich, dass der beschriebene Vorgang in so kurzer Zeit abgearbeitet werden kann. Außerdem könnte die History einen fehlerhaften Screenshot enthalten, wenn ein Screenshot frühzeitig angefertigt würde.

Eine andere Variante, um an realitätsgerechte Abbildungen von Komponentenzuständen zu gelangen, wäre ihren DOM zu kopieren und verkleinert in der History anzuzeigen. Problematisch sind dabei CSS Regeln mit absoluten Einheiten, wie zum Beispiel Pixel oder Punkt. Eine Lösung dafür wäre zur Laufzeit das CSS einer Komponente in relative Einheiten zu transformieren. Abbildung 3.27 zeigt einen naiven Algorithmus als ersten Ansatz. Dabei wird am Wurzelement gestartet (DIV B) und die relative Breite und Höhe zum Elternknoten (DIV A) berechnet (Schritt 1). Diese Attribute werden im CSS gesetzt. Danach wird der Algorithmus rekursiv für die Kinder von DIV B aufgerufen, also DIV C und D, und nach demselben Prinzip fortgefahrene.

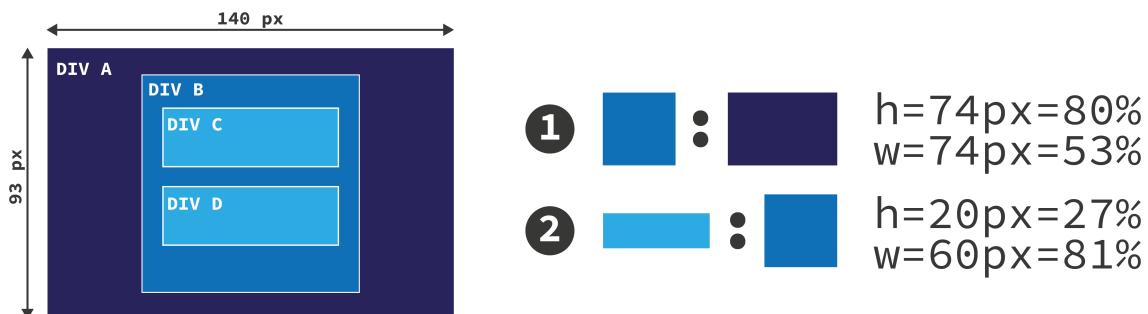


Abbildung 3.27: Naiver Algorithmus um CSS zu in relative Einheiten zu transformieren

Der vorgestellte Algorithmus hat einige offensichtliche Schwachstellen. Beispielsweise gibt es

- Elemente ohne definierte Breite oder Höhe (`svg:g`),
- mit nicht-rechteckiger Größendefinition (`svg:circle`, `svg:path`) oder
- Attribute, welche ebenfalls skaliert werden müssen (`margin`, `x`).

Hinzu kommt, dass sowohl alle möglichen Elemente, also auch Clip Paths, Polygone, Polylinien und so weiter als auch deren Transformationen (verschieben, rotieren, skalieren, verzerren etc.) korrekt skaliert werden müssen. Außerdem ist der Vorgang möglicherweise sehr langsam, wenn der Browser nach jedem Update eines layoutrelevanten CSS Attributs (wie Breite oder Höhe) das Layout neu berechnen muss

(»Reflow«) und währenddessen das User Interface blockiert. Der Vorgang erscheint zwar nicht einfach, aber lösbar und sollte dennoch möglichst vermieden werden.

Noch besser als CSS zur Laufzeit zu transformieren wäre, wenn der Komponentenentwickler von vornherein relative Einheiten verwendete. Das wäre ohnehin zu bevorzugen, da Komponenten theoretisch beliebig ihre Größe ändern können. Dennoch sollte es kein Muss-Kriterium für den Komponentenentwickler sein, da es längere Entwicklungszeiten nötig macht und den Aufwand erhöht. Um relative CSS Einheiten zu forcieren, wird deren Nutzung angenommen. Andernfalls muss der Komponentenentwickler in der Komponentenbeschreibung das Gegenteil angeben, sodass das CSS der Komponente zur Laufzeit transformiert wird.

### Visuelle Repräsentation der State History

Wie in [Hee+08] besteht die State History aus Bildern der Komponenten mit einer Seitenlänge von mindestens 120 Pixel (Abbildung 3.28). Dabei werden Komponenten hervorgehoben, die sich seit dem vorhergehenden Applikationszustand verändert haben. Nach einem Klick auf einen Zustand wird die Komposition in diesen zurückgesetzt.



Abbildung 3.28: UI Mockup: Grafische History

Sollte sich die dynamische Transformation von CSS Regeln als nicht praktikabel herausstellen, werden zwei Buttons für Undo und Redo eingesetzt (Abbildung 3.29), wenn Komponenten ohne relatives CSS in der Komposition vorhanden sind. Diese Art der History ist auch in jeder Desktopapplikation vorhanden. Die Buttons sind sofort verständlich und erfüllen ihren Zweck, allerdings wird die Interaktion umso mühsamer, je weiter der zu erreichende Anwendungszustand zurück liegt.

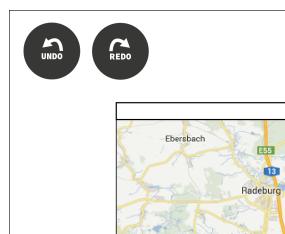


Abbildung 3.29: UI Mockup: Undo/Redo Buttons

## 3.9 Meta-Hilfe

In diesem Konzept wurden bisher sieben unterschiedliche Hilfefunktionen erarbeitet: Intro, Bedienung, Feedback, Verlinkung, Kommunikation, Kommentare und History. Diese sind zwar optisch und vom Interaktionsdesign her konsistent, bringen aber trotzdem viel zusätzliche Komplexität in die Anwendung. Um die Hilfefunktionen und damit die Anwendung in vollem Umfang nutzen zu können (*Vollständigkeit*), benötigt der Benutzer einige Informationen. Dazu gehört sowohl das Wissen, welche Hilfefunktionen existieren und wie sie aufgerufen werden, als auch konzeptionelle Dinge, wie z. B. dass Kommentare mit Bereichen und/oder Datenpunkten in der Visualisierung assoziiert sind. Es ist anzunehmen, dass diese Meta-Hilfe vor allem bei der ersten Nutzung von VizBoard benötigt wird. Aus diesem Grund sollte sie so *unaufdringlich* wie möglich gestaltet sein. Prinzipiell können Hilfestellungen unterteilt werden in dynamische Hilfe, die dem Nutzer bei Bedarf angeboten wird, und statische Hilfe, die der Nutzer selbst aufrufen muss.

### 3.9.1 Statische Meta-Hilfe

Die einfachste Lösung für die statische Meta-Hilfe wäre ein Button in der rechten oberen Ecke, der bei Bedarf eine klassische Hilfefunktion aufruft. Dort werden die verschiedenen Funktionen in Wort und Bild erklärt und ein Tutorialvideo gezeigt. Das bedeutet relativ viel Zeit- und Klickaufwand, bis der Nutzer bei den gewünschten Informationen ist, aber es hat auch Vorteile: Durch den statischen Hilfeknopf ist immer klar, wo weitergeholfen wird. Außerdem kann er gegebenenfalls für andere Sachen benutzt werden, beispielsweise könnte dort konfiguriert werden, ob der Pfeil in der Kommunikationshilfe bei jeder Nachricht angezeigt werden soll.

### 3.9.2 Dynamische Meta-Hilfe

Ein Nachteil der statischen Hilfe ist, dass sie nicht lernt und sich dem Benutzer nicht anpasst. Die dynamische Hilfe ändert das. Allerdings stellt sich die Frage, woher sie weiß, wann sie welche Hilfe anbieten soll.

Die beste Lösung wäre sicher, wenn das Hilfesystem erkennen könnte, wenn der Benutzer verwirrt oder verärgert ist. Das fällt unter den Begriff »User Affection Detection/Recognition«. Mögliche Ansätze dazu benötigen Sensoren wie EEG oder Eyetracking, die in VizBoard nicht verfügbar sind [LJ05; Lia+06]. Eine andere Variante wäre Rückschlüsse auf den Gemütszustand des Benutzers mit Hidden Markov Models [RJ86] zu ziehen. Allerdings könnten in VizBoard nur Mausklicks und Tastaturanschläge als Beobachtungen verwendet werden, was vermutlich nicht zu zufriedenstellenden Ergebnissen führen würde.

Eine weitere Lösungsmöglichkeit ist das Konzept von Trace-Based Reasoning [Cor+13] anzuwenden. Durch maschinelles Lernen [Kot07] von Daten über Nutzer, Komposition, Umgebung und Interaktionsgeschichte werden Regeln z. B. in Form eines Decision Trees gefunden, wann Hilfe angeboten werden soll. So kann das Hilfesystem aus vergangenen Erfahrungen mit Nutzern lernen. Der Nachteil ist allerdings, dass die Ergebnisse nicht zufriedenstellend sein werden, bis dem Hilfesystem ausreichend Trainingsdaten zur Verfügung stehen. Abbildung 3.30 zeigt einen Überblick des Systems.

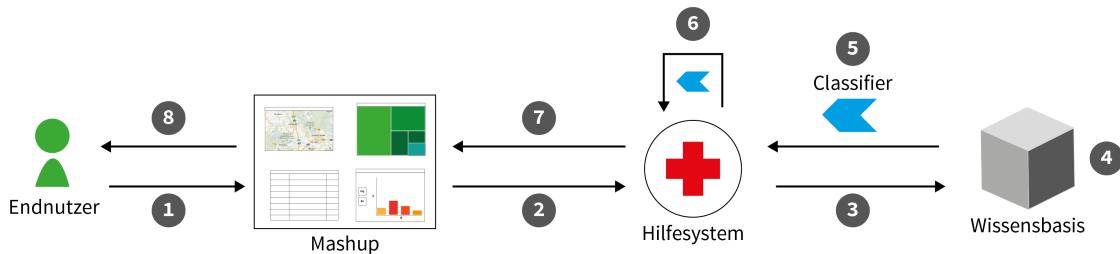


Abbildung 3.30: Architektur der dynamischen Hilfefunktion

Der Ablauf ist wie folgt:

1. Der Nutzer interagiert wie gewohnt mit dem Mashup.
2. Das Hilfesystem nimmt die Interaktionen des Nutzers auf.
3. Das Hilfesystem schickt die erfassten Daten an die Wissensbasis. Um die dynamische Hilfe auch für andere auf CRUISe basierende Mashups verfügbar zu machen, sollte diese im CoRe integriert sein.
4. Die Wissensbasis erlernt Regeln aus den Daten, beispielsweise männliche Architekten brauchen in der Nacht Hilfe zur Kommunikation, wenn sie die Komponente XY nutzen.
5. Die Wissensbasis schickt den generierten Classifier an das Hilfesystem.
6. Das Hilfesystem benutzt den Classifier, um den geeigneten Zeitpunkt für ein Hilfeangebot zu finden.
7. Das Hilfesystem bietet Hilfe an.
8. Der Benutzer profitiert von der Hilfe.

Supervised Machine Learning Algorithmen gehen als Datengrundlage von einer Tabelle aus, in der die Zeilen Instanzen und die Spalten Attribute oder Facetten der Instanzen sind (Abbildung 3.31). Die Menge von Spalten wird »Feature Vector« genannt.

	Attribute		
	Leistung	Max. Geschw.	0-100 km/h
Instanzen	Ford Mustang GT	420 PS	241 km/h
	Ferrari F40	478 PS	324 km/h
	Dodge Challenger	375 PS	293 km/h
			4.7 Sek.
			3.8 Sek.
			4.9 Sek.

Abbildung 3.31: Datengrundlage für SML Algorithmen

Je nach Algorithmus können Attribute unterschiedlichen oder festen Typs sein: Decision Trees arbeiten sowohl mit nominalen als auch ordinalen und quantitativen Variablen, während Support Vector Machines nur quantitative Daten unterstützen.

Allen Algorithmen gemein ist jedoch, dass Attribute eindimensional sein müssen und beispielsweise keine Arrays oder Mengen sein dürfen. Weil für die Meta-Hilfe verschiedene Daten herangezogen werden müssen, wird ein Decision Tree Algorithmus gewählt. Ein klassifizierender Decision Tree nimmt einen Feature Vector als Input und weist ihn einer von mehreren Kategorien zu. Im Fall der dynamischen Meta-Hilfe sind das »Keine Hilfe nötig« und die verschiedenen Hilfefunktionen (Abbildung 3.32).

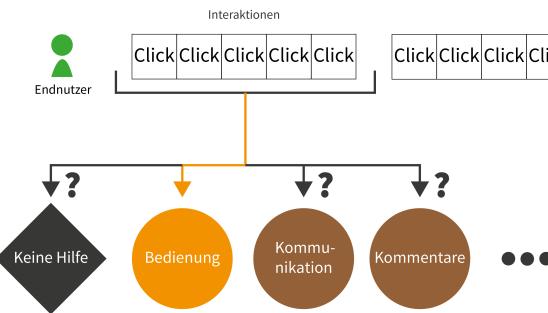


Abbildung 3.32: Klassifizierender Decision Tree Algorithmus

Nun stellt sich die Frage, welche Daten für das maschinelle Lernen herangezogen werden sollen, d. h. wie der Feature Vector aussieht. Wie bereits erwähnt, sind verschachtelte Attribute wie Arrays oder Mengen nicht zulässig, d. h. es können nicht alle Interaktionen des Nutzers herangezogen werden (weil sie nicht eine Zelle der Tabelle passen). Darum werden die Facetten mehrerer Interaktionen, zum Beispiel in welcher Komponente sie stattfand oder welche Operation es war, zum Feature Vector hinzugefügt. Da die Anzahl von Spalten der Tabelle Einfluss auf die Laufzeit des Algorithmus hat (weniger ist besser), aber mehr Spalten potentiell mehr Erkenntnisse ermöglichen (mehr ist besser) und das Hilfeangebot nicht zu spät erfolgen darf (weniger ist besser), scheinen fünf Interaktionen ein guter Kompromiss zu sein (Abbildung 3.32). Jedoch sollte das in Tests genauer evaluiert werden. Es erscheint sinnvoll folgende Informationen zu erfassen:

- Alter, Geschlecht, Domäne und Ort des Benutzers: Diese können im Context oder User Service von VizBoard gespeichert sein. Mit Hilfe der Domäne soll fachabhängigen Fehlinterpretationen beigekommen werden, beispielsweise falls Mathematiker dazu tendieren Pfeile verkehrt herum zu deuten. Falls Menschen in Asien Icons anders interpretieren sollten, kann der Ort des Benutzers das Hilfesystem darauf hinweisen.
- Zeit und Datum der Interaktion: Sie sind nötig um Tag und Nacht unterscheiden zu können. Menschen sind bei Nacht eher müde und wenig konzentriert, benötigen daher früher Hilfe.
- Für die letzten 5 Interaktionen des Benutzers: Grafische Repräsentation der Komponente, Aktion und vergangene Zeit seit Nutzungsstart. Je genereller die gespeicherten Informationen, desto besser für das Hilfesystem, welches die Erkenntnisse in umso mehr Komponenten verwenden kann.

- Ob Hilfe angeboten wurde und wenn ja, welche Funktion: Diese Information ist nötig, um Fehlinterpretationen des Hilfesystems zu erkennen (Abbildung 3.33).
- Ob Hilfe aufgerufen wurde und wenn ja, welche Funktion: Dieses Attribut soll durch den Decision Tree vorhergesagt werden.

		<b>Hilfe angenommen</b>	
		Nein	Ja
<b>Hilfe angeboten</b>	Nein	true negative	false negative
	Ja	false positive	false positive true positive

Abbildung 3.33: Erkennung von False Positives und False Negatives

Das Hilfesystem schickt alle fünf Interaktionen einen solchen Datensatz an die Wissensbasis, welche ihn zu den Trainingsdaten hinzufügt. Außerdem wird der Datensatz an den Classifier weitergegeben, der das letzte Attribut vorhersagt. Ist das Ergebnis ein anderes als »Keine Hilfe nötig«, bietet das Hilfesystem dem Benutzer Hilfe zur entsprechenden Funktion an. Damit die Wissensbasis einen initialen Classifier generieren kann, müssen zunächst auf die beschriebene Art und Testdaten gesammelt werden. Mit diesen kann die Wissensbasis einen ersten Classifier generieren und iterativ verbessern.

### 3.9.3 User Interface und Interaktion

Der Button für die Meta-Hilfe befindet sich in der rechten oberen Ecke des Viewports (Abbildung 3.34). Er ist rot, damit er vom Benutzer bemerkt wird und rund, weil er so aus dem bisher verwendeten optischen Konzept ausbricht und auffällt, aber nicht fehl am Platz wirkt.

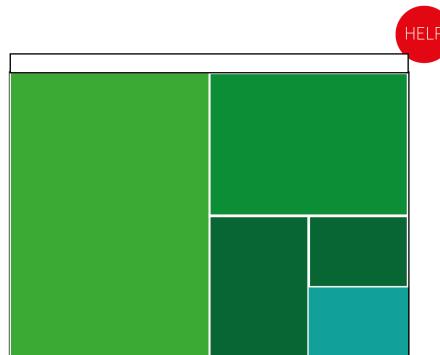


Abbildung 3.34: UI Mockup: Statischer Button für Meta-Hilfe

Nachdem der Benutzer darauf geklickt hat, wird der Viewport abgedunkelt und ein Fenster überblendet, welches Hilfe zu verschiedenen Themen enthält (Abbildung 3.35). Von dort können ebenfalls die verschiedenen Hilfefunktionen (wie z. B. das Intro) aufgerufen werden.

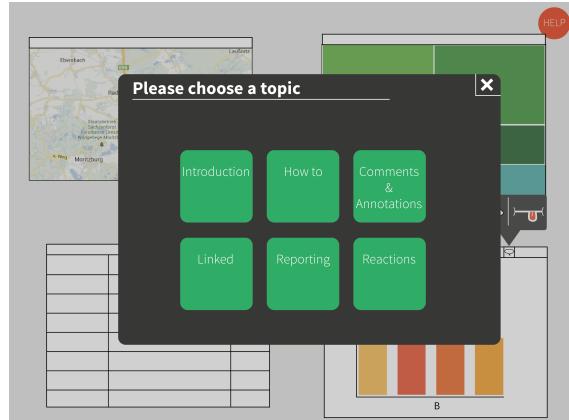


Abbildung 3.35: UI Mockup: Statische Meta-Hilfe

Wenn das Hilfesystem dem Benutzer Hilfe anbietet, rotiert der Button, wird orange eingefärbt und in der Sidebar erscheint das entsprechende Hilfeangebot (Abbildung 3.36). Der Benutzer kann es nutzen, indem er darauf klickt oder selbstständig zur Hilfefunktion navigiert. Tut er das für 5 Interaktionen nicht oder schließt die Sidebar, wird der Button zurück rotiert.

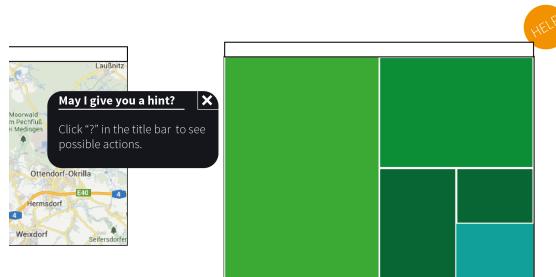


Abbildung 3.36: UI Mockup: Dynamische Meta-Hilfe

## 3.10 Synthese

Für das Konzept wurde das Markup einer Komponente, die Komponentenbeschreibung sowie die API einer Komponente, das CoRe und der Event Broker erweitert. In diesem Abschnitt werden die Änderungen zusammengefasst.

### Markup einer Komponente

- Datenattribute (Legende und Freitext), um Begriffe zur automatischen Verlinkung zu kennzeichnen.

- Datenattribut, um Wurzelement der Visualisierung zu kennzeichnen (Kommentare, History).
- RDFa Attribute, die Datenpunkte kennzeichnen und eventuell beschreiben (Kommentare).

## Komponentenbeschreibung

- Annotation, ob eine Property Teil des Mementos ist (Kommentare, History).
- VISO Operationen, welche eine Aktion ausführen (Bedienung).
- CSS Selektoren für jede Operation, um UI Elemente für Operation zu kennzeichnen (Bedienung).
- Konzept, welches eine Aktion näher beschreibt (Verlinkung).
- Anzahl an Sekunden, die zur vollständigen Änderung des User Interfaces benötigt werden (Bedienung).
- Nähere Beschreibung einer Aktion durch Freitext oder Referenz auf DatenvARIABLE (Bedienung).
- Ob Hilfe für Capability/Operation generiert werden soll oder nicht (Bedienung).
- Optionale Zusatzinformationen zur VISO Operation, beispielsweise welche Taste gedrückt werden muss (Bedienung).
- Beispieldaten für Parameter, falls von VISO Operation benötigt (Kommunikation). Das ist beispielsweise bei einer Drag Operation nötig.
- Testdatensatz: Entweder ID eines DaRe Datensatzes mit SPARQL Query für Mappings oder Informationen über einen hochgeladenen Datensatz (Bedienung).
- Ob die Komponente Datenverarbeitung durchführt (Kommentare).
- Ob die Komponente externe Daten lädt (Kommentare).
- Ob die Komponente beliebig skalierbar ist oder deren CSS zur Laufzeit transformiert werden muss (History).

## API einer Komponente

- Eine Methode, um Koordinaten in Datenpunkte umzurechnen (Kommentare).
- Die Umkehrung der Methode (Datenpunkte → Koordinaten) (Kommentare).
- Eine Methode, welche Attribute zu URIs von Datenpunkten ausgibt (Kommentare).
- Eine Methode, welche Testdaten in die Komponente lädt (Bedienung).

## VISO

- Beschreibung einer grafischen Repräsentation (Intro).
- Link zu Tutorialvideo (Intro).

## CoRe

- Ursprüngliches Mapping an MRE weiterleiten (Intro, Kommentare).
- Wissensbasis für dynamische Meta-Hilfe sowie Schnittstelle zum Zugriff darauf.

## DaRe

- Kommentare speichern sowie eine Schnittstelle zum Abruf.
- Abruf und Speicherung von Konzeptbeschreibungen sowie Schnittstelle zum Zugriff (Bedienung, Verlinkung).

## Event Broker

- Methode, welche Weiterleitung von Events pausiert (Kommentare).
- Methode, welche Weiterleitung von Events wieder aufnimmt (Kommentare).

## Event Nachricht

Für die History müssen folgende Attribute enthalten sein:

- Aufrufende Operation,
- Timestamp,
- eindeutige ID, beispielsweise eine fortlaufende Nummer.

## Context Service

- Alter, Aufenthaltsort, Geschlecht, Domäne des Benutzers (Kommentare, Meta-Hilfe).

## CRUISe

- Jedes Event muss eine Nachricht schicken (History).

## 3.11 Zusammenfassung

In diesem Kapitel wurde ein Hilfesystem für InfoVis Mashups vorgestellt, welches großteils auch für andere, auf CRUISe basierende Mashups eingesetzt werden kann. Es besteht aus folgenden acht Teilkonzepten:

1. Intro in die Funktionalität einer Komponente (Abschnitt 3.2)
2. Bedienung einer Komponente (Abschnitt 3.3)
3. Feedback zu einer Komponente (Abschnitt 3.4)
4. Kommunikation zwischen Komponenten (Abschnitt 3.5)
5. Verlinkung von unbekannten Konzepten (Abschnitt 3.6)
6. Kommentare in Visualisierungen (Abschnitt 3.7)
7. History (Abschnitt 3.8)
8. Meta-Hilfe (Abschnitt 3.9)

Das Intro in die Funktionalität einer Komponente befasst sich damit, dem Benutzer eine grobe Einführung in eine Komponente zu geben. Er soll danach wissen, welche Daten wie dargestellt werden und wie er die Visualisierung interpretieren soll. Zu diesem Zweck werden nacheinander für jede Komponente die Art der Visualisierung (z. B. Balkendiagramm), eine Beschreibung, die enthaltenen Daten, ein Tutorialvideo (wenn vorhanden) und grundlegende Informationen über die Kommentarfunktion sowie ein Hinweis auf die statische Meta-Hilfe angezeigt. Die notwendigen Informationen dafür kommen aus der SMCDL beziehungsweise dem Domain Mapping, sodass das Hilfesystem dem Anwender einheitliche Bezeichnungen und Erklärungen zur Verfügung stellen kann. Für den Entwickler entsteht kein zusätzlicher Aufwand, da die notwendigen Konzepte (wie die Art der Visualisierung) schon jetzt in der SMCDL vorhanden sein müssen.

Die Hilfe zur Bedienung soll dem Benutzer erklären, welche Aktionen in einer Komponente verfügbar sind (z. B. Pan, Zoom, Sortieren) und wie diese ausgeführt werden können. Die Erklärungen werden in Form eines Comics mit drei Panels dargestellt. Das erste Panel zeigt die UI Elemente, welche die Aktion auslösen. Das Zweite informiert den Benutzer darüber, welche Operation (z. B. klicken) auf diesen Elementen nötig ist. Im dritten Panel ist schließlich das Resultat der Aktion zu sehen, beispielsweise eine sortierte Tabelle. Die Bilder für die jeweiligen Panels werden nach der Registrierung einer Komponente automatisch von einem Headless Browser gerendert und über die bestehende SOAP API des Component Repositorys dem Frontend zur Verfügung gestellt. Um eine Komponente in die Bedienungshilfe zu integrieren, muss der Komponentenentwickler die entsprechenden Aktionen und Operationen in der SMCDL annotieren.

Das Feedback zu einer Komponente soll dem Anwender eine Möglichkeit geben, entweder Fehler in einer Komponente zu melden oder diese allgemein zu bewerten. Keine Software ist frei von Bugs und im schlimmsten Fall findet sie der Nutzer. Auch ist es möglich, dass externe APIs sich ändern und Komponenten nicht entsprechend

aktualisiert wurden. Für den Komponentenentwickler entsteht kein Aufwand, da diese Funktion ohne besondere Informationen über die Komponente umgesetzt werden kann.

Die Hilfestellung zur Kommunikation zwischen Komponenten soll sichtbar machen, wann welche Komponenten untereinander Nachrichten austauschen und außerdem die vorhandenen Links im Communication Model erklären. Um ersteres zu erreichen, wird bei jeder gesendeten Nachricht ein Pfeil zwischen die jeweiligen Komponenten gezeichnet, der nach kurzer Zeit wieder verschwindet. Für zweiteres werden die Links zusätzlich zur Bedienungshilfe angezeigt, nachdem diese aufgerufen wurde. Die Hilfe, welche nun den Zusammenhang zwischen Quelle und Ziel des Links erklärt, ist ähnlich der Bedienungshilfe. Sie besteht aus zwei Panels, wobei im ersten das Resultat der Aktion in der Quellkomponente und im zweiten das Resultat der CRUISe Operation in der Zielkomponente zu sehen ist. Alle hierfür notwendigen Informationen müssen bereits in der Bedienungshilfe angegeben werden, sodass für den Komponentenentwickler kein zusätzlicher Aufwand entsteht.

Unbekannte Konzepte sollen verlinkt werden, da nicht davon ausgegangen werden kann, dass der Anwender über alle Details Bescheid weiß. Ein Beispiel: Viele wissen, was mit dem BIP angegeben wird, aber nicht annähernd so viele können genau sagen, was dort mit eingerechnet wird und was nicht. Um diese Hilfe umzusetzen, muss der Komponentenentwickler die entsprechenden Konzepte in Form einer URI zu einer externen Wissensbasis an die gewünschten UI Elemente - beispielsweise eine Legende - mit Hilfe eines HTML Datenattributs annotieren oder in der SMCDL angeben. Das Data Repository lädt die Beschreibung eines Konzepts, nachdem die Komponente im CoRe registriert wurde. Das Hilfesystem kann danach die Beschreibung zur Laufzeit vom DaRe laden.

Um es dem Benutzer zu erleichtern in Daten vorhandene Informationen aufzunehmen, werden Kommentare eingesetzt. Sie bestehen aus einem Kommentartext und einer beliebigen Anzahl von Annotationen, wobei diese eine Selektion von Datenpunkten, Text, Pfeile oder ein Rechteck sein können. Benutzer können Kommentare genau einmal bewerten und es so anderen ermöglichen, die wichtigsten Informationen schnell zu finden. Wenn die Komponente mindestens eine Datatype Property anhand der Position, Breite oder Höhe visualisiert, können Annotationen an den Daten selbst gespeichert und in anderen Komponenten wiederverwendet werden. Damit die Annotationen überall korrekt angezeigt werden, muss der Komponentenentwickler Methoden implementieren, welche zwischen Daten und Koordinaten in der Visualisierung umsetzen.

Die History ermöglicht dem Anwender zwischen verschiedenen Zuständen des Mashups zu wechseln. Er interagiert nie isoliert mit einer Komponente, da diese über Links mit anderen verbunden ist. Deswegen müssen Änderungen, die von einer Nutzeraktion ausgingen, komponentenübergreifend zurückgesetzt werden. Der Komponentenentwickler muss hierfür nur darauf achten, dass jedes Event eine Nachricht zum Event Broker sendet und diese Nachricht die aufrufende CRUISe Operation enthält. Zusätzlich wäre es allerdings hilfreich, wenn das CSS der Komponente ausschließlich relative Einheiten verwendete, da das Hilfesystem dann ohne zu großen Aufwand eine visuelle State History anzeigen kann.

Die sieben beschriebenen Hilfefunktionen bieten zwar einheitliche Interaktionen und Bezeichnungen, bringen aber selbst zusätzliche Komplexität in die Anwendung,

weswegen eine Meta-Hilfefunktion nötig ist. Diese besteht einerseits aus einer statischen Variante, welche die verschiedenen Hilfefunktionen mit Textbeschreibungen und Bildern erklärt. Andererseits wird auch eine dynamische Version eingesetzt. Sie bestimmt mit Hilfe eines Statistical Machine Learning Algorithmus (Decision Tree) auf Basis von Informationen über Nutzerinteraktionen und dem Nutzer selbst, wann welche Hilfefunktion vorgeschlagen werden soll. Für den Komponentenentwickler entsteht hier kein Aufwand, da alle wichtigen Informationen aus den Modulen der TSR gelesen werden können.

# 4 Implementierung

In den folgenden Abschnitten wird die prototypische Implementierung der Konzepte Bedienung (Abschnitt 3.3), Kommunikation (Abschnitt 3.5) und Kommentare (Abschnitt 3.7) beschrieben. Sie sind aufgeteilt in Frontend (Abschnitt 4.1) und Backend (Abschnitt 4.2).

## 4.1 Frontend

Das Frontend der implementierten Konzepte wurde mit Backbone.js<sup>1</sup>, einem populären Javascript MVC Framework, und dessen Erweiterung Marionette.js<sup>2</sup> geschrieben. Des Weiteren kamen die Bibliotheken jQuery<sup>3</sup> (von Backbone verwendet), jQuery UI<sup>4</sup> und D3 [BOH11] zum Einsatz.

Zunächst muss begründet werden, warum ein anderes Javascript Framework als das in CRUISe bereits eingesetzte ExtJS<sup>5</sup> gewählt wurde. Hier müssen zwei Einsatzzwecke unterschieden werden:

1. Als MVC Framework, welches verschiedene Views und Models definiert und strukturiert, sowie
2. als Bibliothek für generische DOM Operationen.

Für letzteres wird jQuery eingesetzt, da ExtJS mehr vom DOM abstrahiert als jQuery. Mit ExtJS denkt der Entwickler nicht in `divs` oder `spans`, sondern in generischen Komponenten wie Listen, Panels und Dialogen. Nicht alle UI Konzepte können damit umgesetzt werden. Beispielsweise werden für Kommentare an einzelnen Ressourcen eine Art Badge angezeigt, was einem `div` mit runden Ecken entspricht. Davon abgesehen ist jQuery bereits in der TSR Codebase vorhanden.

Als MVC Framework wird Backbone unter anderem deswegen eingesetzt, weil in der Einführung des Konzepts (Abschnitt 3.1) ein visueller Unterschied zwischen Hilfesystem und Komponenten gefordert wurde. CRUISe Komponenten verwenden bereits die generischen ExtJS Styles, weswegen das Frontend eigene Styles für das recht komplexe, von ExtJS generierte Markup definieren müsste. Dies erschien zu viel Aufwand zu sein. Von der Einheitlichkeit der Codebase abgesehen gab es auch keinen Grund, ExtJS zu verwenden, weil die UI des Hilfe Frontends recht einfach aufgebaut ist.

---

<sup>1</sup><http://backbonejs.org/>

<sup>2</sup><http://marionettejs.com/>

<sup>3</sup><http://jquery.com/>

<sup>4</sup><http://jqueryui.com/>

<sup>5</sup><http://www.sencha.com/products/extjs>

Nun stellt sich noch die Frage, warum gerade Backbone.js und nicht eines der vielen anderen Frameworks<sup>6</sup> gewählt wurde. Einerseits hat der Autor damit bereits grundlegende Erfahrung gesammelt, sodass die Einarbeitungszeit entfiel. Andererseits ist es gut dokumentiert und sehr populär, weswegen im Internet viele Informationen zu finden sind.

Marionette.js erweitert Backbone um verschiedene Konzepte (z. B. Regionen und unterschiedliche Views für Items, Collections und Bäume) und automatisiert oft vorkommende Anweisungen, wie zum Beispiel einen View zu rendern und in den DOM einzufügen. Der Code ist dadurch kürzer, lesbarer und einfacher zu warten.

D3 wird hauptsächlich beim Hinzufügen von Annotationen eingesetzt, wo ein SVG Element über die Komponente gelegt wird, in welchem der Benutzer seine Annotationen zeichnen kann. Außerdem kann D3 im Gegensatz zu jQuery sowohl mit HTML als auch mit SVG Elementen umgehen. Einige Methoden, die im Frontend gebraucht werden, unterstützt jQuery nicht für DOM Elemente mit SVG Namespace (Beispiele dafür sind `.width()`, `.addClass()` oder jQuery UI's `.position()`). Zwar existiert ein SVG Plugin<sup>7</sup> für jQuery, dieses weist aber eine unnötig komplizierte API auf. Deswegen wird für die Größenbestimmung oder Positionierung von SVG Elementen D3 verwendet, da es ohnehin schon eingesetzt wird.

#### 4.1.1 Clientseitige Architektur

Das Assistance Subsystem arbeitet unabhängig vom Rest der Thin Server Runtime (Abbildung 4.1). Nachdem alle Komponenten integriert wurden, wird es vom Application Manager falls erwünscht gestartet.

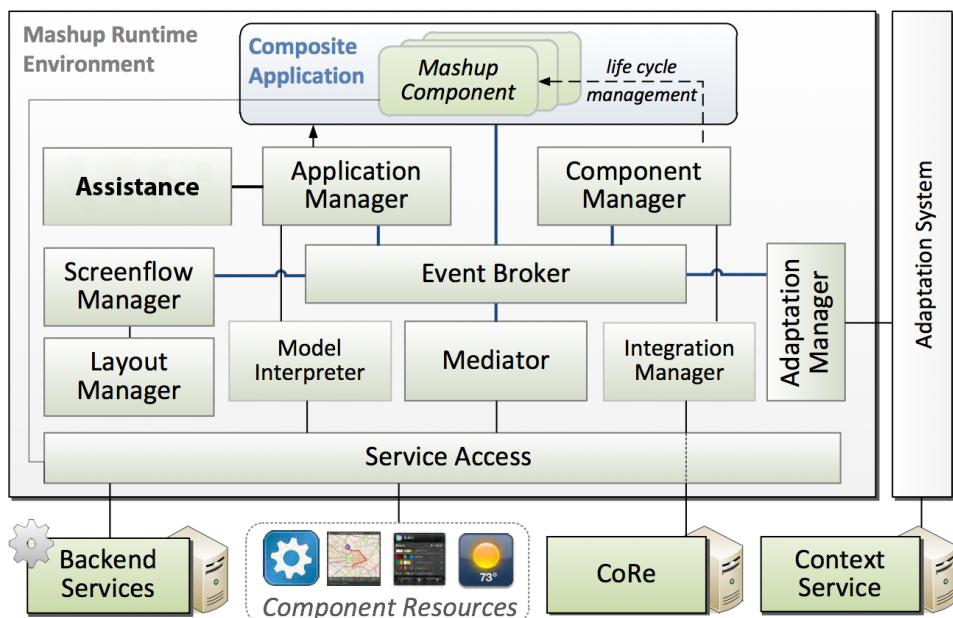


Abbildung 4.1: Architektur von CRUISe mit Assistance

<sup>6</sup>Für eine Übersicht der Wichtigsten sei auf <http://todomvc.com/> verwiesen, wo 33 Javascript MVC Frameworks aufgelistet sind.

<sup>7</sup><https://github.com/kbwood/svg>

Um es zu konfigurieren, wird beim Start ein Objekt mit folgenden Informationen übergeben:

- `comment_url`: Die URL zum Kommentar Backend (Abschnitt 4.2.1).
- `core_url`: Die URL zum CoRe Service.
- `dataset_id`: Die ID des geladenen Datensatzes, notwendig für Kommentare.
- `user_id`: Die ID des angemeldeten Benutzers.
- `user_name`: Der Benutzername des angemeldeten Nutzers.
- `components`: Ein Array mit Informationen zu jeder Komponente.

Die notwendigen Informationen zu einer Komponente sind folgende:

- `component_id`: Die URI der Komponente.
- `component`: Die ID des DOM Elements, in dem sich die Komponente befindet.
- `visualization`: Die ID des DOM Elements, welches die Visualisierung enthält.
- `visualized_propertys`: Ein Array der dargestellten Propertys.
- `data_annotations_enabled`: Ob datenbezogene Bereichskommentare zulässig sind.
- `reference`: Eine Referenz zur Instanz der Komponente, um die API aufrufen zu können.
- `memento`: Ein Array der Propertys, welche das Memento der Komponente ausmachen.
- `capabilities`: Ein Array von Objekten mit Informationen über jede Capability.

Für jede Capability müssen folgende Informationen angegeben werden:

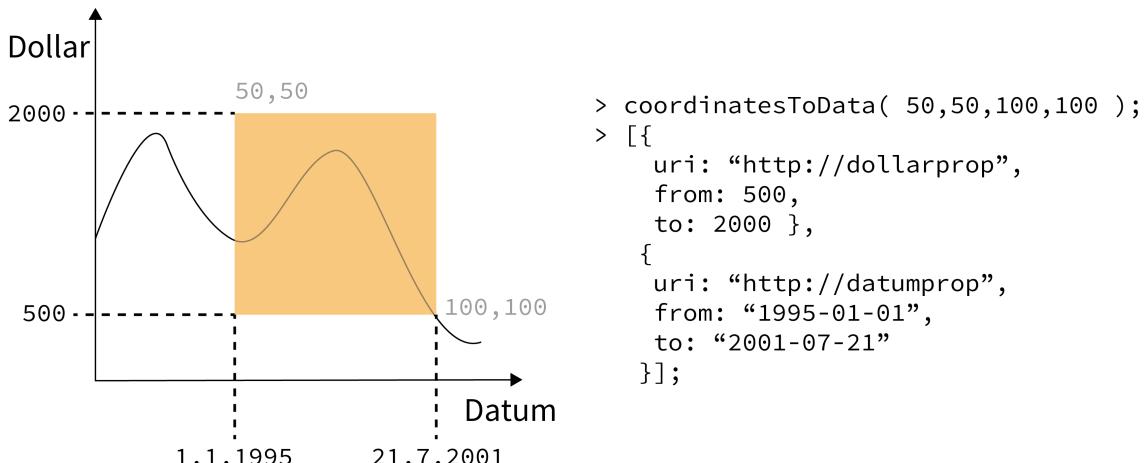
- `capability`: Die ID der Capability.
- `action`: Ein Label, welches die Capability beschreibt, z. B. »sort a movie (Studio)«.
- `component`: Die ID des DOM Elements, in dem sich die Komponente befindet.
- `operation`: Die ID einer atomaren VISO Operation dieser Capability.
- `elements`: Der CSS Selektor, welcher die DOM Elemente für diese Capability findet.

Alle diese Informationen werden vom ApplicationManager aus den SMCDLs der Komponenten gelesen, nachdem diese erfolgreich integriert wurden. Um festzustellen, ob eine Komponente datenbezogene Bereichskommentare unterstützt, wird ein Request ans CoRe gesendet, der die Kodierung der visuellen Attribute aller Propertys zurückgibt. Findet sich darunter mindestens einmal die Position oder Breite beziehungsweise Höhe – also etwas, was sich entlang der X- oder Y-Achse messen lässt – so werden die angesprochenen Kommentare unterstützt. Für die Generierung eines passenden Labels wird der LabelGenerator von EDYRA herangezogen.

### 4.1.2 API einer Komponente

Wie in der Konzeption (Abschnitt 3) erläutert, muss die API einer Komponente um zusätzliche Funktionalität erweitert werden, um Kommentare zu unterstützen und Hilfegenerierung möglich zu machen. Das ist hauptsächlich deswegen nötig, weil die Komponente eine »Black Box« ist und das Hilfesystem keine Annahmen über die Art der Darstellung von Informationen machen kann. Allerdings sind die meisten Methoden optional: Möchte der Komponentenentwickler keine Kommentare unterstützen, sollen sie einfach nicht implementiert werden.

- `loadTestdata( data )` lädt Testdaten in die Komponente. Diese Methode muss implementiert werden, sofern die Komponente Bedienungshilfe für irgendeine Capability zur Verfügung stellen soll. Die Testdaten werden als String übergeben, um kein Serialisierungsformat (z. B. JSON oder XML) vorzuschreiben. Das Parsen des Strings ist der Komponente überlassen. Nachdem in der Regel bereits eine interne Methode besteht, die Daten in die Komponente lädt, stellt die Implementierung von `loadTestdata` kaum Mehraufwand dar.
- `coordinatesToData( x1, y1, x2, y2 )` wandelt das durch die vier Koordinaten angegebene Rechteck in Grenzwerte von Propertys um. Dabei wird ein JSON Array von Objekten mit Attributen `uri`, `from` und `to` zurückgegeben. Für jede in der Komponente dargestellte Property, welche mit den visuellen Attributen Breite, Höhe oder Position kodiert ist, findet sich ein durch die URI gekennzeichnetes Objekt im Array (siehe Abbildung 4.2). Weist eine Property ein nominales Scale of Measurement auf, sind `from` und `to` identisch und enthalten ein Array von möglichen Werten.
- `dataToCoordinates( p1, p2 ... )` wandelt von `coordinatesToData` erhaltenen datenbezogenen »Grenzpunkte« wieder in Koordinaten um. Dazu liefert die Methode ein Objekt mit den Attributen `x1`, `x2`, `y1` und `y2` zurück.
- `getVisualizedpropertys()` liefert ein Array mit den URIs der in der Komponente visualisierten Propertys zurück. Das ist notwendig, damit das Hilfesystem Kommentare für Komponenten laden kann. Diese Propertys bestehen zur Zeit aus der Kombination von Datensatz ID, URI der Klasse der Property (`rdfs:domain`) und URI der Property. So sind sie eindeutig identifizierbar.
- `getDataFromResource()` gibt die Werte der dargestellten Propertys eines Datenpunkts zurück.

Abbildung 4.2: Die Methode `coordinatesToData`

Außerdem wurde das Markup einer Komponente entsprechend erweitert. Die Visualisierung in der Komponente wird mit einem Datenattribut `data-vizboard-visualization` gekennzeichnet. Das Visualisierungselement muss genau so groß sein wie der dargestellte Datenbereich und die Repräsentationen der Datenpunkte enthalten. Abbildung 4.3 zeigt ein Beispiel. Damit das Hilfesystem die Repräsentationen der Datenpunkte finden kann, wird deren kodierte URI mit dem RDFa Attribut `resource` annotiert (siehe Listing 4.1).

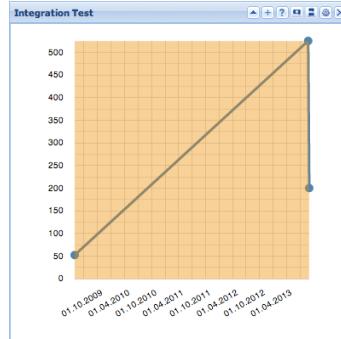


Abbildung 4.3: Visualisierungselement in einem Liniendiagramm

```

<svg>
  <g data-vizboard-visualization="true">
    <rect class="bg" x="0" y="0" width="100" height="100" />
    <circle class="datapoint" cx="0" cy="90" resource="http%3A%2F%2
      Fsomeuri1" />
    <circle class="datapoint" cx="100" cy="0" resource="http%3A%2F%2
      Fsomeuri2" />
  </g>
</svg>

```

Listing 4.1: Markup einer Visualisierung

### 4.1.3 Kommentare

Im Folgenden wird auf die Implementierung der Kommentare (Abschnitt 3.7) eingegangen. Sie erlauben es Benutzern, Erkenntnisse untereinander auszutauschen.

#### Kommentar schreiben

Kommentare werden über Buttons in der Titelleiste einer Komponente aufgerufen. Unterstützt eine Komponente keine Kommentare, sind die Buttons inaktiv. In Abbildung 4.4 ist zu sehen, wie ein Kommentar verfasst wird. Um den Bereich um die Visualisierung herum abzudunkeln, wird ein SVG Element in die Komponente eingefügt. Es enthält eine Maske<sup>8</sup>, welche die Visualisierung freistellt.

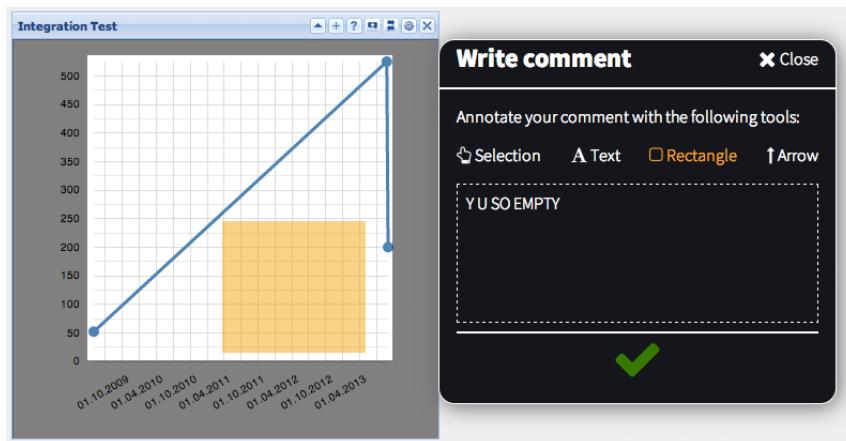


Abbildung 4.4: User Interface um einen Kommentar zu verfassen

Auf diesem SVG Element kann der Benutzer wie in gängigen Zeichenprogrammen üblich Rechtecke und Pfeile zeichnen. Um Text hinzuzufügen, muss er zuerst auf den Visualisierungsbereich klicken und damit die Koordinaten festlegen, an denen der Text erscheinen soll. Danach öffnet sich ein Texteingabedialog und nachdem der Benutzer bestätigt hat, wird der Text ins SVG Element gezeichnet. Der Text kann später editiert oder wie alle Annotationen auch gelöscht werden.

Selektionen verhalten sich etwas anders als Text, Rechtecke und Pfeile, da sie nicht mit SVG Elementen dargestellt werden. Um zu bestimmen, wann sich die Maus über einem Datenpunkt befindet, wird ein `mousemove` Event Handler eingesetzt. In diesem werden die aktuellen Mauskoordinaten sowie alle Bounding Boxes der Datenpunkte abgefragt und dann verglichen, ob sich die Koordinaten innerhalb einer Bounding Box befinden. Analog dazu arbeitet der `click` Handler, welcher Datenpunkte selektiert oder deseletktiert. Die Event Handler können nicht auf den Datenpunkten selbst registriert werden, weil diese keine Mausevents auslösen, da sich das SVG Element über ihnen befindet. Das hat allerdings den praktischen Nebeneffekt, dass vom Entwickler vorgesehene Mausevents in der Visualisierung ebenfalls nicht ausgelöst werden, solange die »Kommentar schreiben« UI aktiv ist.

Nachdem der Benutzer den Kommentar bestätigt hat, werden die Koordinaten und Dimensionen der Annotationen in relative Einheiten (zur Größe der Visualisie-

<sup>8</sup>[https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Clipping\\_and\\_masking](https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Clipping_and_masking)

rung) umgerechnet und an das Kommentar Backend (siehe Abschnitt 4.2.1) gesendet.

## Kommentare lesen

Um Kommentare zu lesen, muss erst ein Button in der Titelleiste der jeweiligen Komponente betätigt werden. Danach wird das entsprechende User Interface neben der Komponente angezeigt (Abbildung 4.5).

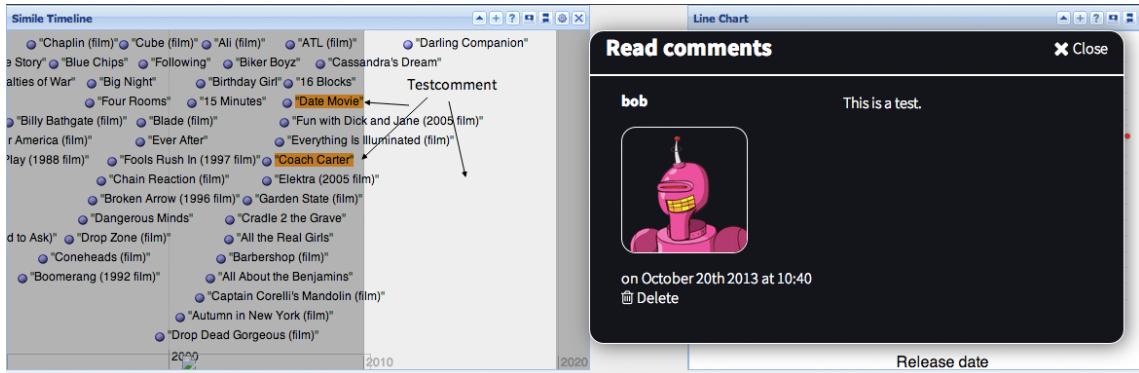


Abbildung 4.5: User Interface um Kommentare zu lesen

Die Annotationen sind verschiedene DOM Elemente, welche bei Bedarf angezeigt oder versteckt werden. Flächenbasierte Annotationen werden als SVG `rects`, `lines` und `texts` umgesetzt, deren Position relativ zur Größe der Visualisierung ist (siehe vorheriger Abschnitt).

## Einschränkungen

Die Lösung, `mouseover` über Datenpunkten mit Hilfe der Bounding Box festzustellen funktioniert zwar, hat aber einen Nachteil: Da die Bounding Box immer rechteckig ist, können »false positives« auftreten (Abbildung 4.6). Variante A ist der Idealfall; die Bounding Box ist gleich groß wie die Repräsentation des Datenpunkts. Das ändert sich allerdings, wenn die Repräsentation transformiert wird (45° Rotation in Variante B) oder eine nicht-eckige Form hat (Kreis in Variante C).

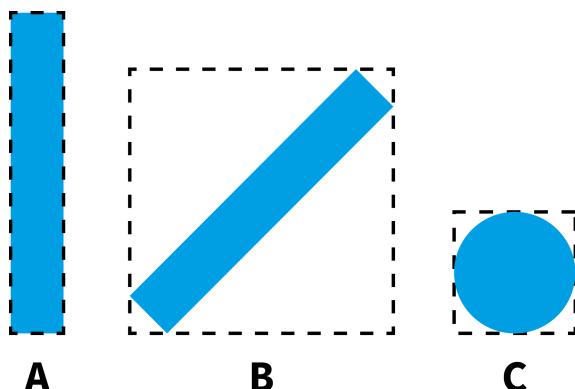


Abbildung 4.6: Genauigkeit der Bounding Box eines Elements

Eine Möglichkeit, um diese Einschränkung zu umgehen, ist eine weitere Funktion in die API einer Komponente hinzuzufügen. Sie teilt dem Hilfesystem mit, ob ein Punkt  $(x, y)$  innerhalb eines Datenpunkts liegt oder nicht. Diese Funktion kann optional genutzt werden: Ist sie implementiert, verwendet sie das Hilfesystem. Ansonsten arbeitet es wie bisher mit Bounding Boxes.

Flächenbasierte Annotationen werden relativ zur Größe der Visualisierung positioniert. Das kann problematisch sein, wenn die Visualisierung genau so groß wie die Komponente ist und diese in unterschiedlicher Größe gerendert wird (siehe Abbildung 4.7).

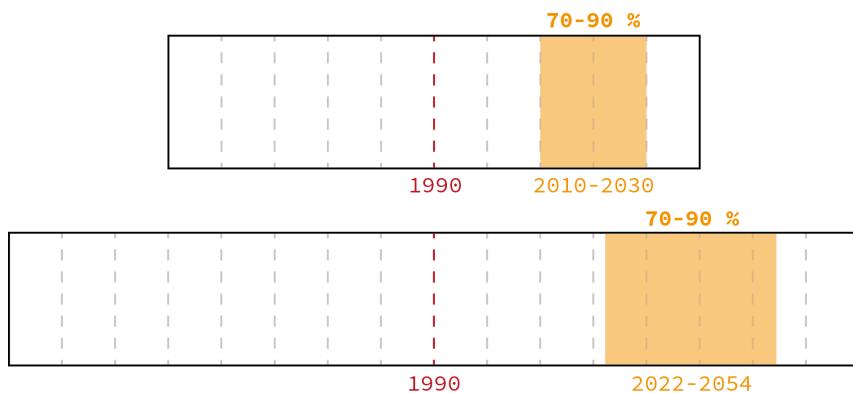


Abbildung 4.7: Problem von flächenbasierten Annotationen bei unterschiedlicher Visualisungsgröße: Das Zentrum ist in beiden Visualisierungen das Jahr 1990, in der unteren ist aber ein vollkommen anderer Bereich markiert.

Das Problem an sich kann nicht wirklich umgangen werden, da die Komponenten »Black Boxes« sind und unbekannt ist, wie sich der Inhalt einer Visualisierung mit unterschiedlicher Größe ändert. Deswegen sollten Komponenten so weit wie möglich datenbasierte Annotationen unterstützen und, falls es aufgrund der visuellen Darstellung nicht möglich ist, den Inhalt einer Visualisierung proportional skalieren. Möglicherweise könnte das Problem gelöst werden, indem die Größenangaben nicht relativ, sondern absolut erfolgen.

#### 4.1.4 Bedienung

Dieser Abschnitt befasst sich mit der Bedienungshilfe (Abschnitt 3.3). Mit ihr lernen Benutzer, wie sie Aktionen in Komponenten ausführen können.

Dafür wird die Komponente zunächst abgedunkelt und für Aktionen notwendige UI Elemente hervorgehoben (Abbildung 4.8). Die Kreise unten links und oben rechts sind abgeschnitten, weil die Datenpunkte sich an der Grenze des dargestellten Datenbereichs befinden und von der SVG Maske abgeschnitten werden.

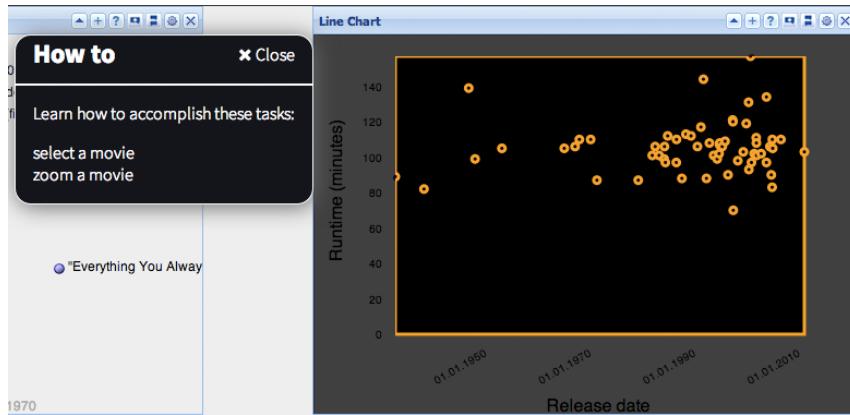


Abbildung 4.8: Assistance zur Bedienung, Schritt 1

Der Algorithmus, um alle eine Aktion auslösenden UI Elemente in ein anderes div zu kopieren, ist wie folgt:

1. Es wird ein teilweise transparentes, dunkles div in die Komponente eingefügt, um den dunklen Hintergrund herzustellen.
2. Danach wird ein weiteres div erstellt, die »Rootcopy«. Diese hat dieselbe Größe und Position wie die Komponente.
3. In die Rootcopy wird wiederum ein svg Element eingefügt, welches sie ausfüllt. Das hat den Grund, dass aktionsauslösende UI Elemente auch SVG rects oder ähnliches sein können. Sie werden vom Browser nur gerendert, wenn sie sich auch in einem svg befinden.
4. Damit Kopien in der Rootcopy möglichst gleich aussehen wie die Originale, werden in der Komponente gesetzte CSS Styles auf die Rootcopy übertragen.
5. Danach werden alle von einem CSS Selektor getroffenen Elemente entweder in die Rootcopy selbst oder ins svg kopiert und gleich wie die Originale positioniert. Sofern sich SVG Elemente nicht in transformierten Gruppen befinden, ist das durch die Kopie des DOM Elements schon geschehen, da ihr visueller Zustand im Markup gespeichert ist (z. B. in Form von x, transform oder width Attributen). Bei HTML Elementen werden die CSS Klassen mitkopiert. Das CSS kann allerdings Regeln enthalten, die von der Struktur des Markups abhängig sind, z. B. position: relative; width: 80%;. Da diese in der Rootcopy nicht dasselbe Ergebnis liefern, werden Breite und Höhe noch einmal explizit gesetzt.
6. Bei jedem durch Selektor  $s_i$  getroffenen Element und dessen Kopie wird außerdem ein Attribut `data-vizboard-copy` auf `copy<i>` gesetzt. Der Grund dafür ist die Verbindung zwischen Aktionen und UI Elementen: Bei `mouseover` über einer Aktion in der Liste werden alle dafür notwendigen UI Elemente hervorgehoben. Um diese zu finden, kann der Selektor nicht in der Rootcopy ausgeführt werden: Selektoren wie z. B. `:nth-child(odd)` würden falsche Ergebnisse liefern. Aus diesem Grund werden die Selektoren bei `mouseover` in

der Komponente ausgeführt und die Kopien der Elemente über den Wert von `data-vizboard-copy` gefunden (siehe Abbildung 4.9).

7. Die aktionsauslösenden UI Elemente müssen nicht leer sein, weswegen dort gesetzte Styles auch bei Kindknoten sichtbar sein könnten, z. B. wenn eine CSS Property auf `inherit` gesetzt ist. Aus diesem Grund werden bestimmte CSS properties (`border`, `color`, `background`) in den Kopien auf die Werte der Originale fixiert.
8. Aktionsauslösende UI Elemente können andere aktionsauslösende UI Elemente enthalten. Das wäre beispielsweise der Fall, wenn das anonyme `div` in Abbildung 4.9, welches die `div.bar` enthält, eine dritte Aktion auslösen würde. Damit diese nicht mehrfach in der Rootcopy vorhanden sind, wird für jedes Original einer Kopie überprüft, ob es Knoten mit dem Attribut `data-vizboard-copy` enthält. Falls ja, wird ein CSS Selektor aus dem Knoten generiert und alle davon getroffenen Elemente in der Kopie gelöscht.

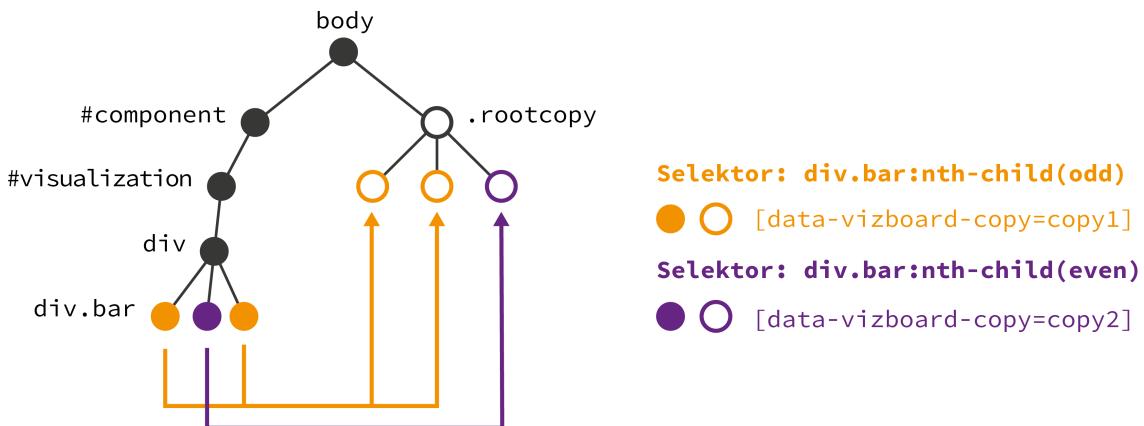


Abbildung 4.9: Die CSS Selektoren würden in der Rootcopy falsche Treffer liefern, da sowohl die Reihenfolge als auch die Elternknoten der DOM Elemente verändert sind. Deswegen werden Originale und Kopien über das Attribut `data-vizboard-copy` verknüpft und auffindbar gemacht.

## Einschränkungen

In diesem Abschnitt werden die Grenzen der gewählten Strategien zur Darstellung der Bedienungshilfe aufgezeigt. Zunächst ist es so, dass die Komponente nicht abgedunkelt werden kann, wenn das relevante UI Element zu groß ist. Die Styles zur Hervorhebung werden außerdem an der Wurzel des UI Elements gesetzt. Falls diese von Kindknoten überschrieben werden, ist kein Effekt sichtbar. Außerdem ist es möglich, dass die Komponente über ihrer abgedunkelten Kopie sichtbar ist, wenn das Attribut `z-index` entsprechend gesetzt ist. Um das zu verhindern, könnte den Komponentenentwicklern von CRUISe ein Maximalwert des Attributs vorgeschrieben werden, welcher nach der Registrierung einer Komponente überprüft wird. Dieser beträgt in den meisten Browsern 2.147.483.647 [Pui09], also könnte den Entwicklern immer noch ein Bereich von Null bis zwei Milliarden zur Verfügung gestellt werden.

Momentan wird der **z-index** von Kopien auf 0 gesetzt und von Originalen nicht verändert.

Ein weiteres Problem der vorgestellten und implementierten Bedienungshilfe ist Performance. War das Markup der Komponente tief verschachtelt, dauerte die Generierung von Kopien der aktionsauslösenden UI Elemente bis zu 15 Sekunden. Das war bei einer ExtJS Tabelle mit Testdaten von 60 Filmen in 6 Spalten der Fall, mit größeren Datensätzen würde es noch viel länger dauern. Nach einer Inspektion mit den Chrome Developer Tools stellte sich heraus, dass eine Kombination aus exzessivem »Layout Thrashing« und häufiger Garbage Collection das Problem war. Das Layout Thrashing hatte folgende Auslöser: Zunächst musste für jedes der Elemente die sichtbare Bounding Box berechnet werden, wofür Zugriff auf die Offsets nötig ist. Dafür wird ein aktuelles Layout benötigt, der Browser muss es also berechnen. Danach wird die Kopie in den DOM eingefügt, was das Layout invalidiert und es somit beim nächsten Element erneut berechnet werden muss. Zum Teil kann der Algorithmus bestimmt optimiert werden, aber nachdem Datensätze in VizBoard beliebig groß und Komponenten in CRUISe beliebig verschachtelt sein können, sollte auf diese Art der Darstellung verzichtet werden. Stattdessen sollten aktionsauslösende UI Elemente in ihrer normalen Kontext hervorgehoben werden, ohne aufwändige Erstellung von Kopien.

Auch die Darstellung von Panels unterliegt gewissen Einschränkungen. Abbildung 4.10 zeigt die Bedienungshilfe für ein Liniendiagramm. Positiv zu bewerten ist, dass die Bilder quadratisch sind und so die Panels ausfüllen. Allerdings ist das Ergebnis der Aktion (der Kreis ist orange gefüllt) im dritten Panel nur sehr schwer zu erkennen. Unter Umständen sollte doch wieder wie im Konzept ursprünglich vorgesehen, aber nach der formativen Nutzerstudie verworfen, ein viertes Panel am Anfang eingefügt werden. Dieses sollte den Ursprungszustand der Komponente darstellen und so ähnlich wie bei einem Suchbild zum Vergleich dienen. Eine andere Möglichkeit wäre die Panels zu vergrößern, was aber nur in Maßen möglich ist und nicht sehr viel Wirkung zeigen wird.

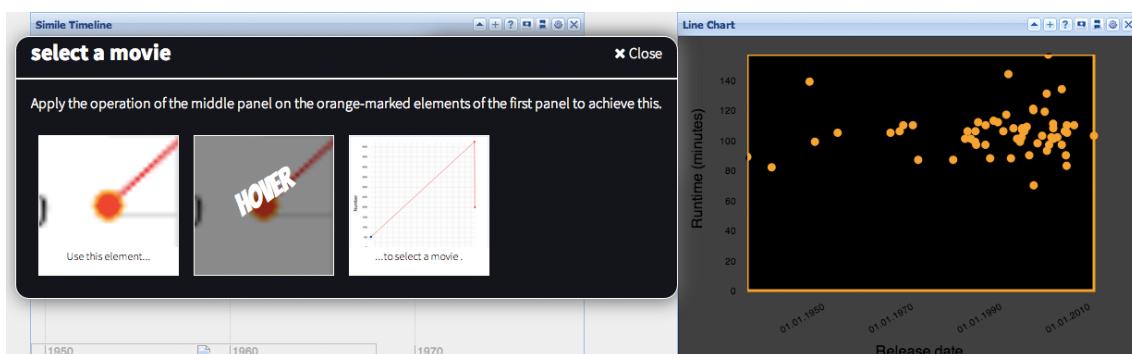


Abbildung 4.10: Assistance zur Bedienung, Liniendiagramm

In Abbildung 4.11 ist die SIMILE Timeline<sup>9</sup> als Negativbeispiel zu sehen. Das relevante Element ist so groß wie die Komponente selbst und hat mit mehr als 6.5 ein recht extremes Seitenverhältnis, weswegen sie das quadratische Panel nur unzu-

<sup>9</sup><http://www.simile-widgets.org/timeline/>

reichend ausfüllt. Die dargestellte Komponente ist außerdem so programmiert, dass sie ihren Zustand kaum merklich verändert.

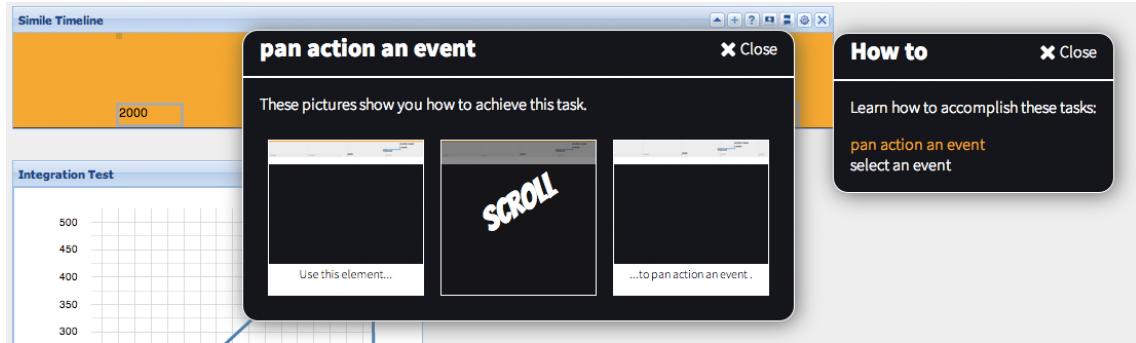


Abbildung 4.11: Assistance zur Bedienung, SIMILE Timeline

Eine mögliche Lösung wäre, das quadratische Panel immer zu füllen. Dann stellt sich allerdings die Frage, welcher Ausschnitt des Bilds gezeigt und wie dieser bestimmt und gefunden werden soll. Füllt man beispielsweise von links, sähe man in diesem Beispiel in allen Bildern hauptsächlich graue Fläche. Dem Problem könnte auch beigekommen werden, wenn die Panels das Seitenverhältnis ihrer Bilder annähmen. Allerdings ist das Layout der Panels dann unvorhersehbar. Ein Kompromiss könnte sein, die Panels quadratisch zu lassen und die Bilder füllend darzustellen, aber es dem Benutzer zu ermöglichen, innerhalb der Panels zu scrollen. Da aber hier die Interaktion etwas kompliziert würde, sollte zunächst ein passender Layoutalgorithmus versucht werden. Er sollte sowohl das Seitenverhältnis der einzelnen Bilder möglichst beibehalten, als auch einem Comic ähneln. Dass ein ansprechenderes Layout als aneinander gereihte Quadrate nötig ist, wurde schon in der formativen Nutzerstudie klar.

#### 4.1.5 Kommunikation

Im Folgenden wird auf die Hilfe zur Kommunikation zwischen Komponenten (Abschnitt 3.5) eingegangen. Sie ist im Prinzip eine Erweiterung der Bedienungshilfe (Abschnitt 4.1.4), da sie jeweils das Panel mit dem Ergebnis einer Aktion in einer Komponente zeigt. Die Kommunikationshilfe wurde in den CapView von EDYRA [RBM13] integriert, da dieser bereits Verbindungen zwischen Komponenten anzeigen kann (Abbildung 4.12).

Dazu wird das schwarze Fragezeichen in der Methode `drawPath` im Connection-Manager des CapView, welche die Verbindungen zwischen Capabilities von Komponenten zeichnet, erstellt und hinzugefügt. In Abbildung 4.13 ist die Kommunikationshilfe zwischen SIMILE Timeline und Liniendiagramm zu sehen.

## 4.2 Backend

Die folgenden Abschnitte behandeln das Backend des Prototypen. Sie adressieren Kommentare (Abschnitt 4.2.1) und Bedienung und Kommunikation (Abschnitt 4.2.2). Das Backend ist sowohl im CoRe (Bedienung/Kommunikation) als auch im DaRe (Kommentare) angesiedelt.

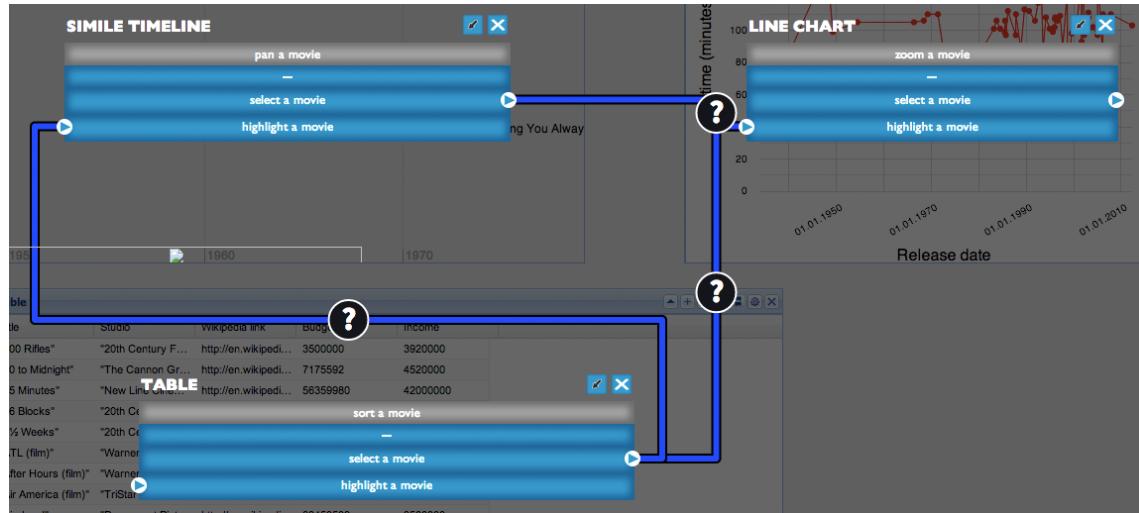


Abbildung 4.12: Erweiterter CapView

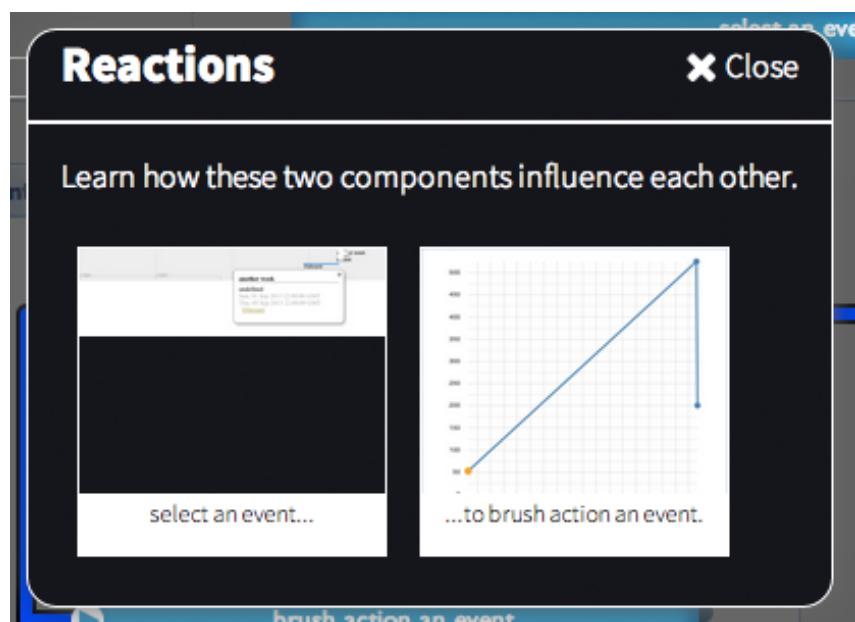


Abbildung 4.13: Assistance zur Kommunikation

### 4.2.1 Kommentare

Das Backend der Kommentare wurde ins Data Repository integriert, da dort bereits Datensätze gespeichert werden und eine entsprechende Infrastruktur existiert. Im Folgenden wird auf die Art der Persistierung, die Architektur und die REST API desselben eingegangen.

#### Repository

In der Konzeption der Kommentare (Abschnitt 3.7) wurde das Data Repository als geeigneter Speicherort identifiziert, da es bereits eine Infrastruktur zur Persistierung von Daten zur Verfügung stellt. Dieses enthält durch Metainformationen erweiterte semantische Datensätze, die im Frontend von Komponenten geladen und visualisiert werden können. Aus diesem Grund wurde für Kommentare ein eigenes Modell angelegt, sodass sie von »regulären« Daten getrennt sind.

Zwischen Front- und Backend wird ein Kommentar auf verschiedene Weise serialisiert (Abbildung 4.14). Das webbasierte Frontend kann JSON am besten verarbeiten, das DaRe Backend ist in Java geschrieben und arbeitet mit einem Jena TDB Triplestore, wo Kommentare letztendlich abgelegt werden. Zwar könnte das Backend ausschließlich zwischen JSON und RDF konvertieren, aber so ist es in Zukunft möglich bestehende Java-Bibliotheken für verschiedene Zwecke (wie z. B. Textklassifikation) einzusetzen.

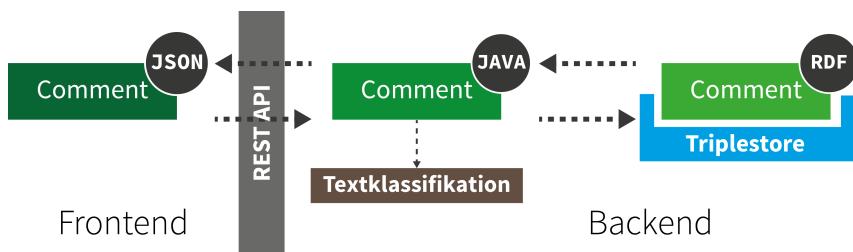


Abbildung 4.14: Verschiedene Serialisierungen eines Kommentars

Die verschiedenen Serialisierungsformate sind im Anhang zu finden (Abschnitte A.1.1, A.1.2, A.1.3).

#### Architektur

Die Architektur des Backends für Kommentare orientiert sich an den verschiedenen notwendigen Serialisierungen eines Kommentars (siehe vorhergehender Abschnitt). Sie besteht aus drei Klassen, die dem »Separation of Concerns« Prinzip folgend genau eine Zuständigkeit haben:

- Der **CommentRequestHandler** nimmt HTTP Requests entgegen, leitet sie an das CommentBackend weiter und antwortet mit JSON.
- Das **CommentBackend** stellt das Interface zum Triplestore dar, in dem die Kommentare abgelegt werden. Es gibt Kommentare in Java aus, sodass hier andere Java Bibliotheken ansetzen können.

- Der **CommentConverter** setzt zwischen verschiedenen Serialisierungen eines Kommentars (JSON, RDFS, Java) um.

Abbildung 4.15 zeigt die Architektur am Beispiel eines GET Requests des Frontends. Braun eingefärbte Kästchen stellen Java Klassen im Backend dar.

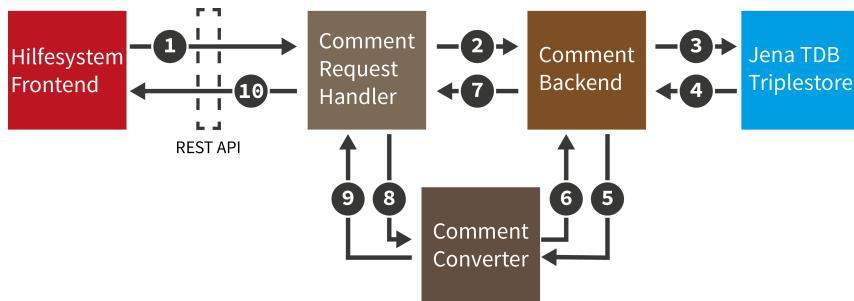


Abbildung 4.15: Architektur des Kommentar Backends

Folgende Schritte werden durchlaufen:

- Das Frontend sendet einen GET Request an das Backend, welcher vom Comment Request Handler behandelt wird.
- Der Request Handler ruft die Methode des Comment Backends auf, welche Kommentare im Datensatz sucht.
- Das Comment Backend startet eine Jena TDB Transaktion.
- Mit Hilfe der Jena API erstellt das Comment Backend eine semantische Repräsentation der Kommentare.
- Das Comment Backend übergibt die RDFS Kommentare dem Comment Converter.
- Der Comment Converter konvertiert diese in eine Liste von Java Objekten.
- Das Comment Backend gibt diese Liste zurück an den Request Handler.
- Dieser ruft noch einmal den Converter auf, um eine JSON Repräsentation der Kommentare zu bekommen.
- Der Converter konvertiert Java Objekte in einen JSON String.
- Der Request Handler sendet den JSON String an das Frontend.

Es spricht natürlich nichts dagegen für solche Fälle ein Kommentar direkt von RDF in JSON zu konvertieren, ohne den Umweg über Java. Im Prototyp wurde darauf aber verzichtet, da es nur zusätzliche Schreibarbeit dargestellt und keinen merkbaren Nutzen gebracht hätte.

## REST API

Um Kommentare dem Frontend zur Verfügung zu stellen, bot sich eine REST API an, da kein Zustand gespeichert werden muss. Eine SOAP Schnittstelle kam nicht in Frage, da das DaRe bereits ein REST Schnittstelle anbietet und einfach ergänzt werden konnte. Es wurden vier Methoden implementiert:

- GET `dare/comment` gibt alle Kommentare aus. Sind `property` (für visualisierte propertys) oder `component` (für Komponenten) Query Parameter vorhanden, wird nach diesen gefiltert und Kommentare gesucht, die mindestens ein Kriterium erfüllen.
- GET `dare/comment/{id}` gibt Kommentar mit ID `id` aus.
- POST `dare/comment` nimmt einen Kommentar in `application/json` entgegen, speichert es ab und antwortet im Erfolgsfall mit Statuscode 201 `Created`.
- PUT `dare/comment/vote/{id}` nimmt ein JSON Objekt entgegen und setzt die Anzahl an angegebenen Stimmen. Ist kein `Authorization` Header vorhanden, antwortet die Methode mit 401 `Unauthorized`. Hat der Benutzer schon einmal diese Anzahl an Stimmen abgegeben, ist die Antwort 403 `Forbidden` und ansonsten 200 `OK`. Auf diese Weise ist PUT wie von RFC 2616 [W3C] gefordert idempotent.
- DELETE `dare/comment/{id}` löscht den Kommentar mit ID `id` aus dem Datensatz. Der `Authorization` Header muss den Benutzer enthalten, der den Kommentar erstellt hat.

Die beiden GET Methoden nehmen optional einen HTTP `Authorization` Header entgegen, der die Benutzer ID (ohne Passwort) angibt. Dann enthalten zurückgelieferte Kommentare Informationen, ob dieser Benutzer schon gevotet hat und wenn ja, wie.

Bei der Methode PUT `dare/comment/vote` überprüft das DaRe nur die Existenz eines `Authorization` Headers und erstellt einen neuen Benutzer im Kommentar Backend, wenn dieser unbekannt ist. Die Methode POST kann ohne Credentials aufgerufen werden. In der Praxis müsste der Header Benutzer ID und Passwort enthalten, sodass das DaRe diese Daten überprüfen kann, indem es eine entsprechende Anfrage an die REST API des User Services stellt. Auf diese Weise würde verhindert, dass beliebige Personen Kommentare erzeugen oder bewerten können. Da die Benutzerinformationen im `Authorization` Header mit Base64 [Jos06] kodiert und nicht verschlüsselt o. ä. werden, sollte die Kommunikation zwischen DaRe und Frontend außerdem mit Transport Layer Security gesichert sein.

### 4.2.2 Bedienung und Kommunikation

Das Backend zur Generierung und Auslieferung der Bilder für die User Assistance wurde im CoRe integriert, da dort Komponenten registriert und gespeichert werden. Die folgenden Abschnitte behandeln seine Architektur, die Erweiterungen der SMCDL, den Generierungsalgorithmus und die zur Verfügung gestellte API.

## Architektur

Abbildung 4.16 zeigt, wie die Bilder für das Hilfesystem im Backend generiert werden. Sobald der Entwickler eine neue Komponente im CoRe registriert, validiert das CoRe diese und transformiert sie in ein semantisches Modell (MCDO). Danach startet das CoRe den Generierungsservice (Assistance Generator). Dieser erstellt eine JSON Datei mit den notwendigen Informationen aus der SMCDL, beispielsweise welche Aktionen (Capabilities) wie ausgeführt werden, und eine HTML Datei, die alle notwendigen Ressourcen, wie etwa Stylesheets und Skripte, enthält. Die JSON und HTML Datei wird an PhantomJS übergeben, welches die Informationen einliest und Screenshots der Komponente erstellt. Testdaten werden von einer in der SMCDL angegebenen URL geladen (siehe Abschnitt 3.3.1). Die bereits bestehende SOAP API des CoRe wurde so erweitert, dass das Hilfesystem auf die URLs der Bilder zugreifen kann. Dafür ist der Assistance Service zuständig.

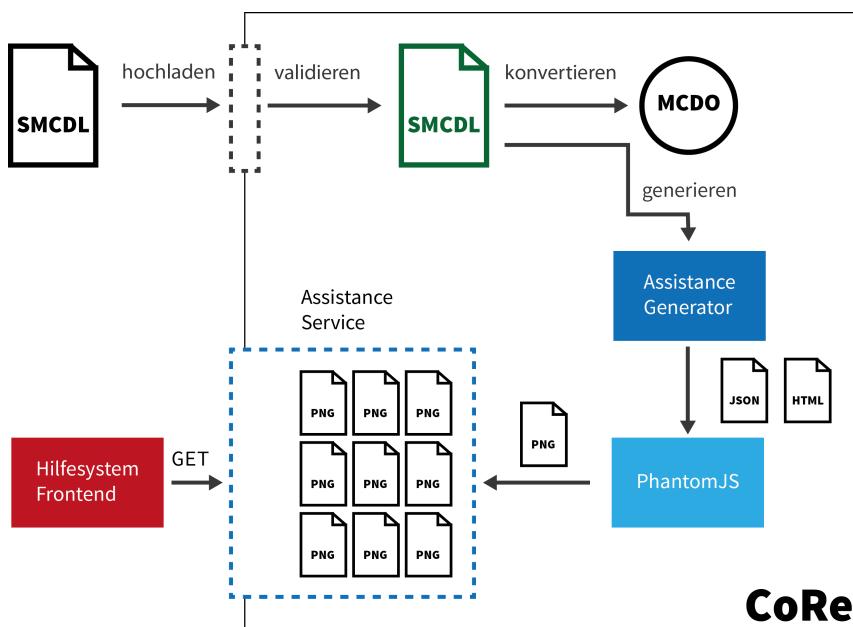


Abbildung 4.16: Generierung der Bilder für das Hilfesystem

Auf die einzelnen Bestandteile des Vorgangs wird im Folgenden näher eingegangen.

## SMCDL

Die SMCDL wurde erweitert, sodass Entwickler die wichtigen Interaktionen in ihrer Komponente annotieren können. Dazu wurden Elemente für atomare, sequentielle und parallele Operationen eingefügt. Mit Hilfe des Attributs `generateAssistance` signalisiert der Entwickler, dass für die entsprechende Capability Hilfe generiert werden soll. Das optionale Attribut `wait` teilt dem Hilfeservice mit, wie lange nach dem Ausführen der VISO Operationen gewartet werden soll, bevor er einen Screenshot erstellt. VISO Operationen entsprechen Nutzeraktionen wie klicken, draggen oder scrollen und sind nicht mit CRUISe Operationen einer Komponente zu verwechseln.

Eine atomare Operation besteht aus einem Element mit folgenden Attributen (Listing 4.2):

- **id** ist zur Unterscheidung von anderen Operationen nötig.
- **element** gibt einen CSS Selektor an, mit dem die UI Elemente gefunden werden können, welche die Aktion auslösen.
- **type** gibt die VISO Operation an. Zur Zeit werden die Mausoperationen Klick, Doppelklick, Move, Drag und Scroll unterstützt.
- **modifier** konkretisiert die Operation, indem es z. B. angibt, welche Maustaste verwendet werden soll.

Um zusätzlich angeben zu können, wie weit der Hilfeservice scrollen soll, kann der atomaren Operation ein JSON Objekt mit den Attributen **dx** und **dy** zur Verfügung gestellt werden (Listing 4.3). Dafür wurde das Element **examplerdata** so angepasst, dass es in einer VISO Operation enthalten sein kann. Im Kontext von Klickoperationen geben diese Informationen den Abstand vom Zentrum des Elements an.

```

<capability
    id="capPan"
    activity="actions:Pan"
    entity="travel:Event"
    wait="2s"
    generateAssistance="true">

    <atomicoperation
        id="panByDblClick"
        element="div.background"
        type="op:DoubleClick"
        modifier="op:LeftButton">
        <examplerdata>{ 'dx': -500, 'dy': 0 }</examplerdata>
    </atomicoperation>
</capability>
```

Listing 4.2: SMCDL für UI Capability

Falls der Entwickler keine Hilfe wünscht, müssen keine Elemente außer **capability** und dessen Attribute **id**, **activity** und **entity** angegeben werden. Soll für diese Capability Hilfe generiert werden, müssen **generateAssistance** und die **atomicoperation** ohne **examplerdata** unbedingt angegeben werden.

Falls die Capability nicht vom Benutzer (UI Capability), sondern von der Komponente ausgeführt wird (System Capability), enthält **capability** Beispieldaten, die an die entsprechende Operation der Komponente weitergegeben werden.

```

<operation
    name="highlightPoint">

    <capability
        id="capHighlight"
        activity="action:brushAction"
        entity="travel:Event"
        provider="system"
        generateAssistance="true">
        <examplerdata>{ 'uri': 'http://mmt.inf.tu-dresden/vizBoard/
SimileTimelineTestData#event1' }</examplerdata>
</capability>
```

```

</capability>

15 <parameter
      name="uri"
      type="xs:string" />

</operation>

```

Listing 4.3: SMCDL für System Capability

Damit die Komponente während der Hilfegenerierung auch tatsächlich Daten anzeigt, muss der Komponentenentwickler in der SMCDL eine URL zu Testdaten angeben (Listing 4.4). Dafür wurde ein weiteres Element `testdata` hinzugefügt. Im Gegensatz zu `exampledata` sind sie nicht Beispieldaten einer Eventnachricht, sondern ein ganzer Datensatz, der in die Komponente geladen wird. Es kann entweder eine `url` zu Testdaten enthalten, welche vor der Hilfegenerierung abgerufen werden, oder aus einer Datensatz ID des DaRe (`darefileid`) und einer SPARQL Query (`mapping`) bestehen, welche die zu ladenden Daten definieren.

```

<testdata>
  <url>http://entwicklerseite.com/testdata/Testdata.json</url>
</testdata>

```

Listing 4.4: SMCDL für Testdaten

Zur Laufzeit und während der Hilfegenerierung muss das Hilfesystem Komponenten in zuvor gespeicherte Zustände zurücksetzen können. Dazu wird das Memento verwendet, welches eine Menge von Propertys ist, deren Werte gespeichert werden müssen. Um anzugeben, welche Propertys einer Komponente Teil ihres Mementos sind, kann der Entwickler diese mit einem Attribut `isMemento` versehen (Listing 4.5). Sie unterscheiden sich von `required` Propertys, die zur Initialisierung der Komponente unbedingt gesetzt sein müssen, weil bei Mementos eben das nicht der Fall ist: Eine Property `sorted`, welche angibt wonach eine Tabelle aktuell sortiert ist, muss bei der Initialisierung nicht bekannt, aber dennoch Teil des Mementos sein, sodass Annotationen die korrekten Bereiche referenzieren. Andersherum müssen `required` Propertys nicht Teil des Mementos sein. Die Propertys `width` und `height` sollten beispielsweise nicht ins Memento aufgenommen werden, da die Komponente später ansonsten plötzlich ihre Größe ändern kann. Ortsbezogene Bereichskommentare funktionieren trotzdem, sofern der Inhalt der Komponente proportional skaliert dargestellt wird.

```

<property name="highlighted" required="false" isMemento="true" type="xs:string" />

```

Listing 4.5: SMCDL für Memento

Im Konzept wurde festgelegt, dass Komponenten keine externen Daten laden oder selbst Datenverarbeitung durchführen dürfen, falls sie Kommentare unterstützen wollen. Für die History ist es notwendig zu wissen, ob das CSS einer Komponente auf relativen Einheiten basiert. Diese Informationen werden am Wurzelement in der SMCDL als Attribute angegeben (Listing 4.6).

```

<component
    id=" http://cruise/ui/SimileTimeline "
    name=" SimileTimeline "
    version=" 1.0 "
    5      isUI=" true "
    doesDataProcessing=" false "
    loadsExternalData=" false "
    hasRelativeStyle=" false ">

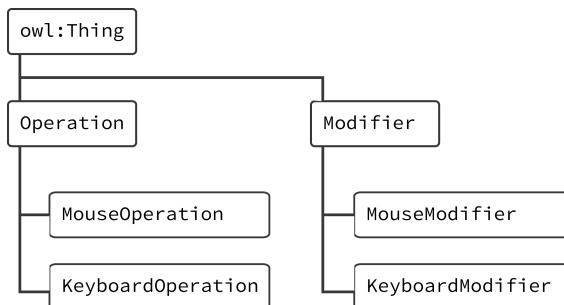
```

Listing 4.6: SMCDL für Zusatzinformationen

Die vorgestellten zusätzlichen Elemente und Attribute der SMCDL wurden zusätzlich auch als semantische Konzepte in der MCDO modelliert. Im nächsten Abschnitt wird gesondert auf die VISO Operationen eingegangen, da diese die größte Neuerung darstellen.

## VISO Operationen

Um Nutzeraktionen semantisch darstellen zu können, wurde eine kleine Ontologie dafür erstellt (Abbildung 4.17), welche hauptsächlich Desktopinteraktionen nachempfunden ist. Sie besteht aus den zwei Klassen **Operation** und **Modifier**, wobei eine **Operation** eine Aktion beschreibt (z.B. klicken) und der optionale **Modifier** die Details liefert (z.B. welche Maustaste). Jede der Klassen hat Subklassen für jedes Eingabegerät, momentan Maus und Tastatur. Soll die Ontologie in Zukunft auch Touchinteraktionen beschreiben, müssen zwei Klassen **TouchOperation** und **TouchModifier** eingefügt werden. **TouchOperation** würde die Geste beschreiben, also etwa Touch, Pinch oder Swipe. Ein **TouchModifier** könnte etwa die Anzahl der verwendeten Finger festlegen.

Abbildung 4.17: Die `activity-operations` Ontologie

Folgende Instanzen sind aktuell in der Ontologie vorhanden:

- **MouseOperation:** Click, DoubleClick, Move, Scroll, Drag.
- **MouseModifier:** LeftButton, MouseWheel, RightButton.
- **KeyboardOperation:** TypeOperation
- **KeyboardModifier:** AltKey, BackspaceKey, ReturnKey, CommandKey, ControlKey, ShiftKey, SpaceKey.

## Assistance Generator & PhantomJS

Bevor der Assistance Generator die PhantomJS Binary startet, erzeugt er eine Datei `mcdl.json`. Dort werden alle zur Generierung wichtigen Informationen gespeichert, zum Beispiel die Capabilities, Testdaten und das Memento der Komponente. Außerdem wird eine Datei `skeleton.html` erstellt, welche alle Abhängigkeiten der Komponente enthält. Wenn keine Abhängigkeit von jQuery angegeben wurde, wird es ebenfalls geladen. Diese Dateien werden zusammen mit anderen Informationen (z. B. dem Pfad zu den Dateien) an PhantomJS weitergegeben, welches das Generierungsskript `generate.js` startet.

Das Skript liest die Informationen der `mcdl.json` ein und führt den in `generate.js` enthaltenen Algorithmus aus:

1. Zunächst wird der Konstruktor der Komponente sowie die Operationen `enable` und `show` aufgerufen.
2. Danach wird ein Memento vom initialen Zustand der Komponente abgerufen, um diesen später wiederherstellen zu können.
3. Die Testdaten der Komponente werden geladen, indem selbige aus der Datei `mcdl.json` über die Methode `loadTestdata` an die Komponente gegeben werden.
4. Es werden sequentiell für alle Capabilities Screenshots generiert. Die Größe der Bilder entspricht dabei den in der SMCDL angegebenen `default` Werten für Breite und Höhe einer Komponente. Es wird nacheinander jeweils die erste atomare Operation jeder (UI) Capability behandelt. Für den Prototypen wurde hier auf sequentielle und parallele Operationen verzichtet, sie stellen aber nur eine Erweiterung des umgesetzten Prinzips dar und können bei Bedarf einfach implementiert werden.

Ist die aktuell behandelte eine UI Capability, werden die dazugehörigen UI Elemente mit dem CSS Selektor aus der SMCDL gefunden, hervorgehoben und die Bounding Box des ersten Elements abgefragt. Danach wird ein Screenshot der Komponente erstellt und als PNG Datei abgespeichert. Der Dateiname setzt sich aus der ID der Operation und der Bounding Box im Format `XxYxWxH` zusammen, also beispielsweise `clickButton-12.5x12.5x50x50.png`. Das Bild wird in einem Ordner, der nach der ID der Capability benannt ist, abgelegt (z. B. `capSelect` in Abbildung 4.18). Danach wird das CSS der hervorgehobenen DOM Elemente zurückgesetzt und die VISO Operation (also z. B. ein Klick) auf dem ersten der Elemente ausgeführt. Das Hilfesystem wartet die angegebene Anzahl von Sekunden, bevor es ein letztes Bild `result.png` erstellt und im Ordner der Komponente ablegt.

Handelt es sich um eine System Capability, sind keine UI Elemente beteiligt. In diesem Fall wird nur die CRUISe Operation mit den angegebenen Beispieldaten ausgeführt und danach ein Screenshot `result.png` erstellt.

Nachdem die Generierung der Screenshots abgeschlossen ist, wird der Assistance Service darüber informiert, wo der Ordner mit den Bildern der behandelten Komponente liegt. Er kopiert die Bilder ins öffentliche Verzeichnis (siehe nächster Abschnitt), von wo sie für das Frontend erreichbar sind.

```

/UI_LineChart-75017412/
    mcdl.json
    skeleton.html
/capSelect/
    clickButton-40x50x2x2.png
    result.png
/capZoom/
    scrollVis-0x0x100x100.png
    result.png

```

Abbildung 4.18: Beispielhafte Ordnerstruktur einer Komponente im Assistance Generator

## API

Die Methode `getImagesByComponentAndCapability` nimmt eine `componentId` und eine `capabilityId` entgegen und antwortet mit den Pfaden zu den gewünschten Bildern im JSON Format (Listing 4.7), damit das Frontend des Hilfesystems auf die generierten Bilder zugreifen kann. Diese sind relativ zum öffentlich zugänglichen CoRe Ordner des Servers. Das Attribut `bbox` gibt dabei die Bounding Box der relevanten Elemente relativ zur Größe des Bildes in der Form  $[x, y, w, h]$  an. Sie wird aus dem Dateinamen des Bildes extrahiert und ist notwendig, um im Frontend auf den gewünschten Teilbereich zoomen zu können.

```

{
  "result": "assistanceImg/<id>/capPan/result.png",
  "operations": {
    "dragBand": {
      "type": "atomic",
      "id": "dragBand",
      "label": "Drag",
      "bbox": [ 0, 0, 100, 69.167 ],
      "url": "assistanceImg/<id>/capPan/dragBand-0x0x100x69
              .167.png"
    }
  }
}

```

Listing 4.7: Antwort des SOAP Interfaces

## 4.3 Zusammenfassung

In diesem Kapitel wurden die prototypische Implementierung der Konzepte Bedienungshilfe (Abschnitt 3.3 im Konzept), Kommunikationshilfe (Abschnitt 3.5) und die Kommentarfunktion (Abschnitt 3.7) beschrieben. Diese Konzepte wurden ausgewählt, weil sie repräsentativ für die Ziele des Hilfesystems (Bedienungs-, Kommunikationshilfe erleichtern Nutzung, Kommentare fördern Erkenntnisgewinn), aber im Vergleich zu anderen nicht trivial zu implementieren sind. Das gesamte Assistance

Modul wurde in einer Backbone.js Applikation gekapselt und wird nur bei Bedarf gestartet.

Für das Backend der Bedienungshilfe (Abschnitt 4.2.2) wurden die SMCDL und MCDO erweitert, sodass der Entwickler mögliche Interaktionen für eine Capability angeben kann. Dazu greift er auf atomare Operationen wie z. B. klicken zurück, welche auf sequentielle und parallele Weise kombiniert werden können. Falls die Capability nicht vom Nutzer, sondern von der Komponente initiiert wird, muss der Entwickler Beispieldaten für die entsprechende CRUISe Operation angeben. Nachdem eine Komponente registriert wurde, startet der Assistance Generator den Headless Browser PhantomJS, welcher die notwendigen Bilder generiert. Im Prototypen kann PhantomJS nur atomare Mausoperationen (Klick, Doppelklick, Drag, Scroll, Hover) verarbeiten, die Erweiterung auf sequentielle Operationen und die Unterstützung von Tastaturoperationen sollten aber kein Problem darstellen, da die wichtigsten Funktionen bereits implementiert wurden. Am Ende wird der Assistance Service informiert, welcher die Bilder in den öffentlich zugänglichen Ordner des CoRe kopiert, sodass das Frontend des Hilfesystems darauf zugreifen kann.

Die Mausoperationen werden in PhantomJS teilweise durch eine Kombination von synthetischen Mausevents des Browsers simuliert. Das funktioniert bis auf die Operation Drag auch sehr gut. Dort jedoch besteht das Problem, dass der Effekt von der Implementierung abhängt. Manche Komponenten wollen am Ende des Draggens ein anderes Event Target als beim Start (z. B. wenn eine Spalte einer Tabelle verschoben werden soll), während es bei anderen gleich bleiben soll (z. B. panning einer Karte). Um dem beizukommen sollte in Zukunft das gewünschte Target bei DragEnd in den Zusatzinformationen zur Operation angegeben werden können.

Das Frontend der Bedienungshilfe (Abschnitt 4.1.4) liest die benötigten Informationen ebenfalls aus der SMCDL einer Komponente, da diese über die TSR zugänglich ist. Um herauszufinden, wie Propertys in der Komponente kodiert werden, stellt das Hilfesystem Anfragen an die Schnittstelle `getVisualAttributes` des CoRe. Da die CSS Selektoren für aktionsrelevante UI Elemente und die unterstützten Capabilities in der SMCDL vorhanden sind, kann das Hilfesystem diese in der Komponente hervorheben. Wenn Bilder für die Comicpanels benötigt werden, stellt es eine Anfrage an die Schnittstelle `getImagesByComponentAndCapability` des CoRe, welche mit den Pfaden zu den Bildern, den Bezeichnungen der Operationen und der Bounding Box der UI Elemente antwortet. Die Bilder werden dann mit Hilfe von CSS3 in animierter Form präsentiert.

Jedoch ist die Art und Weise, wie im Prototypen aktionsrelevante UI Elemente hervorgehoben werden, nicht performant genug. Die DOM Elemente werden aus ihrem vorgesehenen Kontext genommen und es ist viel Aufwand nötig, um ihr Aussehen beibehalten zu können. Im Zuge dieser Berechnungen kommt es zu häufigem Layout Thrashing und Garbage Collections, welche den Tab des Browsers für mehrere Sekunden einfrieren lassen. Da die umgesetzte Form der Darstellung konzeptionell nicht wichtig genug ist, um den Aufwand zu rechtfertigen, sollten in Zukunft UI Elemente in ihrem ursprünglichen Kontext hervorgehoben werden.

Außerdem ist die Darstellung des Comics problematisch, wenn die in den Panels enthaltenen Bilder nicht annähernd quadratisch sind. Sie werden verkleinert gezeigt, sodass unter Umständen der Inhalt nicht mehr erkennbar ist. Mögliche Lösungen dafür sind ein intelligenter, dynamischer Layoutalgorithmus oder der Verzicht auf

Animationen und die Generierung vollständiger Comics im Backend.

Für die Umsetzung der Kommunikationshilfe (Abschnitt 4.1.5) wurde die statische Variante davon implementiert und in den CapView von EDYRA integriert, da dieser schon Verbindungen zwischen Komponenten zeichnet. Im Backend ist dafür nichts weiter zu tun, da die Bilder bereits von der Bedienungshilfe generiert werden. Für die Beschriftung der Captions wird der **LabelGenerator** von EDYRA herangezogen.

Das Backend der Kommentare (Abschnitt 4.2.1) wurde ins Data Repository integriert, da dort bereits eine Infrastruktur zur Datenspeicherung vorhanden ist. Die Kommentare werden als RDF in einem Jena TDB Triplestore abgelegt und ausgelesen. Damit das Frontend diese anzeigen kann, wurde die REST API des DaRe entsprechend erweitert. Es existieren Methoden, um Kommentare hinzuzufügen, abzurufen, zu bewerten und zu löschen.

Das Frontend der Kommentare (Abschnitt 4.1.3) zeigt Annotationen mit Hilfe von SVG Elementen an. Deren korrekte Position und Größe wird durch die von der Komponente implementierten Methode **dataToCoordinates** bestimmt, welche Daten in Koordinaten konvertiert. Nachdem ein Kommentar inklusive Annotations verfasst wurde, werden die betroffenen Datenbereiche mit **coordinatesToData** ausgelesen und das Kommentar ans Backend geschickt. Danach kann es in einer anderen Komponente wieder geladen und dargestellt werden.

Einschränkungen bei der Präsentation der Kommentare existieren einerseits bei der Nutzung von Bounding Boxes, um **mouseover** über Datenpunkten zu registrieren. Bounding Boxes sind immer rechteckig, was bei transformierten oder nicht-eckigen Repräsentationen von Datenpunkten zu False Positives führen kann. Zur Lösung sollte in Zukunft eine optionale Methode in die API der Komponente hinzugefügt werden, welche angibt, ob sich ein Datenpunkt an einem bestimmten Punkt befindet. Andererseits werden ortsbezogene Bereichskommentare nicht korrekt dargestellt, wenn sich die Größe der Komponente ändert, da die Positions- und Größenangaben relativ zur Größe der Visualisierung erfolgen. Ohne Einschränkung der Usability (wenn Komponenten auf für den Anwender unvorhersehbare Weise ihre Größe ändern) kann das Problem auf Grund des »Black Box« Prinzips der Komponenten nicht gelöst werden. Am Besten wäre es, wenn Komponenten ihren Inhalt proportional skalieren würden.

# 5 Evaluation

Nachdem basierend auf dem Szenario (Abschnitt 2.1), den Grundlagen (Abschnitt 2.3) und dem Konzept (Kapitel 3) ein Prototyp entwickelt wurde (Kapitel 4), wird in diesem Abschnitt dessen Tauglichkeit überprüft. Das Hilfesystem soll drei Aufgaben erfüllen:

1. Es soll dem Benutzer erleichtern, Komponenten verschiedener Entwickler zu bedienen (Abschnitte 3.2, 3.3, 3.4, 3.5, 3.8, 3.9).
2. Dem Komponentenentwickler soll die Möglichkeit gegeben werden, seine Komponenten schnell und einfach in das Hilfesystem zu integrieren.
3. Es soll dem Benutzer helfen, die in den Komponenten dargestellten Informationen zu verstehen und Erkenntnisse zu gewinnen (Abschnitte 3.6, 3.7).

Im nächsten Abschnitt wird untersucht, ob Aufgabe 1 erfüllt wird. Abschnitt 5.2 behandelt Aufgabe 3. Da Aufgabe 2 im Prototypen durch Kommentare umgesetzt wird, kann für sie keine qualitative Evaluation durchgeführt werden. Maximal würde das Leseverständnis der Testpersonen überprüft. Durch die Implementierung im Prototypen wurde aber die Umsetzbarkeit des Konzepts nachgewiesen, womit Kommentare über Komponenten hinweg wiederverwendet werden und so den Erkenntnisgewinn bei den Anwendern fördern können.

## 5.1 Nutzerstudie

Das Hilfesystem soll die Usability eines Mashups, bestehend aus Effektivität, Effizienz und Zufriedenheit der Nutzung, erhöhen [ISO98]. Ziel ist zu überprüfen, wie das Hilfesystem angenommen wird und ob es verständlich genug ist. Zehn Teilnehmer verwendeten dieselbe Mashup Komposition und mussten fünf Fragen beantworten, deren Lösung mehr oder weniger offensichtlich ist. Währenddessen wurde falls notwendig nachgefragt, ob die Erklärungen verständlich sind, ob etwas anderes erwartet wurde und ob eine andere Darstellung der Hilfe mehr geholfen hätte. Es wurden Effizienz- und Effektivitätsmetriken in Form der benötigten Zeit zur Lösung und der Korrektheit der Lösung für jede Frage aufgenommen. Um die Zufriedenheit zu überprüfen, wurde am Ende ein Usability Fragebogen ausgefüllt.

### 5.1.1 Mashupkomposition

Die Teilnehmer benutzten ein Mashup aus drei Komponenten (Abbildung 5.1): Eine SIMILE Timeline, ein Liniendiagramm und eine Tabelle.

Als Datensatz wurden die ersten 60 Filme (alphabetisch sortiert) aus der DBpedia verwendet. Jeder Film hatte folgende Attribute: Titel, Beschreibung, Wikipedia

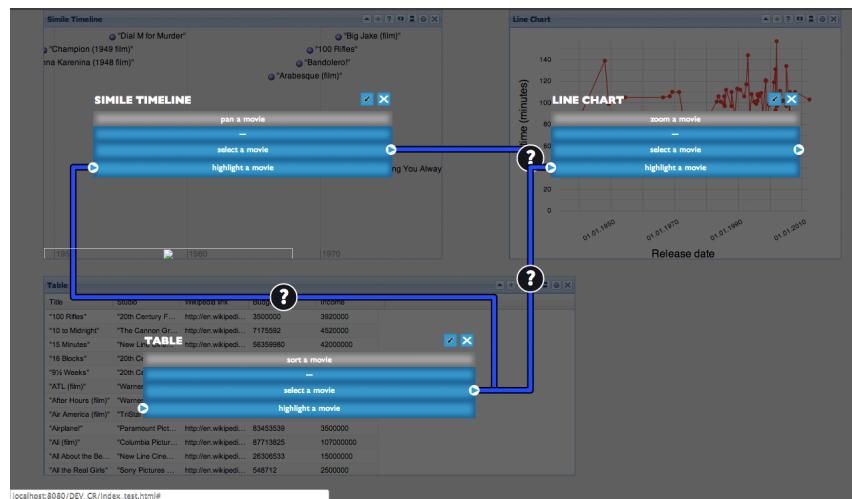


Abbildung 5.1: Objekt der Nutzerstudie

Link, Budget, Einnahmen, Datum der Veröffentlichung, Studio. Eine Selektion in der Tabelle hatte Hervorhebungen in den anderen zwei Komponenten zur Folge, eine Selektion in der Timeline passte das Liniendiagramm an.

### 5.1.2 Aufgaben

Die Teilnehmer sollten folgende Aufgaben lösen:

1. »Wann erschien der Film Ever After (Jahr, Monat, Tag)?« Zur Lösung dieser Aufgabe mussten sie den Film zuerst in der Tabelle und danach in der Timeline auswählen oder direkt in der Timeline danach suchen und auswählen.
2. »War Chaplin im Vergleich zu anderen Filmen in diesem Zeitraum eher länger oder eher kürzer?« Die Teilnehmer mussten den Film Chaplin auswählen und im Liniendiagramm seine vertikale Position ablesen.
3. »Welcher Film wurde am drittbilligsten produziert?« Um diese Frage zu lösen, mussten die Teilnehmer die Tabelle nach Budget sortieren. Für manche ist das intuitiv, andere sollten dazu die Hilfe benötigen.
4. »Wer führte bei Big Night Regie?« Hier musste der Film ausgewählt und seine Beschreibung gelesen werden.
5. »Welcher der beiden Filme ist länger: Cassandras Dream oder Gaby: A true story?« Diese Frage ist die schwierigste von allen. Die gedachte Lösungsstrategie ist wie folgt: Zuerst müssen die beiden Filme ausgewählt werden, um sie im Liniendiagramm zu finden. Selbiges musste dann ungefähr auf den Teilbereich der beiden Filme gezoomt werden, sodass die vertikale Position genauer abgelesen und die Filme verglichen werden können.

### 5.1.3 Teilnehmer

Für die Nutzerstudie wurden 10 Teilnehmer im Starbucks in der Prager Straße in Dresden (Abbildung 5.2) zufällig ausgewählt, da laut Virzzi [Vir92] mit sieben schon

80 % der dringendsten Usabilityprobleme gefunden werden können. Die Teilnehmer waren zwischen 18 und 53 Jahren alt (Durchschnitt 30.3 Jahre, Median 27.5 Jahre, Standardabweichung 10.84 Jahre) und zu 60 % weiblich (siehe Tabelle 5.1). Es gab keine Kriterien, welche die Teilnahme an der Studie verhindert hätten.

Teilnehmer	Geschlecht	Alter	Domäne/Beruf	Englisch
Teilnehmer 1	M	30	Fachberater	B2
Teilnehmer 2	W	28	Krankenpflegerin	B1
Teilnehmer 3	W	18	Studentin	B1
Teilnehmer 4	M	20	Student	B2
Teilnehmer 5	M	27	Softwareentwickler	C1
Teilnehmer 6	M	26	IT Consultant	A1
Teilnehmer 7	W	22	Studentin (Psychologie)	C1
Teilnehmer 8	W	42	Bürokauffrau	A1
Teilnehmer 9	W	53	Grundschullehrerin	A1
Teilnehmer 10	W	37	Betriebswirtin	B1

Tabelle 5.1: Teilnehmer der Nutzerstudie



Abbildung 5.2: Ort der Nutzerstudie

### 5.1.4 Durchführung

Zunächst wurde den Teilnehmern der Studie grob erklärt, was ein Mashup ausmacht: Es gäbe verschiedene Fenster, welche dieselben Daten auf unterschiedliche Weise darstellen. Außerdem bestünden Zusammenhänge zwischen diesen Fenstern in dem Sinne, dass sie auf Änderungen in anderen Fenstern reagierten und sich anpassten. Jedoch gelte das nicht für alle Fenster und nicht in jede Richtung. Danach wurden sie über die zu lösenden Aufgaben aufgeklärt. Die Teilnehmer wurden gebeten, dass sie, falls sie nicht weiter wissen, die Hilfefunktionen nutzen sollten anstatt lange zu

probieren. Sie wurden auf die Bedienungs- und Kommunikationshilfe aufmerksam gemacht und erhielten außerdem den Hinweis, dass sie währenddessen nachfragen dürften, um eine der Hilfefunktionen zu finden.

Danach wurden sie gebeten, die fünf Fragen zu beantworten. Dazu verwendeten sie das vorgestellte Mashup auf einem Macbook Air (13 Zoll, Mid 2012). Es wurde keine Maus zur Verfügung gestellt, die Teilnehmer benutzten das Touchpad des Notebooks. Nach dem ersten Teilnehmer wurde die »Natural Scroll« Funktion<sup>1</sup> deaktiviert. Für jede Aufgabe wurde die Zeit gestoppt und festgehalten, ob sie eine Hilfefunktion verwendeten. Danach füllten die Teilnehmer einen Fragebogen mit allgemeinen Informationen (Alter, Beruf, Geschlecht, Sprachniveau Englisch) und einen SUS Fragebogen [Bro96; Rum13] aus. Das »System«, welches in den SUS Fragen angesprochen wird, referenzierte dabei VizBoard als Ganzes, inklusive Hilfesystem. Den Teilnehmern sollte nicht zugemutet werden, mental zwischen Hilfesystem und VizBoard zu trennen.

### 5.1.5 Ergebnisse

Zunächst muss festgestellt werden, dass die Teilnehmer nur in einem Drittel der möglichen Fälle (17 von 50 Fragen) die Bedienungshilfe verwendeten (Abbildung 5.3). Das legt den Schluss nahe, dass die Bedienungshilfe in der Praxis noch weniger aufgerufen würde, da die Teilnehmer vor der Studie dazu ermuntert wurden. Ausprobieren war bei 90 % der Teilnehmer die bevorzugte Strategie, dabei hätte ihnen die dynamische Meta-Hilfe (Abschnitt 3.9.2 im Konzept) vermutlich mehr geholfen als die statische Bedienungshilfe.

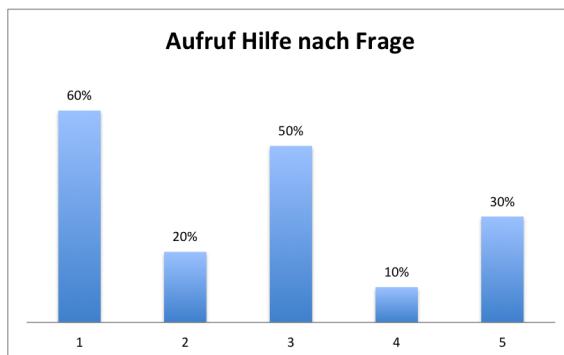


Abbildung 5.3: Wie viele Teilnehmer die Hilfe aufriefen, aufgeschlüsselt nach Aufgabe

Während die Teilnehmer versuchten, die Bedienungshilfe zu verstehen, waren ein paar Missverständisse gehäuft zu beobachten. Zwei Testpersonen versuchten die Comic Panels zu klicken, wobei eine dazu durch die Beschriftung »Click Left Button« in der Mitte verleitet wurde. Dieses Verhalten zeigten auch schon Testpersonen der

<sup>1</sup>Wenn der Inhalt eines Fensters als Dokument unter einer Lupe vergegenwärtigt wird, geht es darum, was beim scrollen bewegt wird. Natural Scroll bewegt das Dokument, es muss also nach oben wandern, um Text weiter unten lesen zu können. In Windows Systemen ist es üblich, dass beim scrollen die Lupe bewegt wird. Sie muss ihre Position nach unten verändern, um Text weiter unten zu lesen.

formativen Studie, woraus geschlossen wurde, dass ein ansprechenderes, mehr nach Comic aussehendes Layout notwendig ist.

Die Formulierung der Operation im mittleren Panel sollte klarer sein, da sie für drei Teilnehmer nicht ganz verständlich war. Für manche war sie zu genau, sie suchten beim Touchpad des Macbooks nach einer linken oder rechten Taste<sup>2</sup>. Diesen Personen hätte »Click« voll ausgereicht. Für eine andere war die Beschreibung wiederum zu ungenau, sie kannte das linke Panel als Aktionselement. Ihr wäre mit der Beschreibung »Click Left Mouse Button« mehr geholfen, weswegen die Formulierungen möglichst genau sein sollten. Außerdem sollte überlegt werden, das Gerät des Nutzers im Context Service zu speichern, sodass die Beschreibungen dahingehend angepasst werden können.

Fast die Hälfte der Teilnehmer sah sich die Bedienungshilfe an und versuchte danach auf den von der Hilfe hervorgehobenen Elementen zu arbeiten. Aus diesem Grund sollte deutlicher gemacht werden, dass diese nur eine Kopie darstellen, oder nach Klick noch einmal abgefragt werden, ob die Hilfe aufgerufen oder geschlossen werden soll.

Nur ein Teilnehmer rief die Kommunikationshilfe auf. Die anderen hatten sie vergessen oder erhofften sich nicht viel davon, obwohl sie ihnen vermutlich geholfen hätte. Um die Benutzer stärker darauf hinzuweisen, sollte die dynamische Kommunikationshilfe umgesetzt und in einer Studie evaluiert werden. Den Testpersonen, welche die Kommunikationshilfe nicht von selber benutzten, wurde sie am Ende gezeigt. Sie war für alle verständlich, was wohl auch daran liegt, dass die Captions der Bilder einen erklärenden Satz bilden (»Select a movie to highlight a movie.«).

Die Ergebnisse des SUS Fragebogens sind durchwachsen, aber insgesamt positiv (Abbildung 5.4). Der Median beträgt 73.75 (Standardabweichung 17.64), was nach [BKM09] einem »Gut« entspricht.

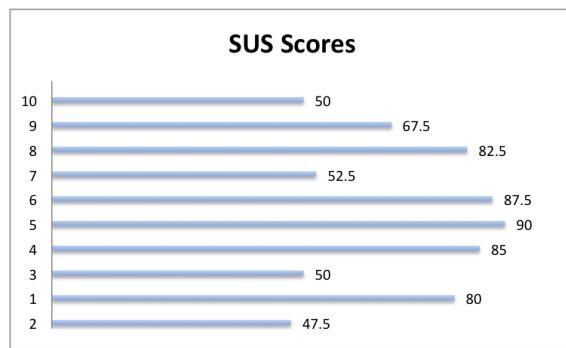


Abbildung 5.4: SUS Scores der Teilnehmer

Auffällig sind die vier Teilnehmer, welche einen SUS Score von ungefähr 50 vergaben. Nummer 2 wirkte während der Studie sehr sicher und merkte auch im Feedback an, dass sie das System VizBoard an sich interessant und nützlich findet. Zeitlich befand sie sich im Mittelfeld und hatte keine offensichtlichen Probleme. Teilnehmerin 4 hatte kein mentales Modell dafür, wie Mashups funktionieren und konnte deswegen auch keine Lösungsstrategie planen, weswegen sie sehr lange für jede Frage benötigte. Ihr hätte wahrscheinlich die dynamische Kommunikationshilfe genutzt

<sup>2</sup>Das Touchpad des Macbooks hat nur eine physische Taste. Ein Rechtsklick wird ausgelöst, nachdem diese mit zwei Fingern gleichzeitig gedrückt wurde.

Teilnehmerin Nummer 7 sah sich erst alle Hilfefunktionen an und löste danach alle Fragen vergleichsweise schnell, bis auf die letzte auch alle korrekt. Sie wirkte dabei kaum angestrengt und sicher. Die letzte Teilnehmerin konnte alle Aufgaben lösen, benötigte aber bei zwei davon deutlich länger als die anderen. Sie hatte Probleme zu verstehen, dass die Hilfe Kopien erstellte, auf denen sie nicht arbeiten kann. Für sie wären die zuvor erarbeiteten Änderungen hilfreich gewesen. Zusammenfassend kann gesagt werden, dass den Beobachtungen nach zu urteilen zumindest die Bewertungen der Teilnehmerinnen 2 und 7 eigentlich besser ausfallen hätten müssen, da sie keine offensichtlichen Probleme hatten. Für die anderen zwei ist eine Lösung entweder im Konzept vorgesehen oder wurde nach der Studie erarbeitet.

Nachdem die Studie mit nur 10 Teilnehmern durchgeführt wurde, nur zwei der fünf Hilfefunktionen zur Bedienung enthielt und das Mashup klein genug war, um die Aufgaben auch ohne Hilfe zu lösen, sollte in Zukunft eine weitere Studie durchgeführt werden. Idealerweise mit mehr Teilnehmern, mehr Komponenten im Mashup mit komplizierterer Kommunikation und weiteren Hilfefunktionen.

## 5.2 Entwicklungsaufwand

Wie einführend erwähnt, ist ein Ziel des vorgestellten Konzepts, Komponenten mit möglichst wenig Aufwand in das Hilfesystem integrieren zu können. Unter Aufwand fällt hier sowohl zeitlicher Aufwand für Planung und Umsetzung als auch Entwicklungsaufwand beispielsweise in Form von Lines of Code (LOC). Je größer die Codebase einer Komponente, desto höher ist der Wartungsaufwand im Laufe der Zeit.

Nun ist es nicht möglich, alle verschiedenen Varianten einer Hilfefunktion zu analysieren und zu vergleichen, weswegen beispielhaft nur ein Konzept umgesetzt wird. Als Testobjekt kommt die SIMILE Timeline<sup>3</sup> (Abbildung 5.5) zum Einsatz, da sie mehr Interaktionen als nur »Auswahl« bietet.

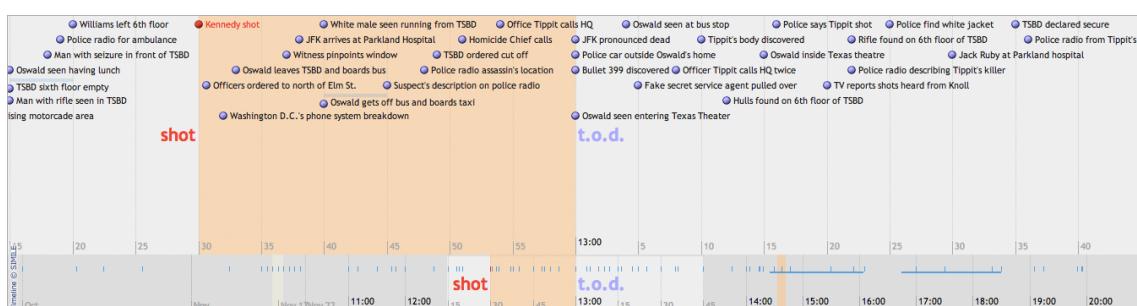


Abbildung 5.5: SIMILE Timeline

### 5.2.1 Individuelle Hilfefunktion

Da der Autor der Arbeit sich in den Grundlagen schon mit der Präsentation und dem Aufbau von Hilfefunktionen und Erklärungen beschäftigt hat, ist er keine geeignete Testperson. Aus diesem Grund wurde ein Medieninformatikstudent gefragt,

<sup>3</sup><http://www.simile-widgets.org/timeline/>

auf welche Weise er die möglichen Interaktionen der SIMILE Timeline einem Benutzer erklären würde. Er entschied sich für ein animiertes GIF, welches am Anfang eingeblendet wird. Die Konzeption dauerte 11 Minuten. Der Autor hat danach die entsprechenden Bilder und die Animation erstellt (Abbildung 5.6). Das dauerte insgesamt 22 Minuten.

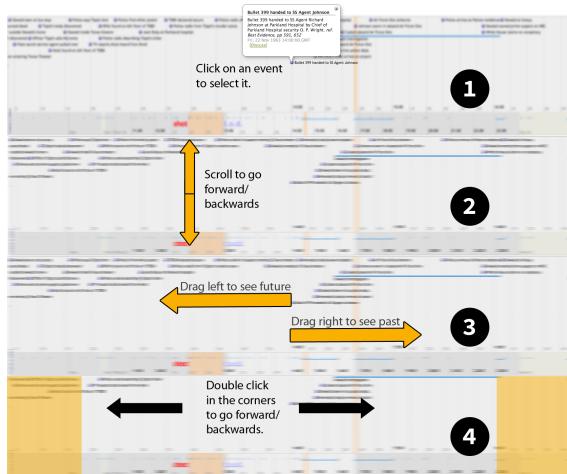


Abbildung 5.6: Frames des animierten GIFs

Die Integration in die Komponente dauerte weitere 10 Minuten und benötigte 8 Zeilen (Listing 5.1). Alles in allem wurden 43 Minuten für diese Hilfefunktion aufgewendet.

```
var help = $( document.createElement( "img" ) );
help.attr( "src", "images/help.gif" )
    .attr( "width", this.width )
    .attr( "height", this.height );
5 help.on( "click", function() {
    $( this ).remove();
});
$( this.body.el.dom ).append( help );
```

Listing 5.1: Integration des GIFs

## 5.2.2 Vorgestelltes Hilfesystem

Für die Integration der Bedienungshilfe in die Komponente müssen die notwendigen Informationen in die SMC DL Datei eingefügt werden. Wie im vorherigen Abschnitt auch wurde ein Kommilitone gebeten, diese Aufgabe auszuführen. Er hat bereits Erfahrung mit CRUISe Komponenten und integrierte z. B. die SIMILE Timeline. Nachdem er ein Tutorial<sup>4</sup> gelesen hatte, sollte er die SMC DL »seiner« SIMILE Timeline entsprechend anpassen. Er benötigte nach eigenen Aussagen 10 Minuten, um das Tutorial zu lesen und 22 Minuten für die tatsächliche Umsetzung, also insgesamt 32 Minuten.

<sup>4</sup><https://trac.mmt.inf.tu-dresden.de/CRUISe/wiki/tutorials/assistance>, ein TRAC Account ist nötig.

### 5.2.3 Auswertung

Für die Integration einer individuellen Hilfefunktion waren 43 Minuten nötig; die SMCDL einer bestehenden Komponente anzupassen erforderte einen Zeitaufwand von 32 Minuten (Abbildung 5.7). Das vorgestellte Hilfekonzept kann also schneller integriert werden als individuelle Hilfefunktionen.

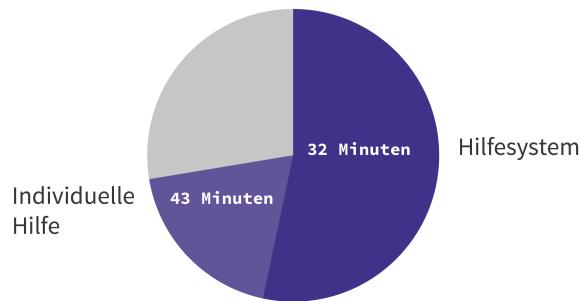


Abbildung 5.7: Benötigte Zeit zur Integration

Der Unterschied beträgt aber nur 11 Minuten und es waren verschiedene Personen beteiligt, weswegen die Zahlen eher als Richtwerte zu sehen sind. Allerdings ist festzuhalten, dass das vorgestellte Hilfesystem einige Vorteile gegenüber einer individuellen Hilfe hat.

Zunächst ist es flexibler: Kommen Funktionen hinzu oder werden welche im Laufe der Entwicklung entfernt, muss im Falle der individuellen Hilfefunktion der ganze Prozess der Hilfeerstellung erneut durchlaufen werden. Das bedeutet Screenshots machen, in ein Grafikverarbeitungsprogramm wie Photoshop laden, dort die Hilfestellungen einfügen, exportieren, alle Bilder in eine andere Software laden und die Sequenz als GIF exportieren. Verwendet der Entwickler aber das Hilfesystem, muss er nur die SMCDL anpassen und updaten.

Ein weiterer Vorteil ist die einfachere Wartbarkeit. Im vorgestellten Beispiel waren nur acht Zeilen Quellcode nötig, allerdings hindert den Komponentenentwickler nichts daran bessere und fortgeschrittenere Hilfefunktionen zu implementieren, die mehr Quellcode nötig machen. Dieser zusätzliche Quellcode, der mit dem vorgestellten Hilfesystem nicht notwendig ist, erhöht die Komplexität der Software und macht sie damit schwieriger wartbar.

Zuletzt gibt es Eigenschaften, die mit einer individuellen Hilfe gar nicht umgesetzt werden können. Dazu gehören einheitliche Formulierungen, Interaktionen und Aussehen. Ein einheitliches Look and Feel ist jedoch wichtig für die Bedienbarkeit von Software (vgl. verwandte Arbeiten, Abschnitt 2.4).

## 5.3 Zusammenfassung

In diesem Kapitel wurde der Prototyp hinsichtlich zweier seiner Aufgaben evaluiert. Um festzustellen, wie die Hilfe angenommen wird und ob sie verständlich ist, wurde eine Nutzerstudie durchgeführt (Abschnitt 5.1). Zur Einschätzung des Entwicklungss-

aufwandes wurde diesbezüglich eine individuelle Hilfefunktion mit dem Hilfesystem verglichen (Abschnitt 5.2).

Die Nutzerstudie wurde mit 10 Teilnehmern zwischen 18 und 53 Jahren (davon 6 Frauen) durchgeführt. Die Studie selbst bestand aus fünf Fragen, die sie mit Hilfe eines Mashups aus drei Komponenten (SIMILE Timeline, Liniendiagramm, Tabelle) lösen sollten. Danach füllten sie einen SUS Usability Fragebogen aus. Dieser fiel insgesamt gut aus, was darauf hindeutet, dass das System mit den weiteren Funktionen aus dem Konzept noch benutzerfreundlicher ist. Es konnten einige Fehlinterpretationen der Hilfe UI identifiziert werden. Die meisten davon hatten entweder damit zu tun, dass die Teilnehmer versuchten, auf den von der Hilfe hervorgehobenen UI Elementen zu arbeiten, oder es gab Missverständnisse bezüglich der Formulierung von Anweisungen. Zur Lösung von ersterem Problem kann eine Abfrage helfen, um zu klären ob Hilfe angezeigt oder die vorgesehene Aktion ausgeführt werden soll. Bezuglich der Formulierungen wird in Zukunft der kleinste gemeinsame Nenner verwendet. Auffällig war außerdem, dass die Hilfe relativ selten aufgerufen wurde, obwohl die Teilnehmer am Anfang der Studie dazu ermuntert wurden. Das ist ein Hinweis darauf, dass die dynamische Meta-Hilfe sehr nützlich gewesen wäre.

Für die Einschätzung des nötigen Entwicklungsaufwandes wurde eine von einem Kommilitonen gewählte individuelle Hilfefunktion für die SIMILE Timeline mit der Integration dieser ins Hilfesystem verglichen. Die individuelle Hilfefunktion bestand aus einem animierten GIF. Dessen Entwicklungsaufwand bestehend aus Konzeption, Umsetzung und Integration betrug 43 Minuten und 8 Zeilen Quellcode. Ein anderer Kommilitone, welche die SIMILE Timeline ursprünglich in CRUISe integrierte, aktualisierte sie für das Hilfesystem. Er benötigte insgesamt 32 Minuten und keinen Quellcode. Somit wurde gezeigt, dass das Hilfesystem schneller als eigene Funktionen integriert werden kann. Allerdings überwiegen vor allem die anderen Vorteile des Hilfesystems: Es ist flexibler, einfacher wartbar und ermöglicht einheitliche Bezeichnungen und Interaktionen.

# 6 Zusammenfassung und Ausblick

Diese Arbeit setzte sich damit auseinander, welche Hilfefunktionen eine Mashup Plattform seinen Benutzern bieten kann, wie diese umgesetzt werden können und wie dabei der Aufwand für Entwickler, die diese Hilfefunktionen für ihre Komponenten einsetzen wollen, möglichst gering gehalten wird. Es wurde ein Konzept bestehend aus acht Lösungsansätzen vorgestellt, von denen drei prototypisch implementiert wurden.

## 6.1 Zusammenfassung

Um die Basis für das Konzept zu schaffen, wurden zuerst die Grundlagen von Informationsvisualisierungen und User Assistance in Mashups sowie im Allgemeinen erarbeitet. Dabei wurden Beispiele für mögliche Informationsvisualisierungen nach dargestellten Daten, Art der Visualisierung und Art der Interaktion kategorisiert vorgestellt. Es stellte sich heraus, dass besonders die Darstellung viele verschiedene, auch weniger gängige Formen annehmen kann (z. B. verschachtelt, Dense Pixel). Trotzdem muss das Hilfesystem generisch genug arbeiten, um mit allen davon umgehen zu können. Um mehr darüber zu lernen, wie Hilfefunktionen aussehen sollen und was dabei beachtet werden muss, wurde einerseits ein Einblick in die kognitive Multimedia-Lerntheorie gegeben. Dabei stellte sich heraus, dass kein Medium inhärent besser zum Lernen geeignet ist als ein anderes. Erklärungen sollen nur relevante Informationen und keine Redundanz enthalten, um zu verhindern, dass ein Aufnahmekanal überlastet wird. Andererseits wurde der Verständnisprozess des Menschen genauer untersucht. Um Menschen effektiv etwas erklären zu können, sollte ihr mentales Modell verwendet werden, was im Kontext von Informationsvisualisierungen vor allem der Aufbau und das Verhalten von Visualisierungen ist. Des Weiteren wurden bestehende Mashups und Visualisierungstools auf ihre Interaktions- und User Assistance Konzepte untersucht. Dabei stellte sich heraus, dass gar nicht so viele Hilfestellungen nötig sind, sofern der Anwender ein korrektes mentales Modell der Anwendung hat. Um das Verständnis der Daten zu fördern, wurden oft Kommentare eingesetzt, deren Wiederverwendbarkeit in anderen Visualisierungen allerdings variierte. Einige Designrichtlinien beinhalteten visuelle und sprachliche Konsistenz, Undo/Redo und wo möglich den mentalen Aufwand des Nutzers zu reduzieren.

Daraufhin wurde ein Konzept bestehend aus acht Hilfestellungen erarbeitet, in einer formativen Nutzerstudie mit Hilfe eines Paper Mockups und fünf Teilnehmern überprüft und danach leicht überarbeitet. Die Hilfestellungen des Konzepts betreffen im Prinzip zwei Aufgaben: Einerseits soll es dem Benutzer erleichtern, Komponenten verschiedener Entwickler zu bedienen und andererseits soll es ihn dabei unterstützen, in Daten enthaltene Informationen aufzunehmen und Erkenntnisse zu gewinnen. Um die Bedienung zu erleichtern, wird der Anwender vor dem Start der Nutzung des Mashups durch jede enthaltene Komponente »geführt«. Dabei werden grund-

legende Informationen wie die Art der Visualisierung, eine Beschreibung und die enthaltenen Daten vermittelt. Damit der Benutzer herausfindet, welche Aktionen wie ausgeführt werden können, wird ein kurzer Comic eingesetzt. Nachdem keine Software frei von Bugs ist, soll der Anwender Fehler in einer Komponente melden können oder allgemeines Feedback geben. Dazu verfasst er eine möglichst ausführliche Textbeschreibung, da er nicht über die nötigen Kenntnisse für vorgefertigte Formulare (»Ursache des Fehlers«) verfügt. Für den Anwender kann die Nutzung eines Mashups verwirrend sein, da andere Komponenten als die aktuell verwendete auf Interaktionen reagieren. Um diese Kommunikation transparenter zu machen, werden einerseits dynamisch für kurze Zeit Pfeile eingeblendet, sobald Nachrichten ausgetauscht wurden. Auf der anderen Seite wird eine statische Version davon in die Hilfe zur Bedienung eingearbeitet, welche ebenfalls mit Comics arbeitet. Es ist völlig offen, welche Konzepte in den Komponenten dargestellt werden und wie viel Kenntnisse der Benutzer darüber verfügt. Aus diesem Grund soll eine Beschreibung eingeblendet werden können, um zum Beispiel zu erklären, wie eine dargestellte Metrik definiert ist. Um auf Ausreißer und Trends in Daten hinzuweisen, können Anwender Kommentare verfassen. Diese beinhalten möglicherweise Annotationen in der Visualisierung und können, sofern die Darstellungsform der Komponente es zulässt, an den Daten selbst gespeichert und in anderen Komponenten wiederverwendet werden. Nachdem ein Anwender auf Grund der vorhandenen Kommunikationskanäle nie isoliert mit einer einzigen Komponente arbeitet, muss die Plattform eine History zur Verfügung stellen, mit der ungewollte Änderungen komponentenübergreifend und einfach rückgängig gemacht werden können. Zuletzt ist es nötig Hilfefunktionen für die Hilfe anzubieten, da diese das Mashup doch viel komplexer macht. Hier wird eine statische Variante bestehend aus Text und Bildern, sowie eine Dynamische eingesetzt, welche mit Machine Learning Algorithmen auf Basis von Nutzerinformationen und Nutzerinteraktionen arbeitet.

Die Bedienungshilfe, Kommunikationshilfe sowie die Kommentarfunktion wurden in einem Prototypen umgesetzt, da sie repräsentativ für die zwei Aufgaben des Hilfesystems sind. Das Frontend wurde mit gängigen Webtechnologien (Backbone.js, D3, jQuery, CSS3) realisiert. Die größte Schwierigkeit war dabei die Art der Darstellung von hervorgehobenen Elementen in der Bedienungshilfe. Nachdem im Konzept ein dunkler Hintergrund vorgesehen war, mussten sie kopiert und über ihren Originale eingefügt werden. Jedoch zeigte sich, dass der Algorithmus, um den Kopien trotzdem das Aussehen der Originale zu geben, sehr aufwändig und unperformant ist. Außerdem konnte keine einheitliche Logik für SVG und HTML Elemente geschrieben werden, da keine der gewählten Bibliotheken beides uneingeschränkt unterstützt. Im CoRe Backend wurde PhantomJS als Headless Browser integriert, welcher die Bilder für die Assistance generiert. Außerdem wurde das DaRe erweitert, sodass es Kommentare speichern und über eine REST API ausgeben kann.

Der Prototyp wurde danach evaluiert. Um zu überprüfen, ob die implementierten Hilfefunktionen verständlich genug sind und um zu sehen, wie sie angenommen werden, wurde eine Nutzerstudie mit 10 zufällig ausgewählten Teilnehmern in einer Starbucks Filiale durchgeführt. Die Testpersonen waren zwischen 18 und 53 Jahren alt und zu 60 % weiblich. Sie mussten fünf Fragen mit Hilfe eines Mashups aus drei Komponenten beantworten und danach einen SUS Usability Fragebogen ausfüllen. Dieser fiel insgesamt gut aus und lässt vermuten, dass er mit mehr Hilfefunktionen

des Konzepts noch besser wäre. Es konnten außerdem ein paar Probleme bei der Nutzung der Hilfe identifiziert werden, denen relativ einfach beigekommen werden kann. Jedoch war vor allem auffällig, dass die Hilfe relativ wenig genutzt wurde, vor allem in Anbetracht der Tatsache, dass die Teilnehmer am Anfang dazu ermuntert wurden. Daraus wird einerseits geschlossen, dass die dynamische Meta-Hilfe nützlich gewesen wäre und andererseits sollte eine weitere Studie mit mehr Daten und Komponenten durchgeführt werden. Ein erklärt Ziel des Hilfesystems war es außerdem, den Entwicklungsaufwand für Komponentenentwickler möglichst gering zu halten. Dazu wurde eine individuell konzipierte Hilfefunktion testweise in die SIMILE Timeline integriert und mit dem Aufwand für das Hilfesystem verglichen. Letzteres konnte 11 Minuten schneller integriert werden, aber vor allem überwiegen andere Vorteile des Hilfesystems: Es ist einfacher wartbar, flexibler und ermöglicht komponentenübergreifend einheitliche Bezeichnungen und Interaktionen.

## 6.2 Ausblick

Der vorgestellte Prototyp kann noch weiter verbessert werden. Zunächst sollten bei der Hervorhebung von aktionsauslösenden UI Elementen keine aufwändigen Kopien mehr erstellt werden, sodass »Layout Thrashing« verhindert und die Hilfefunktion dementsprechend schneller angezeigt werden kann. Zudem müsste für die Comicpanels ein Layoutalgorithmus geschrieben werden, der sowohl »ansprechendes Layout« als auch »effiziente Platznutzung« optimieren kann. Danach müssten im Frontend sowie im Backend bei der Hilfegenerierung parallele und sequentielle Operationen berücksichtigt und Tastaturoperationen unterstützt werden. Dabei kann gleichzeitig die Operation *Drag* so erweitert werden, dass der Entwickler das erwünschte Event Target angeben kann. Außerdem wäre es für Anwender praktisch, wenn sie ihre Kommentare editieren könnten.

Des Weiteren sollten die anderen fünf Teillösungen des Konzepts umgesetzt und in Nutzerstudien evaluiert werden. Einige sollten schnell implementiert sein, da sie nur aus der VISO gelesene Informationen darstellen. Andere, wie zum Beispiel die dynamische Meta-Hilfe oder die History, erfordern hingegen mehr Zeitaufwand, sollten den Anwendern aber auch mehr Nutzen bringen. Zudem sollte noch einmal überprüft werden, ob die eine statische Darstellung der Comicpanels besser geeignet ist die animierte, welche zur Zeit eingesetzt wird.

Danach kann das Konzept um zusätzliche Hilfestellungen erweitert werden. Beispielsweise kann, wie in den verwandten Arbeiten erwähnt, ein dezidierter »Knowledge View« dem Anwender noch mehr beim Verständnis der Daten helfen. Er bietet Benutzern Platz und Werkzeuge, um ihr Wissen zu externalisieren, was den Verständnisprozess fördert. Zusätzlich müsste untersucht werden, ob dieses externalisierte Wissen automatisch in einer datensatzspezifischen Wissensbasis zusammengeführt werden kann. Dann könnten zukünftige Anwender diese Wissensbasis als Ausgangspunkt nehmen und weiter verbessern, was ebenfalls den Verständnisprozess beschleunigt.

# A Anhang

## A.1 Implementierung

### A.1.1 Java Klassendiagramm eines Kommentars

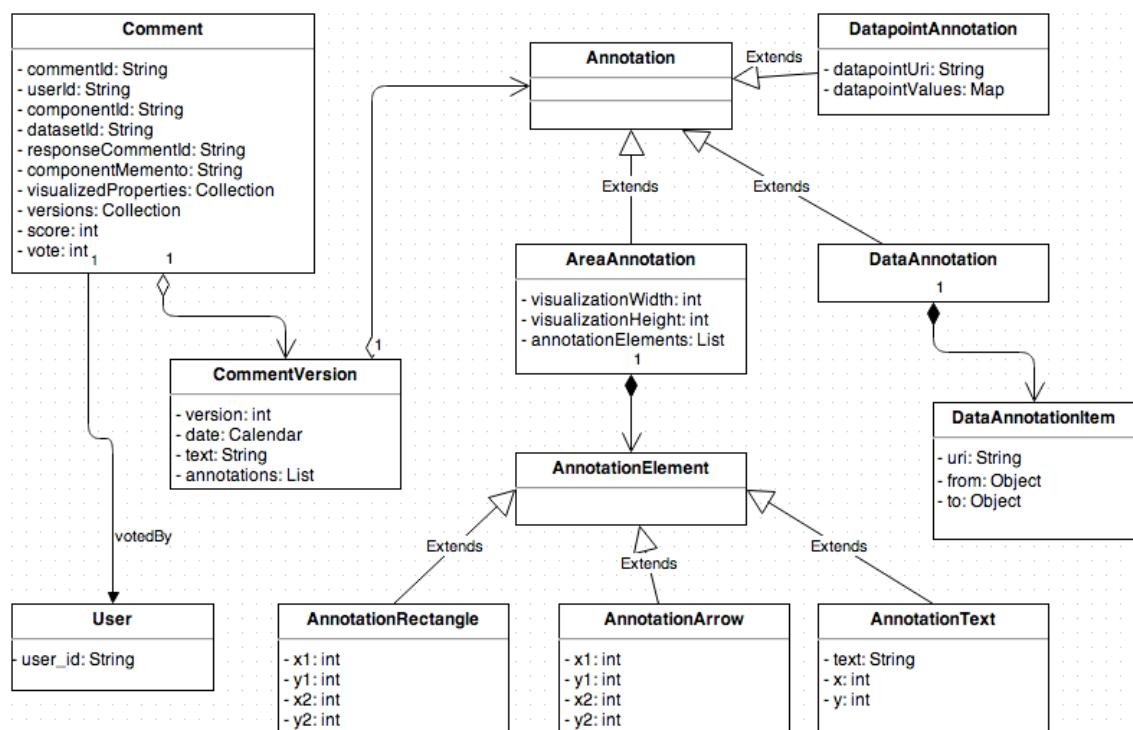


Abbildung A.1: Java Klassendiagramm für Kommentare

### A.1.2 JSON Format eines Kommentars

```

{
    "comment_id": "id",
    "user_id": "bob",
    "dataset_id": "testdataset",
    "component_id": "http://cruise/ui/SimileTimeline",
    "score": 0,
    "response_to": null, // id of other comment
    "memento": {
        "selection": null,
        "centerDate": "2011-09-14T12:00:00.000Z",
        "mousewheelscroll": true,
        "wheelscroll": true
    }
}
  
```

```

        "height": 375,
        "width": 600
    },
15  "visualized_property": [
        // strings of form: dataset + class uri + property uri
    ],
    "versions": [
        {
            "number": 1,
            "text": "This is a test.",
            "timestamp": 1382258459433,
            "annotations": [
                {
                    "uri": "http://dbpedia.org/resource/Date_Movie",
                    "values": {
                        // values of property
                    },
                    "type": "point"
                },
                {
                    "type": "area",
                    "visualization_width": 800,
                    "visualization_height": 352,
                    "elements": [
                        {
                            "type": "arrow",
                            "x1": 56.75,
                            "x2": 58.875,
                            "y1": 20.738636,
40                  "y2": 42.613636
                        },
                        {
                            "type": "text",
                            "x": 51.375,
                            "text": "Testcomment",
                            "y": 17.329546
                        }
                    ]
                },
                {
                    "type": "data",
                    "items": [
                        {
                            "uri": "http://tu-dresden.de/films#release",
                            "from": "2010-01-20T06:00:00.000Z",
55                  "to": "2019-11-24T12:00:00.000Z"
                        }
                    ]
                }
            ]
        }
    ]
}

```

Listing A.1: JSON Format eines Kommentars

### A.1.3 RDF Format eines Kommentars

```

# baseURI: http://mmt.inf.tu-dresden.de/models/comments

@prefix :      <http://mmt.inf.tu-dresden.de/models/
              comments#> .
@prefix owl:   <http://www.w3.org/2002/07/owl#> .
5 @prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns
              #> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .

<http://mmt.inf.tu-dresden.de/models/comments>
10    rdf:type owl:Ontology ;
        owl:versionInfo "Created with TopBraid Composer"^^xsd:
        string .

:Annotation
15    rdf:type owl:Class ;
        rdfs:label "annotation"^^xsd:string ;
        rdfs:subClassOf owl:Thing .

:AnnotationArrow
20    rdf:type :AnnotationElement ;
        rdfs:label "annotation arrow"^^xsd:string .

:AnnotationElement
25    rdf:type owl:Class ;
        rdfs:label "annotation element"^^xsd:string ;
        rdfs:subClassOf owl:Thing .

:AnnotationRectangle
30    rdf:type :AnnotationElement ;
        rdfs:label "annotation rectangle"^^xsd:string .

:AnnotationText
35    rdf:type :AnnotationElement ;
        rdfs:label "text annotation"^^xsd:string .

:AreaAnnotation
40    rdf:type owl:Class ;
        rdfs:label "area annotation"^^xsd:string ;
        rdfs:subClassOf :Annotation .

:Comment
45    rdf:type owl:Class ;
        rdfs:label "comment"^^xsd:string ;
        rdfs:subClassOf owl:Thing .

:CommentVersion

```

```
rdf:type owl:Class ;
rdfs:label "comment version"^^xsd:string ;
rdfs:subClassOf owl:Thing .

50 :DataAnnotation
    rdf:type owl:Class ;
    rdfs:label "data annotation"^^xsd:string ;
    rdfs:subClassOf :Annotation .

55 :DataAnnotationItem
    rdf:type owl:Class ;
    rdfs:label "data annotaiton item"^^xsd:string ;
    rdfs:subClassOf owl:Thing .

60 :DatapointAnnotation
    rdf:type owl:Class ;
    rdfs:label "dataponit annotation"^^xsd:string ;
    rdfs:subClassOf :Annotation .

65 :User
    rdf:type owl:Class ;
    rdfs:label "user"^^xsd:string ;
    rdfs:subClassOf owl:Thing .

70 :Vote
    rdf:type owl:Class ;
    rdfs:label "vote"^^xsd:string ;
    rdfs:subClassOf owl:Thing .

75 :hasAmount
    rdf:type owl:DatatypeProperty ;
    rdfs:domain :Vote ;
    rdfs:label "has amount of votes"^^xsd:string ;
    rdfs:range xsd:integer .

80 :hasAnnotation
    rdf:type owl:ObjectProperty ;
    rdfs:domain :CommentVersion ;
    rdfs:label "has annotation"^^xsd:string ;
    rdfs:range :Annotation .

85 :hasAnnotationElement
    rdf:type owl:ObjectProperty ;
    rdfs:domain :AreaAnnotation ;
    rdfs:label "has annotation element"^^xsd:string ;
    rdfs:range :AnnotationElement .

90 :hasComment
    rdf:type owl:ObjectProperty ;
    rdfs:domain :Vote ;

95 :hasComment
```

```
rdfs:label "has comment"^^xsd:string ;
rdfs:range :Comment .

:hasCommentId
100    rdf:type owl:DatatypeProperty ;
rdfs:domain :Comment ;
rdfs:label "has comment id"^^xsd:string ;
rdfs:range xsd:string .

105 :hasComponentId
     rdf:type owl:DatatypeProperty ;
rdfs:domain :Comment ;
rdfs:label "has component id"^^xsd:string ;
rdfs:range xsd:string .

110 :hasCreatorId
     rdf:type owl:DatatypeProperty ;
rdfs:domain :Comment ;
rdfs:label "has creator id"^^xsd:string ;
rdfs:range xsd:string .

115 :hasDataAnnotationItem
     rdf:type owl:ObjectProperty ;
rdfs:domain :DataAnnotation ;
rdfs:label "has data annotation item"^^xsd:string ;
rdfs:range :DataAnnotationItem .

120 :hasDatapointURI
     rdf:type owl:DatatypeProperty ;
rdfs:domain :DatapointAnnotation ;
rdfs:label "has datapoint uri"^^xsd:string ;
rdfs:range xsd:string .

125 :hasDatapointValue
     rdf:type owl:DatatypeProperty ;
rdfs:domain :DatapointAnnotation ;
rdfs:label "has datapoint value"^^xsd:string ;
rdfs:range xsd:string .

130 :hasDatasetId
     rdf:type owl:DatatypeProperty ;
rdfs:domain :Comment ;
rdfs:label "has dataset id"^^xsd:string ;
rdfs:range xsd:string .

135 :hasDate
     rdf:type owl:DatatypeProperty ;
rdfs:domain :CommentVersion ;
rdfs:label "has date"^^xsd:string ;
rdfs:range xsd:date .

140
145
```

```
:hasFrom
    rdf:type owl:DatatypeProperty ;
    rdfs:domain :DataAnnotationItem ;
150    rdfs:label "has from value"^^xsd:string .

:hasMemento
    rdf:type owl:DatatypeProperty ;
    rdfs:domain :Comment ;
155    rdfs:label "has memento"^^xsd:string ;
    rdfs:range xsd:string .

:hasNumber
    rdf:type owl:DatatypeProperty ;
160    rdfs:domain :CommentVersion ;
    rdfs:label "has (revision) number"^^xsd:string ;
    rdfs:range xsd:integer .

:hasProperty
165    rdf:type owl:DatatypeProperty ;
    rdfs:domain :DataAnnotationItem ;
    rdfs:label "has property"^^xsd:string ;
    rdfs:range xsd:string .

170 :hasScore
    rdf:type owl:DatatypeProperty ;
    rdfs:domain :Comment ;
    rdfs:label "has score"^^xsd:string ;
    rdfs:range xsd:integer .
175 :hasText
    rdf:type owl:DatatypeProperty ;
    rdfs:domain :CommentVersion ;
    rdfs:label "has text"^^xsd:string ;
180    rdfs:range xsd:string .

:hasTo
    rdf:type owl:DatatypeProperty ;
    rdfs:domain :DataAnnotationItem ;
185    rdfs:label "has to value"^^xsd:string .

:hasUser
    rdf:type owl:ObjectProperty ;
    rdfs:domain :Vote ;
190    rdfs:label "has user"^^xsd:string ;
    rdfs:range :User .

:hasUserId
    rdf:type owl:DatatypeProperty ;
195    rdfs:domain :User ;
```

```

    rdfs:label "has user id"^^xsd:string ;
    rdfs:range xsd:string .

:hasVersion
200   rdf:type owl:ObjectProperty ;
      rdfs:domain :Comment ;
      rdfs:label "has version"^^xsd:string ;
      rdfs:range :CommentVersion .

205 :hasVisualizedProperty
      rdf:type owl:DatatypeProperty ;
      rdfs:domain :Comment ;
      rdfs:label "has visualized property"^^xsd:string ;
      rdfs:range xsd:string .

210 :hasX1
      rdf:type owl:DatatypeProperty ;
      rdfs:domain :AnnotationElement ;
      rdfs:label "has x1 value"^^xsd:string ;
      rdfs:range xsd:float .

215 :hasX2
      rdf:type owl:DatatypeProperty ;
      rdfs:domain :AnnotationElement ;
      rdfs:label "has x2 value"^^xsd:string ;
      rdfs:range xsd:float .

220 :hasY1
      rdf:type owl:DatatypeProperty ;
      rdfs:domain :AnnotationElement ;
      rdfs:label "has y1 value"^^xsd:string ;
      rdfs:range xsd:float .

225 :hasY2
      rdf:type owl:DatatypeProperty ;
      rdfs:domain :AnnotationElement ;
      rdfs:label "has y2 value"^^xsd:string ;
      rdfs:range xsd:float .

230 :isResponseTo
      rdf:type owl:DatatypeProperty ;
      rdfs:domain :Comment ;
      rdfs:label "is response to comment"^^xsd:string ;
      rdfs:range xsd:string .

```

Listing A.2: RDF Format eines Kommentars in Turtle

# Literaturverzeichnis

- [Ado13] Adobe. *Adobe roadmap for the Flash runtimes*. White Paper. März 2013 (siehe Seite 47).
- [Ala+03] Harith Alani, Sanghee Kim, David E. Millard u. a. »Automatic ontology-based knowledge extraction from web documents«. In: *Intelligent Systems* 18.1 (2003), Seiten 14–21 (siehe Seite 64).
- [Bau11] Christian Bauer. »Ein Framework zur nutzerorientierten Erklärung hybrider Pläne«. Magisterarbeit. Universität Ulm, Juni 2011 (siehe Seite 22).
- [Ber07] M. Bergman. *An intrepid guide to ontologies*. 2007. URL: <http://www.mkbergman.com/374/an-intrepid-guide-to-ontologies/> (besucht am 26.05.2013) (siehe Seite 12).
- [BKM09] Aaron Bangor, Philip Kortum und James Miller. »Determining what individual SUS scores mean: Adding an adjective rating scale«. In: *Journal of Usability Studies* 4.3 (2009), Seiten 114–123 (siehe Seite 110).
- [BOH11] Michael Bostock, Vadim Ogievetsky und Jeffrey Heer. »D3: data-driven documents«. In: *Transactions on Visualization and Computer Graphics* 17.12 (2011), Seiten 2301–2309 (siehe Seiten 47, 82).
- [Bro96] John Brooke. »SUS - A quick and dirty usability scale«. In: *Usability Evaluation in Industry* 189 (1996), Seite 194 (siehe Seite 109).
- [Cao+10] Jill Cao, Kyle Rector, Thomas H. Park u. a. »A debugging perspective on end-user mashup programming«. In: *Symposium on Visual Languages and Human-Centric Computing*. IEEE, 2010, Seiten 149–156 (siehe Seite 28).
- [Cap+11] Cinzia Cappiello, Maristella Matera, Matteo Picozzi u. a. »DashMash: a mashup environment for end user development«. In: *Web Engineering*. Springer, 2011, Seiten 152–166 (siehe Seiten 1, 24).
- [Che73] Herman Chernoff. »The use of faces to represent points in k-dimensional space graphically«. In: *Journal of the American Statistical Association* 68.342 (1973), Seiten 361–368 (siehe Seite 17).
- [Chi00] Ed H. Chi. »A taxonomy of visualization techniques using the data state reference model«. In: *Symposium on Information Visualization*. IEEE, 2000, Seiten 69–75 (siehe Seite 17).
- [Cho+13] Soudip Roy Chowdhury, Olexiy Chudnovskyy, Matthias Niederhausen u. a. »Complementary assistance mechanisms for end user mashup composition«. In: *Proceedings of the 22nd international conference on World Wide Web companion*. WWW 2013. 2013, Seiten 269–272 (siehe Seite 25).

- [Chu+12] Olexiy Chudnovskyy, Tobias Nestler, Martin Gaedke u. a. »End-user-oriented telco mashups: The OMELETTE approach«. In: *Proceedings of the 21st international conference companion on World Wide Web*. WWW 2012. ACM, 2012, Seiten 235–238 (siehe Seiten 1, 25).
- [Chu+13] Olexiy Chudnovskyy, Stefan Pietschmann, Matthias Niederhausen u. a. »Awareness and control for inter-widget communication: Challenges and solutions«. In: *Web Engineering*. Springer, 2013, Seiten 114–122 (siehe Seite 28).
- [CHW06] Chao-Min Chiu, Meng-Hsiang Hsu und Eric T. G. Wang. »Understanding knowledge sharing in virtual communities: An integration of social capital and social cognitive theories«. In: *Decision Support Systems* 42.3 (Dez. 2006), Seiten 1872–1888 (siehe Seite 61).
- [CJB99] Balakrishnan Chandrasekaran, John R. Josephson und V. Richard Benjamins. »What are ontologies, and why do we need them?« In: *Intelligent Systems and Their Applications* 14.1 (1999), Seiten 20–26 (siehe Seite 11).
- [CKW12] Parmit K. Chilana, Andrew J. Ko und Jacob O. Wobbrock. »LemonAid: selection-based crowdsourced contextual help for web applications«. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI 2012. ACM, 2012, Seiten 1549–1558 (siehe Seite 29).
- [Cla94] Richard E. Clark. »Media will never influence learning«. In: *Educational Technology Research and Development* 42.2 (1994), Seiten 21–29 (siehe Seite 21).
- [CMS99] Stuart K. Card, Jock D. Mackinlay und Ben Shneiderman. *Readings in information visualization: Using vision to think*. Morgan Kaufmann Publications, 1999 (siehe Seite 13).
- [Cor+13] Amélie Cordier, Marie Lefevre, Pierre-Antoinene Champin u. a. »Trace-based reasoning: Modeling interaction traces for reasoning on experiences«. In: *Proceedings of the 26th International FLAIRS Conference*. FLAIRS 2013. 2013 (siehe Seite 72).
- [CTC09] Chia-Jung Chan, Ruck Thawonmas und Kuan-Ta Chen. »Automatic storytelling in comics: A case study on World of Warcraft«. In: *Extended Abstracts on Human Factors in Computing Systems*. CHI 2009. ACM, 2009, Seiten 3589–3594 (siehe Seiten 29, 32).
- [CYR09] Yang Chen, Jing Yang und William Ribarsky. »Toward effective insight management in visual analytics systems«. In: *Visualization Symposium*. PacificVis 2009. IEEE, 2009, Seiten 49–56 (siehe Seite 64).
- [EB12] Micheline Elias und Anastasia Bezerianos. »Annotating BI visualization dashboards: needs & challenges«. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI 2012. ACM, 2012, Seiten 1641–1650 (siehe Seiten 27, 28, 31).

- [FCK12] Kristie Fisher, Scott Counts und Aniket Kittur. »Distributed sense-making: Improving sensemaking by leveraging the efforts of previous users«. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI 2012. ACM, 2012, Seiten 247–256 (siehe Seite 23).
- [FJ10] Camilla Forsell und Jimmy Johansson. »An heuristic set for evaluation in information visualization«. In: *Proceedings of the International Conference on Advanced Visual Interfaces*. AVI 2010. ACM, 2010, Seiten 199–206 (siehe Seite 30).
- [Fol10] Mary Jo Foley. *Microsoft: Our strategy with Silverlight has shifted*. Okt. 2010. URL: <http://www.zdnet.com/blog/microsoft/microsoft-our-strategy-with-silverlight-has-shifted/7834> (besucht am 10.06.2013) (siehe Seite 47).
- [Gam+94] Erich Gamma, Richard Helm, Ralph Johnson u. a. *Design patterns: Abstraction and reuse of object-oriented design*. Pearson Education, 1994 (siehe Seite 59).
- [Ges12] Johannes F. Gesell. *Textuelle Repräsentation von Planerklärungen*. Nov. 2012 (siehe Seiten 22, 29).
- [GGW89] Ruth Garner, Mark G. Gillingham und C. Stephen White. »Effects of seductive details on macroprocessing and microprocessing in adults and children«. In: *Cognition and Instruction* 6.1 (1989), Seiten 41–57 (siehe Seite 21).
- [GLB02] Olivier Gapenne, Charles Lenay und Dominique Boullier. »Defining categories of the human/technology coupling: Theoretical and methodological issues«. In: *Adjunct Proceedings of the 7th ERCIM Workshop on User Interface for All*. ERCIM 2002. 2002, Seiten 197–198 (siehe Seite 20).
- [Gra12] Lars Grammel. »User interfaces supporting information visualization novices in visualization construction«. Dissertation. University of Victoria, 2012 (siehe Seiten 23, 29, 40, 65).
- [Gru95] Thomas R. Gruber. »Toward principles for the design of ontologies used for knowledge sharing«. In: *International Journal of Human-Computer Studies* 43.5 (1995), Seiten 907–928 (siehe Seite 11).
- [GS06] Dennis P. Groth und Kristy Streefkerk. »Provenance and annotation for visual exploration systems«. In: *Transactions on Visualization and Computer Graphics* 12.6 (2006), Seiten 1500–1510 (siehe Seite 26).
- [GS10] Lars Grammel und Margaret-Anne Storey. »Choosel: Web-based visualization construction and coordination for information visualization novices«. In: *Information Visualization Conference*. Band 10. InfoVis 2010. IEEE, 2010 (siehe Seite 26).
- [HBO10] Jeffrey Heer, Michael Bostock und Vadim Ogievetsky. »A tour through the visualization zoo«. In: *Communications of the ACM* 53.6 (2010), Seiten 59–67 (siehe Seite 13).

- [Hee+08] Jeffrey Heer, Jock Mackinlay, Chris Stolte u. a. »Graphical histories for visualization: Supporting analysis, communication, and evaluation«. In: *Transactions on Visualization and Computer Graphics* 14.6 (Nov. 2008), Seiten 1189–1196 (siehe Seiten 30, 48, 65, 71).
- [HVW07] Jeffrey Heer, Fernanda Viégas und Martin Wattenberg. »Voyagers and voyeurs: Supporting asynchronous collaborative information visualization«. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI 2007. ACM, 2007, Seiten 1029–1038 (siehe Seiten 27, 31, 56, 59).
- [ID91] Alfred Inselberg und Bernard Dimsdale. »Parallel coordinates«. In: *Human-Machine Interactive Systems*. Springer, 1991, Seiten 199–233 (siehe Seite 13).
- [ISO98] ISO. *ISO 9241-11: Ergonomic requirements for office work with visual display terminals: Part 11 - Guidance on usability*. 1998 (siehe Seite 106).
- [Jos06] S. Josefsson. *RFC 4648: The Base16, Base32, and Base64 Data Encodings*. 2006. URL: <http://tools.ietf.org/pdf/rfc4648.pdf> (besucht am 05.09.2013) (siehe Seite 97).
- [KAK95] Daniel A Keim, Mihael Ankerst und Hans-Peter Kriegel. »Recursive pattern: A technique for visualizing very large amounts of data«. In: *Proceedings of the 6th conference on Visualization*. Visualization 1995. IEEE, 1995, Seiten 279–286 (siehe Seite 17).
- [Kal08] Slava Kalyuga. »Relative effectiveness of animated and static diagrams: An effect of learner prior knowledge«. In: *Computers in Human Behavior* 24.3 (Mai 2008), Seiten 852–861 (siehe Seite 22).
- [Kei02] Daniel A. Keim. »Information visualization and visual data mining«. In: *Transactions on Visualization and Computer Graphics* 8.1 (2002), Seiten 1–8 (siehe Seite 13).
- [KK09] Andrea E. Kohlhase und Michael Kohlhase. »Semantic transparency in user assistance systems«. In: *Proceedings of the 27th ACM International Conference on Design of Communication*. SIGDOC 2009. ACM, 2009, Seiten 89–96 (siehe Seiten 22, 42).
- [KMH06a] Gary Klein, Brian Moon und Robert R. Hoffman. »Making Sense of Sensemaking 1: Alternative Perspectives«. In: *Intelligent Systems* 21.4 (2006), Seiten 70–73 (siehe Seite 23).
- [KMH06b] Gary Klein, Brian Moon und Robert R. Hoffman. »Making Sense of Sensemaking 2: A Macrocognitive Model«. In: *Intelligent Systems* 21.5 (2006), Seiten 88–92 (siehe Seite 23).
- [Koh+11] Jörn Kohlhammer, Daniel Keim, Margit Pohl u. a. »Solving problems with visual analytics«. In: *Procedia Computer Science* 7 (2011), Seiten 117–120 (siehe Seiten 13, 14).
- [Kot07] Sotiris B. Kotsiantis. »Supervised machine learning: A review of classification techniques«. In: *Informatica* 31 (2007), Seiten 3–24 (siehe Seite 72).

- [Koz94] Robert B. Kozma. »Will media influence learning? Reframing the debate«. In: *Educational Technology Research and Development* 42.2 (1994), Seiten 7–19 (siehe Seite 21).
- [KSR13] Sandeep Kaur Kuttal, Anita Sarma und Gregg Rothermel. »Debugging support for end user mashup programming«. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI 2013. ACM, 2013, Seiten 1609–1618 (siehe Seite 30).
- [KSS96] David Kurlander, Tim Skelly und David Salesin. »Comic Chat«. In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH 1996. ACM, 1996, Seiten 225–236 (siehe Seiten 29, 32).
- [Kuu96] Kari Kuutti. »Activity theory as a potential framework for human-computer interaction research«. In: *Context and Consciousness: Activity Theory and Human Computer Interaction* (1996), Seiten 17–44 (siehe Seite 43).
- [LD09] Brian Y. Lim und Anind K. Dey. »Assessing demand for intelligibility in context-aware applications«. In: *Proceedings of the 11th International Conference on Ubiquitous Computing*. Ubicomp 2009. ACM, 2009, Seiten 195–204 (siehe Seite 28).
- [Lia+06] Wenhui Liao, Weihong Zhang, Zhiwei Zhu u. a. »Toward a decision-theoretic framework for affect recognition and user assistance«. In: *International Journal of Human-Computer Studies* 64.9 (2006), Seiten 847–873 (siehe Seite 72).
- [LJ05] Xiangyang Li und Qiang Ji. »Active affective state detection and user assistance with dynamic Bayesian networks«. In: *Transactions on Systems, MAN, and Cybernetics - Part A: Systems and Humans* 35.1 (2005), Seiten 93–105 (siehe Seite 72).
- [LS10] Zhicheng Liu und John T. Stasko. »Mental models, visual reasoning and interaction in information visualization: A top-down perspective«. In: *Transactions on Visualization and Computer Graphics* 16.6 (2010), Seiten 999–1008 (siehe Seiten 23, 29).
- [MA91] Richard E. Mayer und Richard B. Anderson. »Animations need narrations: An experimental test of a dual-coding hypothesis«. In: *Journal of educational psychology* 83.4 (1991), Seiten 484–490 (siehe Seite 21).
- [Maz09] Riccardo Mazza. *Introduction to information visualization*. Springer, 2009 (siehe Seite 40).
- [McC08] Scott McCloud. *Google Chrome: Behind the open source browser project*. Google, 2008. URL: <http://www.scottmccloud.com/googlechrome/> (siehe Seite 22).
- [McC94] Scott McCloud. *Understanding comics: The invisible art*. William Morrow Paperbacks, 1994 (siehe Seiten 22, 45).

- [MCS12] Donald Matheson, George M. Coghill und Somayujulu Sripada. »Integrating natural language generation and model-based reasoning for explanation generation«. In: *Proceedings of the 12th UK Workshop on Computational Intelligence*. 2012, Seiten 1–7 (siehe Seiten 22, 29).
- [MG90] Richard E. Mayer und Joan K. Gallini. »When is an illustration worth ten thousand words«. In: *Journal of Educational Psychology* 82.4 (1990), Seiten 715–726 (siehe Seite 22).
- [MM02a] Richard E. Mayer und Roxana Moreno. »Aids to computer-based multimedia learning«. In: *Learning and Instruction* 12.1 (2002), Seiten 107–119 (siehe Seiten 21, 22).
- [MM02b] Richard E. Mayer und Roxana Moreno. »Animation as an aid to multimedia learning«. In: *Educational Psychology Review* 14.1 (2002), Seiten 87–99 (siehe Seite 22).
- [MM99] Roxana Moreno und Richard E. Mayer. »Cognitive principles of multimedia learning: The role of modality and contiguity«. In: *Journal of Educational Psychology* 91 (1999), Seiten 358–368 (siehe Seite 21).
- [Nie94] Jakob Nielsen. »Heuristic evaluation«. In: *Usability Inspection Methods*. John Wiley & Sons. 1994, Seiten 25–62 (siehe Seite 30).
- [Nor86] Donald A Norman. »Cognitive engineering«. In: *User-centered System Design* (1986), Seiten 31–61 (siehe Seite 29).
- [Nov07] Jasminko Novak. »Helping knowledge cross boundaries: Using knowledge visualization to support cross-community sensemaking«. In: *Proceedings of the 40th Hawaii International Conference on System Sciences*. IEEE, 2007, Seiten 38–38 (siehe Seite 23).
- [Pha+13] Tuan Pham, Julia Jones, Ronald Metoyer u. a. »Interactive visual analysis promotes exploration of long-term ecological data«. In: *Ecosphere* 4.9 (2013) (siehe Seite 1).
- [Pic12] Nikolaus Piccolotto. *Navigation und Selektion in großen semantischen Datensätzen*. Mai 2012 (siehe Seite 9).
- [Pie+08] Stefan Pietschmann, Annett Mitschick, Ronny Winkler u. a. »CroCo: ontology-based, cross-application context management«. In: *Proceedings of the 3rd International Workshop on Semantic Media Adaptation and Personalization*. SMAP 2008. IEEE, 2008, Seiten 88–93 (siehe Seite 10).
- [Pie+09] Stefan Pietschmann, Martin Voigt, Andreas Rümpel u. a. »CRUISe: Composition of rich user interface services«. In: *Web Engineering*. Springer, 2009, Seiten 473–476 (siehe Seite 7).
- [Pie12] Stefan Pietschmann. »Modellgetriebene Entwicklung adaptiver, komponentenbasierter Mashup-Anwendungen«. Dissertation. TU Dresden, 2012 (siehe Seiten 1, 7–10, 64, 69).
- [Pui09] Eric Puidokas. *Maximum value of z-index*. Apr. 2009. URL: <http://www.puidokas.com/max-z-index/> (besucht am 04.10.2013) (siehe Seite 91).

- [PV13] Jan Polowinski und Martin Voigt. »VISO: A shared, formal knowledge base as a foundation for semi-automatic information visualization systems«. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI 2013. ACM, 2013 (siehe Seite 8).
- [QF05] Yan Qu und George W. Furnas. »Sources of structure in sensemaking«. In: *Extended Abstracts on Human Factors in Computing Systems*. CHI 2005. ACM, 2005, Seiten 1989–1992 (siehe Seite 23).
- [RBM13] Carsten Radeck, Gregor Blichmann und Klaus Meißner. »CapView: Functionality-aware visual mashup development for non-programmers«. In: *Proceedings of the 13th International Conference on Web Engineering*. ICWE 2013. Springer, 2013 (siehe Seite 93).
- [Rib+13] Hrvoje Ribicic, Jürgen Waser, Raphael Fuchs u. a. »Visual analysis and steering of flooding simulations«. In: *Transactions on Visualization and Computer Graphics* 19.6 (Juni 2013), Seiten 1062–1075 (siehe Seite 1).
- [RJ86] Lawrence Rabiner und Biing-Hwang Juang. »An introduction to hidden Markov models«. In: *ASSP Magazine* 3.1 (1986), Seiten 4–16 (siehe Seite 72).
- [Rob07] Jonathan C. Roberts. »State of the art: Coordinated and multiple views in exploratory visualization«. In: *Proceedings of the 5th International Conference on Coordinated and Multiple Views in Exploratory Visualization*. CMV 2007. IEEE, 2007, Seiten 61–71 (siehe Seite 1).
- [RRD07] Jörg Rech, Eric Ras und Björn Decker. »Intelligent assistance in german software development: A survey«. In: *Software* 24.4 (2007), Seiten 72–79 (siehe Seiten 20, 21).
- [Rüm+11] Andreas Rümpel, Carsten Radeck, Gregor Blichmann u. a. »Towards DIY development of composite web applications«. In: *Proceedings of the International Conference on Internet Technologies and Society*. Band 12. 2011, Seiten 231–235 (siehe Seite 8).
- [Rum13] Bernard Rummel. *System Usability Scale (Translated into German)*. Apr. 2013. URL: <http://www.sapdesignguild.org/resources/sus.asp> (besucht am 12.10.2013) (siehe Seite 109).
- [SBS94] Maarten W. van Someren, Yvonne F. Barnard und Jacobijn A. C. Sandberg. *The think aloud method: A practical guide to modelling cognitive processes*. Academic Press, 1994 (siehe Seiten 28, 36).
- [SCB98] Deborah F. Swayne, Dianne Cook und Andreas Buja. »XGobi: Interactive dynamic data visualization in the X window system«. In: *Journal of Computational and Graphical Statistics* 7.1 (1998), Seiten 113–130 (siehe Seite 18).
- [Sch+13] Kurt Schlegel, Rita L. Sallam, Daniel Yuen u. a. *Magic quadrant for business intelligence and analytics platforms*. Feb. 2013. URL: <http://www.gartner.com/technology/reprints.do?id=1-1DZLPEP&ct=130207&st=sb> (besucht am 01.07.2013) (siehe Seite 25).

- [Seb02] Fabrizio Sebastiani. »Machine learning in automated text categorization«. In: *Computing Surveys* 34.1 (2002), Seiten 1–47 (siehe Seiten 50, 64).
- [SH02] Chris Stolte und Pat Hanrahan. »Polaris: A system for query, analysis and visualization of multi-dimensional relational databases«. In: *Transactions on Visualization and Computer Graphics* 8.1 (Jan. 2002), Seiten 1–14 (siehe Seite 25).
- [Shn92] Ben Shneiderman. »Tree visualization with tree-maps: 2-d space-filling approach«. In: *Transactions on Graphics* 11.1 (1992), Seiten 92–99 (siehe Seiten 13, 17).
- [Shn96] Ben Shneiderman. »The eyes have it: A task by data type taxonomy for information visualizations«. In: *Symposium on Visual Languages*. IEEE, 1996, Seiten 336–343 (siehe Seiten 27, 30).
- [Smi12] Mark Smiciklas. *The power of infographics: Using pictures to communicate and connect with your audiences*. Que Publishing, 2012 (siehe Seite 13).
- [SMP98] John Sweller, Jeroen J. G. van Merriënboer und Fred G. W. C. Paas. »Cognitive architecture and instructional design«. In: *Educational Psychology Review* 10.3 (1998), Seiten 251–296 (siehe Seite 22).
- [SRL06] Ariel Shamir, Michael Rubinstein und Tomer Levinboim. »Generating comics from 3D interactive computer graphics«. In: *Computer Graphics and Applications* 26.3 (2006), Seiten 53–61 (siehe Seite 29).
- [SSZ11] Nelly Schuster, Raffael Stein und Christian Zirpins. »A mashup tool for collaborative engineering of service-oriented enterprise documents«. In: *Information Systems Evolution*. Springer, 2011, Seiten 166–173 (siehe Seite 26).
- [SW08] Yedendra Babu Srinivasan und Jarke J. van Wijk. »Supporting the analytical reasoning process in information visualization«. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI 2008. ACM, 2008, Seiten 1237–1246 (siehe Seite 27).
- [Uma10] Karthikeyan Umapathy. »Requirements to support collaborative sense-making«. In: *CSCW CIS Workshop*. Band 10. CSCW 2010. 2010 (siehe Seite 23).
- [VFM13] Martin Voigt, Martin Franke und Klaus Meißner. »Capturing and reusing empirical visualization knowledge«. In: *Proceedings of the 1st International Workshop on User-Adaptive Visualization*. WUAV 2013. 2013 (siehe Seite 50).
- [Vie+07] Fernanda B. Viegas, Martin Wattenberg, Frank van Ham u. a. »Many Eyes: A site for visualization at internet scale«. In: *Transactions on Visualization and Computer Graphics* 13.6 (2007), Seiten 1121–1128 (siehe Seiten 27, 31).
- [Vir92] Robert A. Virzi. »Refining the test phase of usability evaluation: How many subjects is enough?« In: *Human Factors* 34.4 (1992), Seiten 457–468 (siehe Seite 107).

- [Voi+12] Martin Voigt, Stefan Pietschmann, Lars Grammel u. a. »Context-aware recommendation of visualization components«. In: *Proceedings of the 4th International Conference on Information, Process, and Knowledge Management*. eKNOW 2012. 2012, Seiten 101–109 (siehe Seite 42).
- [VPM13] Martin Voigt, Stefan Pietschmann und Klaus Meißner. »A semantics-based, end-user-centered information visualization process for semantic web data«. In: *Semantic Models for Adaptive Interactive Systems*. Springer, 2013 (siehe Seiten 1, 8).
- [VSK96] Robert A. Virzi, Jeffrey L. Sokolov und Demetrios Karis. »Usability problem identification using both low- and high-fidelity prototypes«. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI 1996. ACM, 1996, Seiten 236–243 (siehe Seite 35).
- [W3C] W3C. *RFC 2616*. URL: <http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html> (besucht am 13.08.2013) (siehe Seite 97).
- [W3C12] W3C. *RDFa Core 1.1 - Syntax and processing rules for embedding RDF through attributes*. Juni 2012. URL: <http://www.w3.org/TR/rdfa-core/> (besucht am 25.06.2013) (siehe Seite 60).
- [Web+12] Erika Noll Webb, Gayathri Balasubramanian, Ultan O’Broin u. a. »Wham! Pow! Comics as user assistance«. In: *Journal of Usability Studies* 7.3 (2012), Seiten 105–117 (siehe Seiten 22, 29).
- [Wij05] Jarke J. van Wijk. »The value of visualization«. In: *Visualization*. VIS 2005. IEEE, 2005, Seiten 79–86 (siehe Seiten 13, 14).
- [WMS11] Anna Wong, Nadine Marcus und John Sweller. »Instructional animations: More complex to learn from than at first sight?« In: *Proceedings of the 13th IFIP TC 13 International Conference on Human-Computer Interaction*. INTERACT 2011. Springer, 2011, Seiten 552–555 (siehe Seite 22).
- [WTC12] Ming-Hui Wen, Ruck Thawonmas und Kuan-Ta Chen. »Pomics: A computer-aided storytelling system with automatic picture-to-comics composition«. In: *Proceedings of the 2012 Conference on Technologies and Applications of Artificial Intelligence*. TAAI 2012. IEEE, 2012, Seiten 314–318 (siehe Seiten 29, 32).
- [Yee+03] Ka-Ping Yee, Kirsten Swearingen, Kevin Li u. a. »Faceted metadata for image search and browsing«. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. CHI 2003. ACM. 2003, Seiten 401–408 (siehe Seite 18).
- [Yeh+11] Tom Yeh, Tsung-Hsiang Chang, Bo Xie u. a. »Creating contextual help for GUIs using screenshots«. In: *Proceedings of the 24th annual ACM Symposium on User Interface Software and Technology*. UIST 2011. ACM, 2011, Seiten 145–154 (siehe Seite 29).