Universität Leipzig
Wirtschaftswissenschaftliche Fakultät
Institut für Wirtschaftsinformatik
Prof. Dr. Ulrich Eisenecker
Dipl.-Wirtsch.-Inf. Pascal Kovacs

Thema

Integration von Quelltext in 3D-Softwarevisualisierungen

Bachelorarbeit zur Erlangung des akademischen Grades
Bachelor of Science – Informatik

vorgelegt von: Annanias, Yves

Email-Adresse: yvesannanias@web.de

Leipzig, den 06. Oktober 2015

Abstract

Die Einarbeitung in ein bestehendes Softwareprojekt ist mit einigen Schwierigkeiten behaftet. Um das Verständnis über Aufbau und Funktionsweise von Software zu fördern, werden die zugrundeliegenden, meist abstrakten Daten oft mithilfe einer Metapher in eine verständlichere visuelle Form übertragen. Ein entscheidender Bereich, der sich dieser Aufgabe annimmt, ist die Softwarevisualisierung. Mithilfe der Visualisierung bietet sich ein Überblick auf ein gesamtes Softwareprojekt.

Zur Verbesserung des Verständnisses ist es auch notwendig, einen direkten Bezug vom abstrakten Quellcode zu den Bestandteilen der Visualisierung zu ziehen. Dadurch können erkannte Auffälligkeiten sofort am Quellcode untersucht werden. Ziel dieser Arbeit war es, den Quellcode in die Visualisierung zu integrieren und damit diesen Bezug herzustellen. Zusätzlich wurde die Darstellung des Quellcodes durch das Einbinden des Frameworks *Prism* mit einem Syntax-Highlighting versehen. Das Syntax-Highlighting erhöht dessen Lesbarkeit und bietet dem Benutzer damit eine bessere Orientierung. Der Bezug wurde gleichzeitig dadurch gestärkt, dass der Benutzer mit dem Quellcode auf die gleiche Weise interagieren kann, wie mit den Elementen innerhalb der Visualisierung.

Schlüsselwörter

Softwarevisualisierung, 3D, Recursive Disc Metaphor, Syntax-Highlighting

Gliederung

Gliederung I

Abbildungsverzeichnis II

Quellcode-Listing III

Tabellenverzeichnis IV

Abkürzungsverzeichnis V

1 Einleitung 1

1.1 Motivation 1
1.2 Zielstellung 2
1.3 Aufbau der Arbeit 3

2 Stand der Forschung 4

2.1 Softwarevisualisierung 4
2.2 Der Generator 9
2.3 Extensible 3D und X3DOM 11
2.4 Die Oberfläche 13

3 Theoretische Grundlagen 16

3.1 Anzeigen des Quellcodes 16
3.2 Der HTML-Standard 18
3.3 JavaScript 19
3.4 Reguläre Ausdrücke 20

4 Syntax-Highlighter 22

4.1 Syntax-Highlighting 22
4.2 Evaluation eines Syntax-Highlighters 24
4.3 Prism 25

5 Erweiterung des Prototyps 30

5.1 Funktionsweise 30
5.2 Implementierung 33

5.2.1 Architektur der Oberfläche 33
5.2.2 Laden und Anzeigen des Quellcodes 35
5.2.3 Hervorheben des selektierten Elements 37
5.2.4 Interaktion mit dem Quellcode 38
5.2.5 Darstellung im Extra-Fenster 41

6 Fazit und Ausblick 43

Literaturverzeichnis VI

Abbildungsverzeichnis

Abbildung 2-1: Balkendiagramm zu den Messdaten 5
Abbildung 2-2: Darstellung von Graphen 8
Abbildung 2-3: Cone-Tree 8
Abbildung 2-4: Visualisierungspipeline 10
Abbildung 2-5: Wuefel.x3d im Browser 12
Abbildung 2-6: Die Oberfläche 13
Abbildung 2-7: Aufbau der Recursive Disc Metaphor 14
Abbildung 4-1: Syntax-Highlighting 23
Abbildung 4-2: Mit Prism hervorgehobener Quellcode 26
Abbildung 4-3: Ausgabe von Prism in HTML 27
Abbildung 5-1: Darstellung des Quellcodes 30
Abbildung 5-2: Hervorhebung eines Attributs 31
Abbildung 5-3: Elementselektion im Quellcode 32

Quellcode-Listing

Quellcode-Listing 2-1: Würfel in X3D 11
Quellcode-Listing 2-2: Verwendung von X3DOM 11
Quellcode-Listing 3-1: JavaScript in HTML 19
Quellcode-Listing 3-2: Beispiele für reguläre Ausrücke 21
Quellcode-Listing 4-1: Einbindung von Prism 26
Quellcode-Listing 4-2: Prism und BeispielMarkup 27
Quellcode-Listing 4-3: Style für BeispielMarkup 28
Quellcode-Listing 4-4: Funktion extend für BeispielMarkup 28
Quellcode-Listing 4-5: Plugin für Prism 29
Quellcode-Listing 5-1: Plugin zum Anzeigen des Quellcodes 33
Quellcode-Listing 5-2: onEntitySelected 35
Quellcode-Listing 5-3: displayCode 36
Quellcode-Listing 5-4: textNodesToSpan 37
Quellcode-Listing 5-5: highlighSelectedElement 38
Quellcode-Listing 5-6: addInteraction 39
Quellcode-Listing 5-7: addInteractionForMethods 41
Quellcode-Listing 5-8: Mapperfunktion im Extra-Fenster 42

Tabellenverzeichnis

Tabelle 2-1: Messdaten über Niederschlagsmenge in [mm] 5

Tabelle 3-1: Sonderzeichen in regulären Ausdrücken 20

Tabelle 4-1: Vergleich der Frameworks 25

Tabelle 5-1: Reguläre Ausdrücke aus der Signatur einer Methode 40

Abkürzungsverzeichnis

3D	dreidimensional
HTML	Hypertext Markup Language
ID	Identifikator
IETF	Internet Engineering Task Force
IP	Internetprotokoll
OMG	Object Management Group
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UML	Unified Modeling Language
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WIMP	Windows, Icons, Menus, Pointer
WWW	World Wide Web
X3D	Extensible 3D
XML	Extensible Markup Language

1 Einleitung

1.1 Motivation

Software hat spezielle Eigenschaften, die entscheidende ist dabei ihre Immaterialität. Das heißt, sie ist weder greif- noch sichtbar. Aufgrund dessen stimmt auch oft die Dokumentation nicht exakt mit dem dazugehörigen Quellcode überein (vgl. [Balzert 2009a]). Dieser abstrakte Charakter führt häufig zum Abbruch vieler Softwareprojekte (vgl. [Balzert 2009a]). Daher beschreibt H. Balzert [2009a] in seinem Lehrbuch der Softwaretechnik die Notwendigkeit und Relevanz der Planung und Dokumentation von Softwareprojekten sowie die Kommunikation aller Stakeholder eines Projektes untereinander. Dies wird anhand der Definition von Software deutlich: „*Software (kurz SW): Menge von Programmen oder Daten zusammen mit begleitenden Dokumenten, die für ihre Anwendung notwendig oder hilfreich sind*“ [Hesse et al. 1984].

Nun haben aber nicht alle Stakeholder die gleichen Kenntnisse über Software, da sie verschiedene Rollen, wie zum Beispiel Softwareentwickler, Auftraggeber und Benutzer, innehaben. Nicht einmal alle Entwickler eines Projektes müssen über den gleichen Kenntnisstand verfügen. So kann sich die Kommunikation der Stakeholder über die Software als problematisch erweisen, weil die Softwaredokumentation nicht immer als Kommunikationsgrundlage ausreicht. Einen anderen Ansatz bietet dazu die Softwarevisualisierung. Durch Stilmittel wie Formen und Farben veranschaulicht sie die immaterielle Software. Daher findet sie auch zur Planung von Software Verwendung. So sind beispielsweise Fluss- und Baumdiagramme bereits aus anderen Bereichen bekannt, wodurch eine gemeinsame Kommunikationsbasis für alle Stakeholder geschaffen wird. Dadurch können die aufbereiteten Daten unabhängig vom vorhandenem Fachwissen schnell erfasst werden.

Aufgrund dessen wurde am Institut für Wirtschaftsinformatik der Universität Leipzig ein Generator entwickelt, der Software in ihrer Struktur erfassen und in eine dreidimensionale Visualisierung überführen kann, aus welcher der Betrachter einen Überblick über Komplexität und Aufbau der Software erlangt. Woran es der Visualisierung zum jetzigen Stand noch mangelt, ist die Integration des ihr zugrunde liegenden Quellcodes in seiner ursprünglichen Textform, was vor allem für diejenigen Stakeholder interessant ist, die eine höhere Fachkenntnis besitzen.

1.2 Zielstellung

Die Erweiterung der Visualisierung um diese Funktionalität sowie die Implementierung der darauf aufbauenden Interaktion zwischen Visualisierung und Quellcode ist das Ziel dieser Arbeit. Dieses Ziel wird nachstehend über die Definition von vier Teilzielen genauer in seinen Einzelheiten beschrieben. Die Teilziele erhalten mit der Angabe von */ZN/* eine Nummerierung, um auch in späteren Kapiteln gezielt auf diese referenzieren zu können:

- */Z10/* Anzeigen des Quellcodes des zugrundeliegenden Softwareprojektes
- */Z20/* Erweiterung der Textdarstellung um Syntax-Highlighting
- */Z30/* Interaktion zwischen der Visualisierung und deren Quellcode
- */Z40/* Interaktion innerhalb des Quellcodes

Das Teilziel */Z10/* bildet die Grundlage für die anderen drei Teilziele. Dabei geht es darum, die entsprechenden Quellcodedateien zu laden und in einer Oberfläche anzuzeigen. Innerhalb dieser ist dafür ein geeigneter Bereich für die Darstellung zu identifizieren.

Teilziel */Z20/* soll die Anzeige des Quellcodes erweitern. Dabei wird dieser, ähnlich wie Java in Eclipse, mit einem Syntax-Highlighting versehen, um so dessen Lesbarkeit zu erhöhen. Da die Entwicklung eines eigenen Highlighters einen großen Mehraufwand bedeutet, wird daher ein schon bestehendes, möglichst leicht anpassbares und frei verfügbares Framework verwendet, welches diese Aufgabe übernimmt.

Durch das Auswählen eines Elements in der Visualisierung soll durch das Teilziel */Z30/* das entsprechende Element im Quellcode hervorgehoben werden. Die Anzeige wird dabei durch folgendes Vorgehen angepasst: Handelt es sich bei dem selektierten Element um eine Klasse, dann wird deren gesamter Quellcode geladen, mit einem Syntax-Highlighting versehen und angezeigt. Bei der Selektion einer Methode oder eines Attributs soll ebenfalls der Quellcode der zugehörigen Klasse, wie eben beschrieben, angezeigt werden. Zusätzlich erfolgt eine Hervorhebung aller Vorkommen der Methode beziehungsweise des Attributs im Quellcode. Bei der Selektion eines Packages erfolgt dagegen keine Anzeige von Quellcode.

Die bisherige Interaktion wirkt sich auf die Darstellung innerhalb des Quellcodes aus. Das vierte Teilziel */Z40/* ergänzt die Visualisierung um eine weitere Interaktionsmöglichkeit, welche sich auf den Quellcode selbst bezieht und sich auf die Darstellung der Visualisierung auswirkt. Ein Klick mit der Maus auf die Definition einer

Methode oder eines Attributs im Quellcode soll sich dabei genauso verhalten, als wenn in der Visualisierung eine Methode beziehungsweise ein Attribut selektiert wurde. Dabei muss das entsprechende Element in der Visualisierung hervorgehoben sowie alle zugehörigen Informationen angezeigt werden.

1.3 Aufbau der Arbeit

Nach der Einleitung wird in Kapitel 2 erläutert, worum es sich bei der Visualisierung, im speziellen bei der Softwarevisualisierung, handelt. Dabei wird auch der schon erwähnte Generator in seiner prototypischen Umsetzung von Müller näher betrachtet. Ebenso wird die Oberfläche vorgestellt, welche die Darstellung der Visualisierung übernimmt und welche die Grundlage für die Umsetzung der Ziele ist.

Im dritten Kapitel werden die theoretischen Grundlagen gelegt. Es wird festgelegt wo der Quellcode liegt und wie dieser geladen wird. Dabei werden mögliche Probleme erfasst und deren Lösung beschrieben. Zusätzlich erfolgt eine Einführung in den HTML-Standard, der bei der Umsetzung eingehalten werden muss. Weiterhin wird die Sprache JavaScript vorgestellt, welche zur Erweiterung der Oberfläche genutzt wird.Abschließend folgt ein kurzer Überblick über reguläre Ausdrücke gegeben.

Im vierten Kapitel wird das Thema Syntax-Highlighting vorgestellt. Anschließend erfolgt die Evaluation eines geeigneten Frameworks für das Syntax-Highlighting. Dabei werden verschiedener Frameworks miteinander verglichen. Anhand verschiedener Kriterien wird das für diese Aufgabe am besten geeignete ausgewählt sowie im Anschluss im Detail näher vorgestellt.

Im fünften Kapitel werden zuerst die Erweiterungen an der Oberfläche präsentiert. Danach folgt eine Einführung in die Architektur und das Eventsystem der Oberfläche. Im Rahmen dieser erfolgt die Umsetzung, die im Anschluss beschrieben wird.

Das sechste Kapitel beschließt die Arbeit mit einer Zusammenfassung und gibt einen Ausblick auf weitere Ideen für Interaktionen, die für zukünftige Entwicklungen und für das bessere Arbeiten mit der Visualisierung als sinnvoll erachtet werden.

2 Stand der Forschung

In diesem Kapitel wird der Begriff der Visualisierung eingeführt. Es zeigt dabei die Bedeutung und den Nutzen von Visualisierungen auf. Anschließend wird erklärt, wie diese Verwendung finden, um Software selbst zu visualisieren. Betrachtet werden verschiedene Aspekte der Darstellung. Weiterhin wird der Vorteil von Interaktionsmöglichkeiten aufgezeigt. Dies bildet den Übergang zum bereits erwähnten Generator. Ausgehend vom Quellcode eines Softwareprojektes erzeugt dieser Visualisierungen, die im Anschluss von einer speziellen Oberfläche dargestellt werden. Diese wiederum wird im letzten Abschnitt dieses Kapitels vorgestellt.

2.1 Softwarevisualisierung

Softwarevisualisierung ist ein Teilgebiet der Visualisierung. Zentrales Thema der Visualisierung ist dabei die graphische Darstellung von meist abstrakten Daten, beispielsweise Texten, Messdaten und den in ihnen beinhalteten Relationen. So lässt sich Visualisierung wie folgt definieren: *„The use of computer-supported, interactive, visual representations of data to amplify cognition"* [Card et al. 1999]. Durch eine andere Darstellungsform der Daten soll das Verständnis dieser verbessert werden. Gegebenenfalls sollen weitere Informationen über diese Daten gewonnen werden, was als zentrales Ziel der Visualisierung gilt (vgl. [Müller 2009]). Sie nutzt dabei die menschlichen Fähigkeiten im Bereich der Wahrnehmung, zum Beispiel beim Erkennen von Formen und unterschiedlichen Farben in einer Darstellung (vgl. [Müller 2009]).

Daten können auf unterschiedliche Weise visualisiert werden. So bieten sich unter anderem Balkendiagramme für Messdaten an. Das nachfolgende Beispiel soll dies an fiktiven Messdaten über Niederschlagsmengen in Leipzig im Vergleich zum Regenwald verdeutlichen. Die eher abstrakten Daten, die als Grundlage der anschließenden Visualisierung dienen, werden in Tabelle 2-1 dargestellt. Es handelt sich dabei um drei Spalten. In der ersten Spalte erfolgt eine Auflistung der Monate eines Jahres. In der Zweiten sind die Messwerte für Leipzig eingetragen und in der dritten Spalte die Messwerte für den Regenwald jeweils zu den Monaten. Ein Ziel soll es sein, Vergleiche der Werte innerhalb einer Region und zwischen den beiden Regionen zu erstellen. Interessante Fragen sind dabei, in welchem Monat der Niederschlag am größten war, in welchem am geringsten oder ob er eher gleich verteilt über das ganze Jahr auftrat.

	Leipzig	Regenwald
Jan	100	400
Feb	90	350
Mär	80	420
Apr	90	400
Mai	100	550
Jun	120	400
Jul	130	200
Aug	120	150
Sep	100	100
Okt	90	100
Nov	90	150
Dez	100	300

Tabelle 2-1: Messdaten über Niederschlagsmenge in [mm]

Die Antworten zu den Fragen können dabei direkt aus der Tabelle entnommen werden. Eine geeignete Visualisierung der Daten kann diese greifbarer machen, um einen besseren Überblick zu geben. Verdeutlicht wird dies in Abbildung 2-1. Die Visualisierung als Balkendiagramm ist dabei recht einfach und stellt keine neuen Herausforderungen dar. Die Darstellungsform ist dem Betrachter geläufig. Er muss sich nicht lange in diese einarbeiten, um zu verstehen, aus welchen Bestandteilen sie besteht. In diesem Fall werden einfache Balken in den Farben blau und rot dargestellt. Aus der rechts stehenden Legende ist zu entnehmen, dass blau für Leipzig und rot für den Regenwald steht. Jeweils zwei Balken stehen direkt nebeneinander für jeweils einen Monat. Entlang der x-Achse wurden die Monate abgetragen, die Streckung der Balken entlang der y-Achse repräsentiert dabei die Werte der gemessenen Niederschlagswerte.

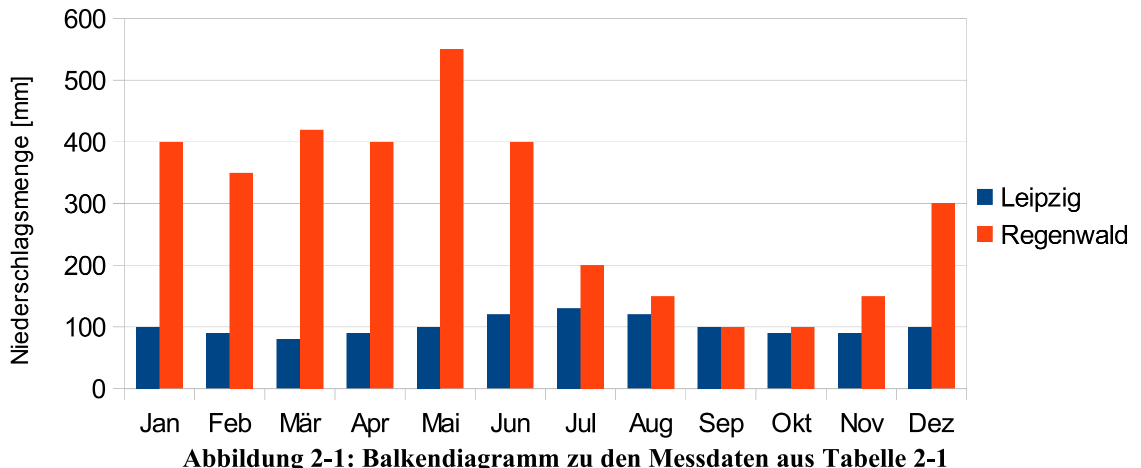

Abbildung 2-1: Balkendiagramm zu den Messdaten aus Tabelle 2-1

Die obigen Fragen lassen sich nun sogar besser aus dem Diagramm heraus beantworten. Um beispielsweise die Frage nach dem Monat mit dem höchsten Niederschlag zu beantworten, müssten bei der Tabelle 2-1 erst alle Werte der Spalten visuell aufwendiger miteinander verglichen werden. Im Diagramm ist das Ergebnis dagegen auf den ersten Blick deutlich sichtbarer: Der Balken im Mai für den Regenwald hat die größte Ausdehnung.

Visualisierungen müssen dabei nicht immer die gesamte Datenmenge umfassen. Sie können sich auch auf bestimmte Bereiche konzentrieren oder sogar einzelne Informationen weniger detailliert darstellen. Wie in obiger Abbildung 2-1 erkennbar, ist hierbei der konkrete Zahlenwert nicht eindeutig feststellbar. Dafür ist die Skalierung an der y-Achse zu grob. Hier greift der Aspekt der Interaktivität der Visualisierung, welcher in der Definition von Card genannt wird. So wäre es vorteilhaft, wenn der Betrachter des Diagramms die Möglichkeit hat, die Skalierung an der y-Achse selbst zu bestimmen. Durch eine solche Skalierung könnten auch die Unterschiede der Balkenausdehnung für Leipzig noch besser verdeutlicht werden. Und die eingangs gestellte Beispielfrage nach dem niederschlagsreichsten Monat ließe sich damit ebenso schnell beantworten. Weitere Interaktionen könnten dabei die Ausblendung ganzer Bereiche sein, um andere gezielt sowie näher zu betrachten. Diese Interaktion wird durch das *Filtering* ermöglicht.

Card unterteilt die Visualisierung in zwei Teilgebiete. Das wichtigste Unterscheidungsmerkmal sind dabei die Daten, die die Grundlage der Visualisierung darstellen (vgl. [Card et al. 1999]). So liegen der wissenschaftlichen Visualisierung physische Daten zugrunde. Physische Daten werden aus natürlichen Prozessen gewonnen. Ein solcher Prozess ist beispielsweise das Messen von Niederschlagswerten in einer Region. Die Messdaten bilden dann die physischen Daten. Das obige Beispiel ist damit der wissenschaftlichen Visualisierung zuzuordnen. Im Gegensatz dazu steht die Informationsvisualisierung. Ihre zugrundeliegenden Daten sind abstrakt und können nicht direkt zu natürlichen Prozessen zugeordnet werden. Aufgrund des abstrakten Charakters von Software, lässt sich die Softwarevisualisierung der Informationsvisualisierung zuordnen (vgl. [Diehl 2007]).

Es gibt mehrere Versuche Softwarevisualisierung zu definieren. Die nachfolgende Definition spiegelt dabei die in dieser Arbeit gezeigten Aspekte am besten wider: *„Software visualisation is a discipline that makes use of various forms of imagery to provide insight and understanding and to* <u>*reduce complexity*</u> *of the existing software*

system under consideration" [Knight/Munro 1999, Hervorhebung im Original]. Mit der Definition von Software werden dabei allerdings nicht alle relevanten Bereiche abgedeckt. Daher wird die Definition der Softwarevisualisierung durch Diehl erweitert, der Softwarevisualisierung als *„[...] the visualization of artifacts related to software and its development process [...]"* [Diehl 2007] auffasst. Artefakte sind hierbei alle Dokumente, die während der Entwicklung von Software erzeugt werden. Bisher zählten dazu der Quellcode und die dazugehörige Dokumentation. Zusätzlich umfassen diese nun auch alle weiteren Dokumente, wie beispielsweise Entwurfsmuster, Änderungsentscheidungen und ebenso verschiedene Versionen des Quellcodes. Aus dieser erweiterten Definition lassen sich weitere Ziele ableiten. So soll durch die Visualisierung eine einfache Möglichkeit geboten werden, einen schnellen Einblick in ein Softwaresystem zu bekommen. Gleichzeitig dessen Struktur und Aufbau zu erfassen und anhand daraus erzielter neuer Erkenntnisse die Weiterentwicklung der Software zu steuern (vgl. [Müller 2009]).

Wie schon bereits in Abbildung 2-1 gezeigt, ist die Verwendung von Diagrammen eine Möglichkeit der Visualisierung. Diagramme finden häufig Anwendung in Softwarevisualisierungen, so zum Beispiel durch die *Unified Modeling Language* (kurz UML). UML ist eine standardisierte Sprache zur Modellierung sowie Dokumentation von Ausschnitten einer Software und wird von der *Object Management Group* (kurz OMG) verwaltet (vgl. [OMG 2015]). Ein Bereich der UML sind zum Beispiel Klassendiagramme. Sie dienen der graphischen Betrachtung und Modellierung von Klassen, ihren Attributen und Methoden sowie Relationen zwischen den Klassen. Konkreten Quellcode zeigen sie dabei nicht an (vgl. [Balzert 2009a]). Dadurch ist es auch für Stakeholder mit geringerer Fachkenntnis möglich, in kurzer Zeit einen Überblick über Umfang und Funktionalitäten der Software zu gewinnen. Weiterführende Informationen zur UML und Klassendiagrammen finden sich in der Literatur, beispielsweise bei Heide Balzert „*UML 2 in 5 Tagen*" [Balzert 2009b].

Klassendiagramme können dabei als Graphen aufgefasst werden. Graph-basierte Modelle sind eine weitere Möglichkeit der Softwarevisualisierung. Ein Graph dient dazu, bestimmte Elemente einer Anschauung, die sogenannten Knoten (beispielsweise Klassen) durch Kanten (beispielsweise Relationen zwischen den Klassen) miteinander in Beziehung zu setzen. Wie ein Graph aber letztendlich visualisiert wird, ist von dieser Beschreibung unabhängig. Häufig werden einfache geometrische Formen verwendet, die die Knoten

darstellen, welche über Linien miteinander verbunden sind. Beispiele sind in Abbildung 2-2 dargestellt.

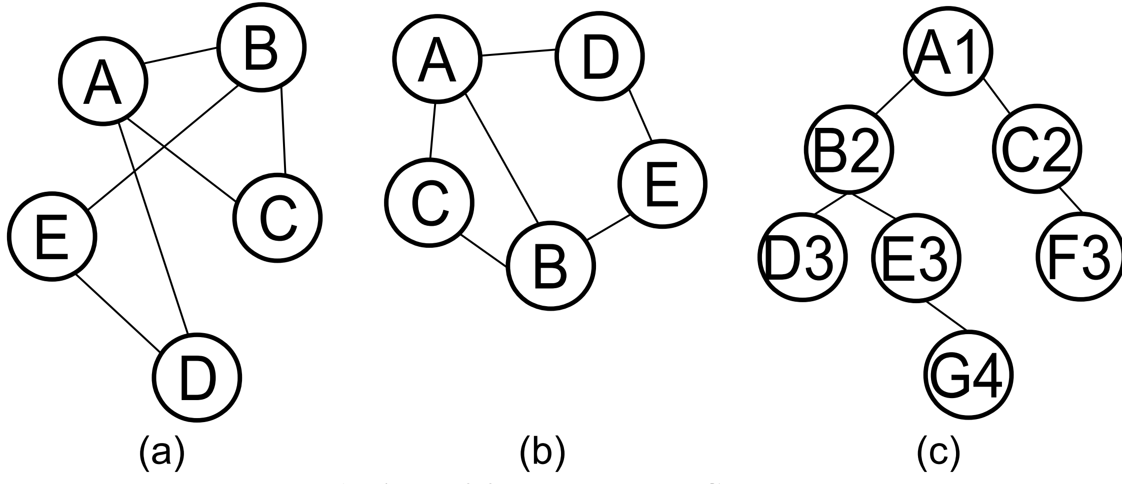

Abbildung 2-2: Darstellung von Graphen

In obiger Abbildung stellen (a) und (b) jeweils den gleichen Graphen dar, nur die Anordnung der Knoten variiert. So überschneiden sich in (b) keine Kanten mehr und dieser ist gegenüber dem Graphen (a) auch kompakter. Die Knoten in (c) sind demgegenüber hierarchisch angeordnet. Die Anordnung der Knoten auf verschiedenen Ebenen liefert weitere Informationen über die Relationen zu Knoten der Ebene darüber und darunter sowie zu Knoten auf der gleichen Ebene. Diese Art der Visualisierung wird umso schneller erfasst, je geringer die Anzahl der Kantenüberschneidungen ausfällt.

Neben der oben beschriebenen zweidimensionalen Softwarevisualisierung kann diese auch auf eine weitere Dimension ausgedehnt werden, wenn es die Übersichtlichkeit verbessert. Ein Beispiel hierfür sind sogenannte *cone trees* wie in Abbildung 2-3.

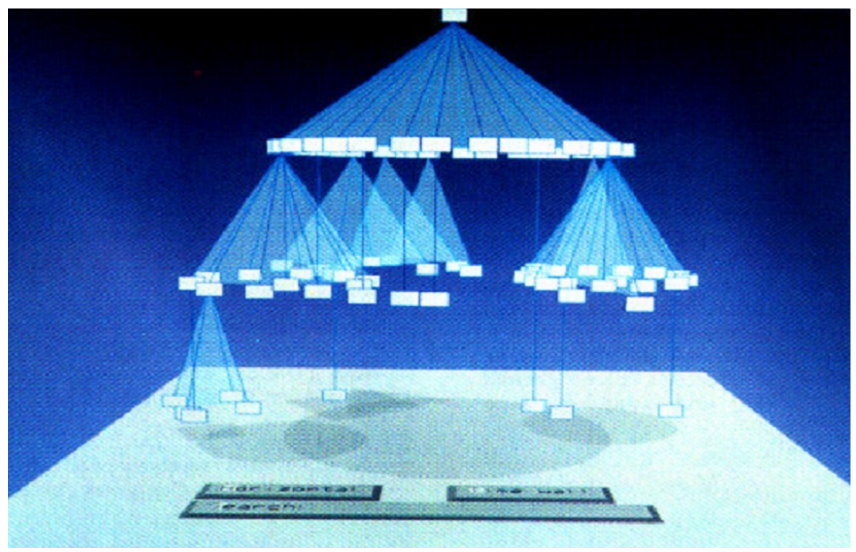

Abbildung 2-3: Cone-Tree [Robertson et al. 1991]

Die Veranschaulichung kann noch weiter verbessert werden, wenn auf Metaphern zurückgegriffen wird. Diese versuchen einen Sachverhalt durch schon bekannte, verständlichere Beziehungen, Darstellungen oder Begrifflichkeiten auszudrücken. So werden oft Metaphern in der Benutzung von Programmen genutzt. Das Symbol *X* steht dabei beispielsweise für das Schließen eines Fensters, das Symbol *Diskette* für das Speichern von Daten. Symbole werden von verschiedenen Programmen benutzt und erfüllen in diesen stets die gleichen Funktionen. Weil einige dieser Funktionen in ähnlicher Weise schon aus vielen anderen Programmen bekannt sind, erleichtert sich dadurch die Einarbeitungszeit in ein neues Programm (vgl. [Müller 2009]). Aus diesem Grund bauen graphische Benutzerschnittstellen oft auf dem *WIMP*-Prinzip auf. Es beschreibt die einheitliche Verwendung der vier Bereiche *Windows, Icons, Menus* und *Pointer* (vgl. [WIMP 2014]).

Um eine Visualisierung zu erstellen, müssen die zugrunde liegenden Daten aufbereitet werden. Im Falle der hier verwendeten *Recursive Disc Metaphor*, welche die dritte Dimension einschließt, wird diese Arbeit von einem Generator automatisch verrichtet. Dabei dienen ihm die Quellcodedateien als Eingabe. Über verschiedene Schritte wird daraus die Visualisierung erstellt, die in einer speziellen Oberfläche betrachtet werden kann. Die einzelnen Schritte werden im folgenden Abschnitt 2.2 beschrieben.

2.2 Der Generator

Müller beschreibt in seiner Arbeit die Grundlagen für die Entwicklung eines Generators zum Erstellen von Softwarevisualisierungen und setzt diese in einem Prototypen um (vgl. [Müller 2009]). Dabei benennt er das Problem, dass bisherige Lösungen vom Entwicklungsprozess der Software entkoppelt sind und noch nicht vollständig automatisiert ablaufen. Des Weiteren wird ein Format für das 3D-Modell benötigt, welches sich leicht in andere Umgebungen importieren und exportieren lässt (vgl. [Müller 2009]).

Der Generator wurde dabei als Plugin für die Entwicklungsumgebung Eclipse konzipiert, da diese eine häufig genutzte Umgebung für die Erstellung von Softwareprojekten ist (vgl. [Müller 2009]). Dadurch wird das Problem der Entkopplung vom Entwicklungsprozess abgeschwächt. Eine Anforderung an das Format der Visualisierung ist dabei die Plattformunabhängigkeit, um so eine bessere Portabilität zu erreichen. Der von ihm entwickelte Prototyp orientiert sich dabei an der Visualisierungspipeline (vgl. [Haber/McNabb 1990]), die in der folgenden Abbildung 2-4 dargestellt wird.

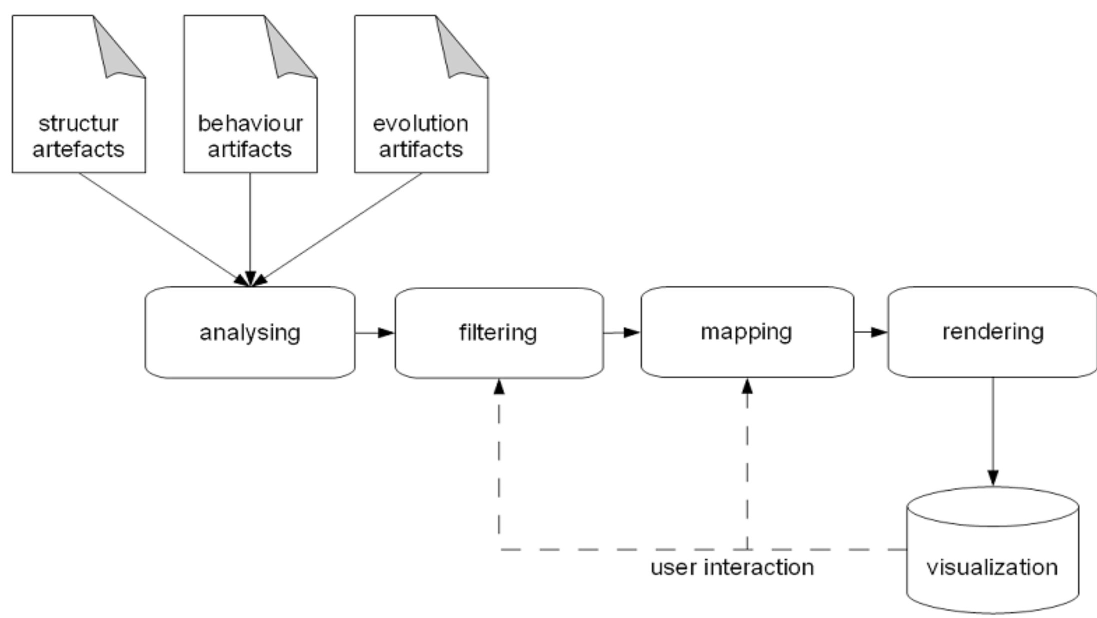

Abbildung 2-4: Visualisierungspipeline [Kovacs 2010]

Die Grundlage für die Visualisierungspipeline bilden Softwareartefakte. Diese werden im Verlauf verschiedener Phasen in eine Visualisierung übersetzt. Die erste Phase ist dabei die Analyse, in welcher alle zugrundeliegenden Daten auf Vollständigkeit und Korrektheit überprüft werden. Die aufbereiteten Daten werden der nächsten Phase, dem *Filtering*, übergeben. Es ist nicht immer notwendig alles zu visualisieren, da dadurch die Übersichtlichkeit leiden könnte. Mit dem *Filtering* erhält der Benutzer eine Möglichkeit der Interaktion im Bezug auf die Selektion. Hier können aus der Ausgangsmenge der Daten über verschiedene Kriterien Teilmengen gebildet werden, die jeweils den Bedürfnissen verschiedener Stakeholder entsprechen. Nur diese selektierten Informationen werden der *Mapping*-Phase übergeben.

In dieser Hauptphase des gesamten Visualisierungsprozesses werden die Daten auf geometrische Formen abgebildet. Die jeweiligen Formen werden dann mit Attributen (wie Farben, Ausdehnung und Anordnung zueinander) entsprechend der verwendeten Metapher versehen. In der letzten Phase der Pipeline, dem *Rendering*, werden diese geometrischen Formen in Bilddaten transformiert. Das *Rendering* wird von einer speziellen Oberfläche übernommen. Innerhalb dieser Oberfläche hat der Benutzer weitere Interaktionsmöglichkeiten.

2.3 Extensible 3D und X3DOM

Die in der *Mapping*-Phase erzeugten Daten werden vom prototypischen Generator von Müller im *Extensible-3D*-Format (kurz X3D) gespeichert. Jenes Format ist plattform- sowie anwendungsunabhängig und wird vom Web3D-Konsortium verwaltet (vgl. [Web3D 2015a]). Damit entspricht es den eingangs genannten Anforderungen. X3D ist dabei eine Beschreibungssprache für 3D-Modelle und basiert auf der *Extensible Markup Language* (kurz XML) (vgl. [Web3D 2015b]). In dieser Form können virtuelle 3D-Welten in einer Baumstruktur modelliert werden. Eine Szene besteht aus verschiedenen Elementen, wie verschiedenen Grundformen, Anweisungen für Farben, Licht und Ausdehnungen. Quellcode-Listing 2-1 zeigt einen Ausschnitt einer solchen X3D-Datei. Dabei wird ein einfacher grauer Würfel mit der Ausdehnung von zwei Längeneinheiten entlang der drei Achsen modelliert.

```
1     <Transform>
2           <Shape>
3                 <Appearance>
4                       <Material diffuseColor="1 1 1"/>
5                 </Appearance>
6                 <Box size="2 2 2"/>
7           </Shape>
8     </Transform>
```

Quellcode-Listing 2-1: Würfel in X3D

Zur Darstellung dieser Szene wird ein X3D-Browser benötigt. Dieser rendert das Modell entsprechend der Anweisungen der X3D-Datei. Das Web3D-Konsortium bietet eine große Übersicht über mehrere verschiedene solcher Browser (vgl. [Web3D 2015c]). Für die normalen Web-Browser existieren Plugins, um auch diese für die Darstellung zu verwenden. Eine weitere Möglichkeit einen Web-Browser zu verwenden, dabei aber auf die Installation eines Plugins zu verzichten, ist die Verwendung des Frameworks X3DOM [X3DOM 2015]. Quellcode-Listing 2-2 zeigt anhand eines einfaches Beispiel, wie X3DOM als Script in eine HTML-Seite eingebunden wird (Zeile 1).

```
1     <script src='http://www.x3dom.org/download/x3dom.js'></script>
2     <x3d width='600px' height='600px'>
3           <scene>
4                 <Inline nameSpaceName="Wuefel" mapDEFToID="true"
5                       url="wuerfel.x3d" />
6           </scene>
7     </x3d>
```

Quellcode-Listing 2-2: Verwendung von X3DOM

Der 3D-Würfel kann nun einfach in die Seite geladen werden (Zeile 5). X3DOM übernimmt die Darstellung automatisch. Der Vorteil hierbei liegt darin, dass die Elemente der X3D-Szene als HTML-Elemente verwendet werden können. Zusätzlich ist es möglich die HTML-Events, wie *onclick*, für Interaktionen mit der Szene zu verwenden. Das Ergebnis ist in Abbildung 2-5 dargestellt.

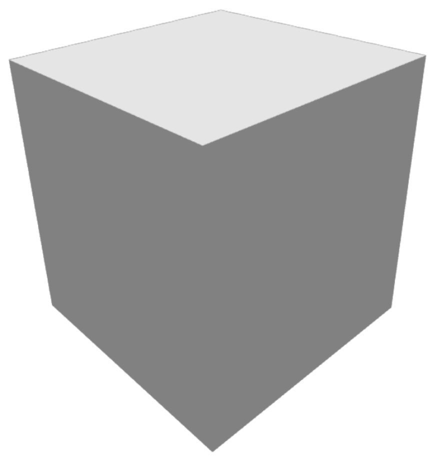

Abbildung 2-5: Wuefel.x3d im Browser

2.4 Die Oberfläche

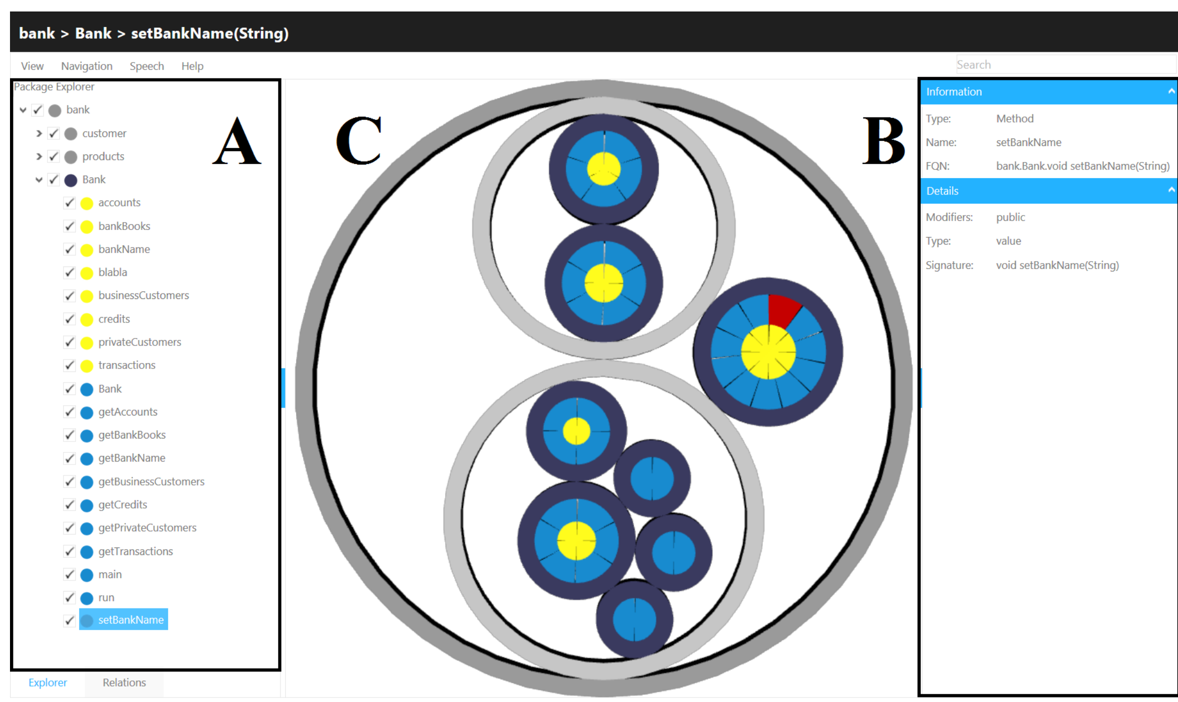

Abbildung 2-6: Die Oberfläche

Da der Generator das 3D-Modell im plattformunabhängigen X3D-Format abspeichert, bietet es sich an, auch die Visualisierung plattformunabhängig umzusetzen. Das geschieht durch die Realisierung der Oberfläche als Website, die von einem modernen Browser dargestellt wird. Durch die Verwendung von X3DOM ist dabei kein spezieller Browser mehr notwendig. Abbildung 2-6 zeigt diese Darstellung. Es ist zu erkennen, dass die Oberfläche hauptsächlich in drei Bereiche unterteilt ist: In den *Filtering*-Bereich (A), den *Details-on-demand*-Bereich (B) und den *Overview- / Rendering*-Bereich (C).

Im *Overview*-Bereich wird das dreidimensionale Modell dargestellt. Die Anordnungen, Formen und Farben der Elemente spielen ein wichtige Rolle und werden von der zugrundeliegenden Metapher bestimmt. Auf eine Metapher wird zurückgegriffen, da Software ein immaterielles Produkt mit einem stark abstrakten Charakter ist. Sie soll das Softwareprodukt durch eine greifbare Visualisierung veranschaulichen. Die hier verwendete Metapher ist die *Recursive Disc Metaphor*, welche von Müller und Zeckzer ausgearbeitet wurde (vgl. [Müller/Zeckzer 2015]).

Sie soll, auf Glyphen basierend, Software sichtbar machen. Alle in Abbildung 2-6 im *Overview*-Bereich dargestellten graphischen Elemente sind solche Glyphen. Sie besitzen zwei Arten von Eigenschaften: Einerseits geometrische Eigenschaften wie Größe, Position sowie Orientierung. Andererseits besitzen sie das Aussehen beschreibende

Eigenschaften wie Farbe und Transparenz. In prozeduralen und objektorientierten Sprachen besteht Software aus mehreren Packages, Klassen, Methoden und Attributen. Diese stehen aber nicht einfach so im Raum, sondern unterliegen bestimmten Beziehungen untereinander. So enthält jede Klasse ihre eigenen Methoden und Attribute sowie gegebenenfalls noch innere Klassen. Ein Package umschließt mehrere Klassen und weitere Packages. So entsteht die hierarchische Struktur, die im Package Explorer ersichtlich ist.

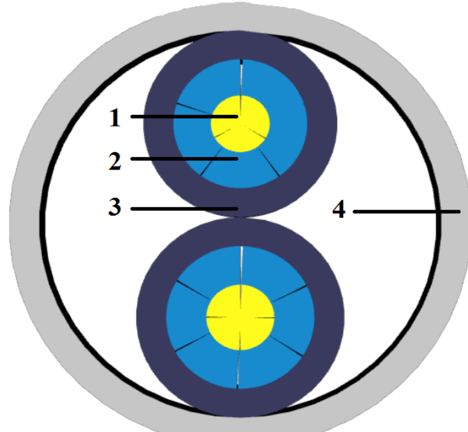

Abbildung 2-7: Aufbau der Recursive Disc Metaphor

Der Aufbau der Metapher, wie Müller und Zeckzer ihn beschreiben, wird in Abbildung 2-7 gezeigt. Alle Attribute einer Klasse bilden eine Kreisfläche, die in gleichgroße Teile zerlegt ist. Attribute (1) werden gelb dargestellt. Um die Attribute werden die Methoden (2) angeordnet, wiederum als blaue Kreisfläche. Jedoch unterscheiden sich diese Flächen in ihrer Größe, da die Größe einer Methode nach der Anzahl ihrer Anweisungen (engl. Statements) ermittelt wird. Dabei kommt auch die Position, genauer ihre Anordnung, zum Tragen. Die größte Methode wird nach oben gesetzt, alle weiteren Methoden daneben, entsprechend ihrer Größe, absteigend im Uhrzeigersinn.

Auf der nächsten Ebene befinden sich die Klassen (3), deren Größen durch die Anzahl ihrer Attribute, den Größen ihrer Methoden und inneren Klassen bestimmt werden. Klassen-Glyphen werden durch violette Ringe dargestellt. Wie die Methoden einer Klasse, werden die Klassen eines Packages nach ihrer Größe ebenfalls absteigend im Uhrzeigersinn angeordnet. Gleiches gilt für die Packages (4), die in grau dargestellt werden. Deren Größe und Anordnung bestimmt sich nach den Größen ihrer enthaltenen Klassen und Packages. Die oben beschriebene Anordnung aller Packages, Klassen, Methoden und Attributen stellt die Orientierung des Modells dar. Aufgrund ihrer zirkulären Anordnung und rekursiven Berechnung der Größen wird diese Metapher als *Recursive Disc Metaphor* bezeichnet. (vgl. [Müller/Zeckzer 2015])

Der *Overview*-Bereich erlaubt als Interaktionsmöglichkeit das Auswählens eines Elements, welches dann in rot hervorgehoben wird. Zusätzlich kann das 3D-Modell im Raum gedreht und verschoben werden. Über einen Doppelklick auf ein Element wird dieses zentriert, mithilfe des Mausrades kann hinein- beziehungsweise herausgezoomt werden. Der *Filtering*-Bereich (A) ist hier als Package Explorer umgesetzt, der eine hierarchische Auflistung aller Packages, Klassen, Methoden und Attribute bietet. Die Filterfunktion wird damit erreicht, dass jeder Eintrag eine *Checkbox* enthält, über die einzelne Elemente im *Overview*-Bereich ein- und ausgeblendet werden können. Beim Ausblenden einer Klasse oder eines Packages werden auch alle darin enthaltene Elemente ausgeblendet. Eine weitere Interaktion in diesem Bereich ist das Auswählen eines Elements, wodurch im *Details-on-demand*-Bereich (B) weitere Informationen über das Element angezeigt werden. So werden zum Beispiel für eine Klasse deren Klassenname, in welchem Package sie sich befindet und über wie viele Attribute und Methoden sie verfügt, dargestellt.

3 Theoretische Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen betrachtet. Es wird gezeigt, wo sich der Quellcode befindet, der in der Oberfläche dargestellt werden soll. Da diese Oberfläche als Web-Seite umgesetzt ist, folgt eine Vorstellung der Auszeichnungssprache HTML und der Skriptsprache JavaScript. Abschließend werden noch reguläre Ausdrücke eingeführt.

3.1 Anzeigen des Quellcodes

Um das Ziel */Z10/ Anzeigen des Quellcodes des zugrundeliegenden Softwareprojektes* zu erfüllen, muss dieser vorhanden und von der Oberfläche aus erreichbar sein. Es stellt sich daher die Frage, an welcher Stelle die Quellcodedateien gespeichert sind. Dabei werden in diesem Abschnitt zwei Möglichkeiten dargestellt: Einerseits die Verwendung eines *Repositories* sowie andererseits die lokale Bereitstellung der Ressourcen.

Viele Softwareprojekte greifen auf sogenannte *Repositories* mit einer Versionsverwaltung zurück. So werden *Filehoster* im Web zum Speichern und Verwalten des Quellcodes und weiterer zum Projekt gehörender Dateien eingesetzt. Daher liegt es nahe, dies als Möglichkeit in Betracht zu ziehen, entsprechende Quellcodedateien direkt aus diesen *Repositories* zu laden. Ein Vorteil wäre, dass dabei die Dateien an einem zentralen Ort gespeichert sind und von überall darauf zugegriffen werden kann. Jedoch ergeben sich hiermit verschiedene Probleme, die im Folgenden aufgeführt werden. Gleichzeitig verdeutlichen diese Probleme, warum die Nutzung eines solchen *Repositories* hier nicht von Vorteil ist und daher als Lösung auch nicht weiter betrachtet wird.

Wenn die Quellcodedateien von einem *Filehoster* im Internet verwaltet werden, müssen sie auch von dort geladen werden. Diese Übertragung kostet Zeit und es ist nicht auszuschließen, dass dabei Fehler auftreten. So könnte der *Filehoster* für kurze Zeit nicht erreichbar sein, wodurch es zu Verzögerungen beim Laden der Quellcodedateien und deren Darstellung kommt. Hierbei ist mit einer schlechteren Performance zu rechnen. Das Speichern der Dateien im *Cache* des Browsers bietet nur bedingt eine Verbesserung, da jede benötigte Datei im vornherein mindestens einmal geladen werden muss.

Aufgrund der *Same-Origin-Policy* ist es nicht möglich, mittels JavaScript auf Dateien zuzugreifen, die auf anderen Seiten liegen. Denn die *Origin* der Seite mit den Dateien unterscheidet sich von der *Origin* der Oberfläche. Die *Origin* wird dabei durch das verwendete Protokoll, den Host und den Port eines *Uniform Resource Locator* (kurz URL)

bestimmt (vgl. [Flanagan 2011]). Ein Server müsste diese Dateien zuvor aus dem *Repository* laden und sie dann der Oberfläche zur Verfügung stellen. Jeder der etablierten Anbieter, darunter Google Drive und Dropbox sowie einige auf Softwareprojekte spezialisierte *Filehoster*, wie GitHub, SourceForge oder Bitbucket, hat seine eigenen Zugriffsparameter definiert. Dadurch erschwert sich ein einheitlicher Zugriff auf die Ressourcen.

Das größte Problem bei der Verwendung von *Filehostern* ohne Versionsverwaltung ist, dass sich die Software noch in der Entwicklung befindet. Damit unterliegt der Quellcode noch Änderungen, beispielsweise dem Hinzufügen, Löschen, Umbenennen von Klassen und Methoden. Diese Änderungen können im Zeitraum der Erstellung der Visualisierung bis zum Zeitpunkt der Verwendung vorgenommen werden. Visualisierung und zugehöriger Quellcode würden damit nicht mehr übereinstimmen und die Analyse wäre inkonsistent.

Um diese Probleme zu umgehen, wurde entschieden, dass die Quellcodedateien lokal neben den Dateien der Visualisierung liegen müssen, da diese auch als Grundlage zur Erstellung der Visualisierung dienen. Auf diese Weise kann der Generator diese gleich mit liefern. Der Quellcode wäre hierbei immer verfüg- und von überall erreichbar sowie jederzeit konsistent zur Visualisierung. Dadurch bietet diese Möglichkeit ähnliche Vorteile wie eines der oben genannten *Repositories*. Als Nachteil kann angesehen werden, dass immer der gesamte Quellcode vorliegen muss und nicht, wie bei Verwendung eines *Repositories*, nur betreffende Dateien nachgeladen werden.

Die *Same-Origin-Policy* greift bei Google Chrome auch auf lokaler Ebene. Deshalb können Ressourcen, welche sich in verschiedenen Verzeichnissen befinden, nicht geladen werden. Eine Möglichkeit dies zu umgehen, besteht darin, in den Browser-Konfigurationen die *Same-Origin-Policy* abzuschalten. Allerdings stellt das keine gute Vorgehensweise dar. Zusätzlich ist nicht sichergestellt, dass alle Browser das Abschalten ermöglichen. Es empfiehlt sich daher, über einen Server auf die Oberfläche zuzugreifen, welcher die Ressourcen unter einer *Origin* zusammenfasst.

3.2 Der HTML-Standard

Ohne Standards würde das *World Wide Web* (kurz WWW), wie wir es heute kennen, wohl nicht existieren. Tim Berners-Lee, der Begründer des WWW und Erfinder der *Hypertext Markup Language* (kurz HTML), befürchtete, dass *„die Leute [...] HTTP, HTML und URIs nicht in konsistenter Weise benutzen würden. Dadurch könnten sie unbeabsichtigt Blockaden einführen, die Verknüpfungen wirkungslos machen würden“* [Berners-Lee 1999]. Als die beste Lösung erschien ihm, die Spezifikationen der Protokolle von der *Internet Engineering Task Force* (kurz IETF) standardisieren zu lassen. Da schon zuvor die Aufgabe der IETF darin bestand, die im Internet verwendeten Protokolle wie beispielsweise IP, UDP und TCP zu standardisieren. Später wurde dafür das *World Wide Web Consortium* (kurz W3C) gegründet. Jenes übernahm die Aufgaben des IETF in diesem Bereich, mit dem Zweck *„das volle Potential des Webs auszuschöpfen“* [Berners-Lee 1999].

Der HTML-Standard bietet eine große Menge an *Tags*, die zur semantischen Auszeichnung von Informationen genutzt werden können. Dabei beschreibt er auch deren genaue Verwendung. Ein *Tag* besitzt in der Regel zwei Ausprägungen, einmal als *Start-Tag* und einmal als *End-Tag*. Eine Überschrift wird dabei folgendermaßen ausgezeichnet: *<h1>Überschrift</h1>*. So definiert der Standard auch *Tags*, die zur Auszeichnung und Darstellung von Quellcode auf einer Website verwendet werden sollen. Zwei dieser *Tags*, die für diese Arbeit besondere Bedeutung tragen, sind das *pre-Tag* und das *code-Tag*. Das *pre-Tag* ist wie folgt definiert: *„The pre element represents a block of preformatted text, in which structure is represented by typographic conventions rather than by elements“* [W3C 2014]. Laut dieser Definition eignet sich das *pre-Tag*, um den Quellcode in seiner ursprünglichen Formatierung darzustellen, wodurch dessen Lesbarkeit beibehalten wird. Weiterhin trägt die Formatierung in einigen Programmiersprachen syntaktische Informationen, die durch die Verwendung des *pre-Tags* nicht verloren gehen. Um dabei den Quellcode auch als solchen auszuzeichnen, wird das *code-Tag* verwendet. Zusätzlich kann dieses *Tag* mit einem *class*-Attribut versehen werden, um die verwendete Programmiersprache anzugeben. Der Wert des Attributs sollte dabei mit dem Präfix *„language-“* beginnen.

3.3 JavaScript

„The overwhelming majority of modern websites use JavaScript, and all modern web browsers […] include JavaScript interpreters, making JavaScript the most ubiquitous programming language in history“ [Flanagan 2011]. Diese Aussage beschreibt, welche Bedeutung JavaScript im WWW erlangt hat. Während HTML zur Strukturierung von Inhalten einer Website Verwendung findet, wird JavaScript zur Modellierung des Verhaltens der Website eingesetzt. JavaScript wurde 1995 von Brendan Eich entwickelt und 1997 von ECMA als ECMA-262 standardisiert (vgl. [ECMA 2015]). *ECMA International* ist eine Organisation, die sich Standardisierung von Informations- und Kommunikationssystem zur Aufgabe gemacht hat (vgl. [ECMA 1961]). Der eigentlicher Name von JavaScript lautet daher ECMAScript. Sie ist eine dynamische, typenlose und *C*-ähnliche Sprache.

Der Vorteil in der Verwendung von JavaScript innerhalb einer Website liegt darin, dass deren Inhalte dynamisch verändert und nachgeladen werden können. Durch sie können Daten, beispielsweise in Formularfeldern, auf ihre Korrektheit überprüft werden. Um JavaScript zu verwenden, wird das *script-Tag* benötigt. Innerhalb dessen können JavaScript-Anweisungen geschrieben werden, wie das folgende Beispiel in Quellcode-Listing 3-1 zeigt. Mit HTML wird ein Button definiert und mit einem *onclick*-Ereignis versehen. Durch einen Mausklick auf diesen Button wird die JavaScript-Funktion *zeigeMeldung* aufgerufen. Dabei erscheint auf dem Bildschirm ein kleines Fenster, in dem die Zeichenkette *„Test-123“* ausgegeben wird. Für weitere Informationen zu JavaScript sei hier auf den ECMA-262-Standard [ECMA 2015] verwiesen. Eine umfassendere Einführung bietet Flanagan in *„JavaScript: The Definitive Guide“* [Flanagan 2011].

```
1    <!DOCTYPE html>
2    <html>
3          <body>
4                <button onclick="zeigeMeldung()">Click me</button>
5                <script>
6                      function zeigeMeldung(){
7                            alert("Test-123");
8                      }
9                </script>
10         </body>
11   </html>
```

Quellcode-Listing 3-1: JavaScript in HTML

3.4 Reguläre Ausdrücke

Reguläre Sprachen stammen aus dem Bereich der formalen Sprachen und sind eine Form der algebraischen Beschreibung. Diese wird verwendet, um Zeichenketten durch bestimmte Ausdrücke zu beschreiben. Aufgrund dessen finden sie häufig Einsatz in Systemen, die Zeichenketten verarbeiten müssen. Texteditoren sind solche Systeme. Sie nutzen reguläre Ausdrücke dafür, dem Nutzer eine Möglichkeit zu bieten, gezielt nach Mustern im Text zu suchen (vgl. [Hopcroft et al. 2011]). Dies kann über die *„Suchen und Ersetzen"*-Funktion realisiert sein. Eine weiteres Einsatzgebiet ist die Suche nach Schlüsselwörtern, um diese dann hervorzuheben. Dadurch sind sie bestens für die Verwendung in Syntax-Highlightern geeignet.

In JavaScript wird ein solcher Ausdruck durch die Angabe von Text innerhalb zweier *Slashes* (/) erzeugt. Dabei wird dieser Ausdruck auch *Pattern* genannt, da diese Ausdrücke für *Pattern Matching* (deutsch: Musterabgleich) verwendet werden. Die Zeichenkette zwischen den *Slashes* gibt das Muster an, nach dem gesucht werden soll. In Tabelle 3-1 sind auszugsweise einige Zeichen aufgeführt, die innerhalb eines regulären Ausdrucks eine bestimmte Sonderrolle ausüben.

Zeichen	Bedeutung
*	Das vorangegangene Zeichen darf beliebig oft oder auch 0-mal auftreten.
+	Das vorangegangene Zeichen muss mindestens einmal auftreten.
?	Das vorangegangene Zeichen darf einmal oder gar nicht auftreten.
.	Ein Punkt repräsentiert ein beliebiges Zeichen.
\b	Symbolisiert eine Wortgrenze, also Leerzeichen, Satzzeichen oder Start und Ende einer Zeichenkette.
\w	Symbolisiert ein alphanumerisches Zeichen, einschließlich Unterstrich.
\W	Symbolisiert ein nicht-alphanumerisches Zeichen.

Tabelle 3-1: Sonderzeichen in regulären Ausdrücken (in Anlehnung an [MDN 2015])

Zeichen, die dem zweiten *Slash* folgen, verändern die Bedeutung des *Patterns* (vgl. [Flanagan 2011]). Wird kein Zeichen angegeben, dann wird nur das erste Auftreten des Musters gefunden. Alle Vorkommen des Musters werden durch die Angabe des Zeichens *g* gefunden. Quellcode-Listing 3-2 verdeutlicht dies an ein paar Beispielen und zeigt zugleich eine Möglichkeit der Verwendung von regulären Ausdrücken in JavaScript.

```
1     var text = "Eine HTML-Seite. Zeichenkette HTML wird gefunden." ;
2
3     var pattern1 = /HTML/ ;
4     console.log( text.replace( pattern1, "..." ) ) ;
5     // Ausgabe: Eine ...-Seite. Zeichenkette HTML wird gefunden.
6     var pattern2 = /HTML/g ;
7     console.log( text.replace( pattern2, "..." ) ) ;
8     // Ausgabe:  Eine ...-Seite. Zeichenkette ... wird gefunden.
9
10    text = "Zeichenkette oder nur ein Zeichen?" ;
11    var pattern3 = /\bZeichen\b/g ;
12    console.log( text.replace( pattern6, "..." ) );
13    // Ausgabe: Zeichenkette oder nur ein ...?
```

Quellcode-Listing 3-2: Beispiele für reguläre Ausrücke

Das Thema der regulären Ausdrücke ist wesentlich komplexer. Da im Rahmen dieser Arbeit hierauf nicht weiter eingegangen werden kann, sei für ein allgemeineres Verständnis auf Hopcroft *„Einführung in Automatentheorie, Formale Sprachen und Berechenbarkeit“* [Hopcroft et al. 2011] verwiesen. Weiter Informationen über den Einsatz regulärer Ausdrücke in JavaScript bietet ebenfalls Flanagan in *„JavaScrip: The Definitive Guide“* [Flanagan 2011].

4 Syntax-Highlighter

In diesem Kapitel wird definiert, was Syntax-Highlighting ist und wozu es dient. Weiterhin werden Kriterien vorgestellt, anhand derer verschiedene Frameworks miteinander verglichen werden. Das Framework, dass alle Kriterien am besten erfüllt, wird anschließend noch in Bezug auf die Erweiterbarkeit um neue Sprachen sowie Funktionalitäten genauer betrachtet.

4.1 Syntax-Highlighting

Beim Syntax-Highlighting (deutsch: Syntaxhervorhebung) geht es darum, innerhalb eines Quellcodes bestimmte syntaktische Elemente hervorzuheben. Die Hervorhebung kann durch die Nutzung verschiedener Farben und Schriftarten geschehen. Diese Elemente können Schlüsselwörter oder bestimmte Zeichenkombinationen sein, die speziell für eine Programmiersprache von Bedeutung sind. Zuvor wurden verschiedene Möglichkeiten der Softwarevisualisierung beschrieben, das Syntax-Highlighting stellt dabei eine Form der Visualisierung von Quelltext dar.

Die visuelle Wahrnehmung des Menschen ist gut dafür gerüstet, Farben und vor allem Unterschiede zwischen ihnen zu erkennen. Diese nimmt er sogar bis zu drei mal schneller wahr als Unterschiede zwischen geometrischen Figuren (vgl. [Lohse 1993]). Daher eignet sich das Hervorheben, um die Lesbarkeit des Quellcodes für den Menschen zu vereinfachen. Gleichzeitig ermöglicht dies ebenso eine bessere und schnelle Orientierung innerhalb des Quellcodes. Ein weiterer Vorteil ist, dass auf diese Art fehlerhafte Syntax schneller erkannt wird. Deshalb gehört das Syntax-Highlighting auch zum Standardumfang eines Texteditors im Bereich der Programmierung.

Die Anwendung des Syntax-Highlighting ist dabei nicht nur direkt auf Quellcode beschränkt. Reijers et al. beschreiben in ihrem Artikel *„Syntax highlighting in business process models"* [Reijers et al. 2011] die Anwendung von Syntax-Highlighting im Bereich der Modellierung von Geschäftsprozessen, um positive Effekte zu erzeugen. Dabei formulieren sie zwei Hypothesen, die den Kern der obigen Aussagen treffen und als Motivation für den Einsatz eines Syntax-Highlighters dienen:

*„**H1.** The use of colors to highlight matching operator transitions will have a significant positive impact on understanding accuracy."*
*„**H2.** The use of colors to highlight matching operator transitions will have a significant positive impact on understanding speed."* [Reijers et al. 2011, Hervorhebung im Original]

```
1    package TestSyntax;
2
3    public class SyntaxHighlighting {
4
5        public static void test(float param) {
6           int wert = 12;
7           this.syntaxTest();
8           // Ein Kommentar
9           thiss.eineFunkton("eine Zeichenkette");
10       }
11   }
```
(a)

```
1    package TestSyntax;
2
3    public class SyntaxHighlighting {
4
5        public static void test(float param) {
6           int wert = 12;
7           this.syntaxTest();
8           // Ein Kommentar
9           thiss.eineFunkton("eine Zeichenkette");
10       }
11   }
```
(b)

Abbildung 4-1: Syntax-Highlighting

Die Übertragung der beiden Hypothesen auf das Syntax-Highlighting zeigt die Abbildung 4-1 am Beispiel eines Java-Quellcode-Fragments. In (b) wurden im Vergleich zu (a) alle Schlüsselwörter sowie alle Literale, wie Zahlen und Zeichenketten hervorgehoben. In Zeile 9 ist zu erkennen, dass durch einen Tippfehler das Schlüsselwort *this* nicht hervorgehoben wird. Erkennbar ist eine klare Abgrenzung von syntaktischen Elementen der Sprache durch die Verwendung verschiedener Farben. In diesem Beispiel wurden Schlüsselwörter blau markiert, während Datentypen rot hervorgehoben wurden. Damit grenzen sich diese auch klar gegenüber den Werten der Variablen ab, welche bei Zahlenwerten grün und bei Zeichenketten orange markiert wurden.

4.2 Evaluation eines Syntax-Highlighters

Die Aufgabe, Syntax zu *highlighten*, ist nicht neu. Eine kurze Suche im Web nach Syntax-Highlightern liefert eine Vielzahl von Frameworks, die genau diese Aufgabe erledigen. Aufgrund dessen sowie der Tatsache, dass die Entwicklung eines eigenen Highlighters großen Mehraufwand bedeutet, soll einer der schon bestehenden Highlighter in die Oberfläche eingebaut werden. Wie schon erwähnt, existiert dabei eine große Auswahl. Daher müssen Kriterien erstellt werden, um diese Frameworks zu vergleichen und das für dieses Projekt am besten geeignetste herauszufiltern. Es stellt sich daher die Frage, welche Kriterien für diesen Vergleich herangezogen werden. Ein Kriterium soll die Einhaltung des W3C-Standards bezüglich der Verwendung der *pre-* und *code-Tags* sein. Dieses Kriterium wird dabei als sogenanntes K.O.-Kriterium eingestuft. Ein Framework, dass den Standard nicht einhält, wird von Beginn an als Option für den Einsatz ausgeschlossen.

Weiterhin spielt die Erweiterbarkeit des Frameworks um weitere Funktionalitäten eine wichtige Rolle. Beispielsweise ist es notwendig, spezielle Elemente, wie Schlüsselwörter der Sprache, mit einer ID oder einer besonderen Klasse zu versehen. Mithilfe dieser Angaben können Elemente über JavaScript referenziert und somit auch Interaktionen mit dem Quellcode umgesetzt werden. Da alle Frameworks innerhalb der Oberfläche und damit im Browser laufen müssen, kommen nur jene in Frage, die in JavaScript geschrieben wurden. Von Vorteil wäre dabei, wenn das Framework zur Erweiterung eine Plugin-Struktur bieten würde. So müsste der originale Quellcode nicht modifiziert werden und kann problemlos durch neuere Versionen ausgetauscht werden.

Auch die Erweiterbarkeit des Highlighters um weitere Sprachen soll unterstützt werden, um die Softwarevisualisierung nicht nur auf in Java geschriebene Softwareprojekte zu beschränken. Daraus ergeben sich zwei weitere Kriterien: Zum einen soll es positiv bewertet werden, wenn das Framework bereits mehrere Sprachen unterstützt. Zum anderen existieren Sprachen, die oft in andere Sprachen eingebettet werden. Beispielsweise findet sich innerhalb von HTML-Seiten in vielen Fällen JavaScript-Code, welcher dann ebenfalls korrekt hervorgehoben werden sollte. Die Lizenz, unter der das Framework steht, muss die freie und kostenlose Verwendung erlauben.

Als letztes Kriterium wird die Aktualität, gemessen am Zeitpunkt des letzten Updates, eingehen. Ein seit längerem nicht mehr aktualisiertes Framework muss nicht zwingend schlechter sein. Jedoch ist die Wahrscheinlichkeit geringer, dass das Framework

bei größeren Fehlern oder Änderungen im Standard noch aktualisiert wird. Tabelle 4-1 zeigt eine Auflistung aller Kriterien und bewertet die vier Frameworks *Prism* [PRISM 2015], *highlight.js* [HIGHLIGH 2015], *Rainbow* [RAINBOW 2013] und *SyntaxHiglighterJavaScript* [SHJS 2008]. Weitere Frameworks wie *SyntaxHighlighter* [SH 2011], *beautyOfCode* [BOC 2014], *Sunlight* [SL 2013] oder *DlHighlight* [DH 2015] werden nicht weiter untersucht, da diese den W3C-Standard nicht einhalten oder im Falle von *DlHighlight* zu wenige Sprachen unterstützen.

Kriterium	Prism	highlight.js	Rainbow	SHJS
W3C-Standard	ja	ja	eigenes language-Attribut	nur pre-Tag, ohne code-Tag
Erweiterbarkeit	Sprachen Plugins	nur Sprachen	nur Sprachen	nur Sprachen
Sprachunterstützung	94	135	19	39
eingebettete Sprachen	ja	ja	ja	nein
Lizenz	frei verfügbar (MIT)	frei verfügbar (BSD)	frei verfügbar (Apache)	frei verfügbar (GNU v.3)
Aktualität	2015	2015	2013	2008

Tabelle 4-1: Vergleich der Frameworks

Aus Tabelle 4-1 wird ersichtlich, dass *Prism* und *highlight.js* die Kriterien am besten erfüllen. Für die weitere Verwendung wurde *Prism* ausgewählt, weil es noch zusätzlich durch Plugins erweiterbar ist.

4.3 Prism

Entwickelt wurde *Prism* von Lea Verou, einem Mitglied der *CSS Working Group* des W3C [W3C 2015]. *Prism* steht unter der MIT-Lizenz, die es ermöglicht, *Prism* uneingeschränkt zu verwenden, Änderungen vorzunehmen und weiter zu verbreiten. Da das Projekt *Prism* einsetzt, wird im Folgendem dessen Funktionsweise und genaue Verwendung erklärt. Dazu benötigt werden lediglich die Skript-Datei und ein *Stylesheet*, das beschreibt, wie der Quellcode hervorgehoben werden soll.

Die genannten Dateien müssen in die Oberfläche eingebunden werden, so wie im Quellcode-Listing 4-1 in den Zeilen 4 und 5 gezeigt. Als nächstes muss der Quellcode eingefügt werden. Dies geschieht, wie im Abschnitt HTML-Standard beschrieben, über ein *pre-Tag* und ein *code-Tag*. Letzteres enthält den Quellcode, zu sehen ab Zeile 8. Für das einfachste Beispiel ist das ausreichend. *Prism* nimmt anschließend die Hervorhebungen beim Laden der Seite für alle *pre-* und *code-Blöcke* automatisch vor.

```
1     <!DOCTYPE html>
2     <html>
3         <head>
4             <link rel="stylesheet" href="prism.css">
5             <script src="prism.js"></script>
6         </head>
7         <body>
8             <pre class="line-numbers" >
9                 <code class="language-java">
10               public void setValue(int val){
11                   if (eineMethode(val)) {
12                       this.val = val;
13                   }
14                   System.out.println("setValue aufgerufen");
15               }
16                </code>
17            </pre>
18        </body>
19    </html>
```

Quellcode-Listing 4-1: Einbindung von Prism

Im *code-Tag* wird das Attribut *class* mit dem Wert „*language-java*“ in Zeile 9 ergänzt. Als Code-Fragment wurde ein kurzes Beispiel in Java übergeben. Das Ergebnis nach dem Laden der Seite ist in Abbildung 4-2 dargestellt.

```
1  public void setValue(int val){
2      if (eineMethode(val)) {
3          this.val = val;
4      }
5      System.out.println("setValue aufgerufen");
6  }
```

Abbildung 4-2: Mit Prism hervorgehobener Quellcode

Für Java typische Schlüsselwörter werden in blau dargestellt. Die Methoden in rot und die Zeichenkette in grün. Durch ein Plugin für *Prism* werden zusätzlich links die Zeilennummern angezeigt. Weitere Plugins können auf der *Prism*-Webseite [PRISM 2015] ausgewählt werden. Die hervorgehobenen Bereiche werden durch *Prism* mit *span-Tags* umgeben. Je nach Typ bekommen diese eine Klasse zugeordnet, welche wiederum mittels des *Stylesheets* eingefärbt werden, wie in Abbildung 4-3 ersichtlich ist.

```
<!DOCTYPE html>
▼<html>
  ▶<head>…</head>
  ▼<body>
    ▼<pre class="line-numbers language-java">
      ▼<code class=" language-java">
          <span class="token keyword">public</span>
          <span class="token keyword">void</span>
        ▼<span class="token function">
            "setValue"
            <span class="token punctuation">(</span>
          </span>
          <span class="token keyword">int</span>
          " val"
          <span class="token punctuation">)</span>
          <span class="token punctuation">{</span>
           .
           .
           .
```

Abbildung 4-3: Ausgabe von Prism in HTML

Wie zuvor erwähnt, können Sprachen in andere eingebettet werden. *Prism* erkennt dies und hebt auch die eingebettete Sprache korrekt hervor. Für den Fall, dass eine Sprache noch nicht unterstützt wird, kann *Prism* leicht um neue Sprachen ergänzt werden. Das Ergänzen einer Sprache wird im folgenden Beispiel an der fiktiven Sprache *BeispielMarkup* erklärt. Diese Sprache verwendet nur einfache *Tags*, die jeweils als öffnender *|xyz|* und als schließender *|/xyz|* *Tag* eingesetzt werden. Weiterhin existiert nur eine Form des Kommentars: */* Kommentartext */*. Um diese Sprache zu ergänzen, sind nur die sechs Zeilen aus dem Quellcode-Listing 4-2 notwendig.

```
1      <script>
2          Prism.language.BeispielMarkup = {
3                  'comment': {pattern: /\/\*[\w\W]*?\*\//},
4                  'tag': {pattern: /\|\w*\||\|\/\w*\|/g}
5          }
6      </script>
```

Quellcode-Listing 4-2: Prism und BeispielMarkup

Für das oben Dargestellte ist lediglich ein gutes Verständnis für reguläre Ausdrücke nötig. Die beiden Angaben *comment* und *tag* werden als Wert für das *class*-Attribut im *span-Tag* benutzt. Daher müssen beide noch in ein *Stylesheet* eingetragen werden, wie in Quellcode-Listing 4-3 gezeigt.

```
1     .token.tag {
2         /* violett */
3         color: #9932cc;
4     }
5
6     .token.comment {
7         /* gruen */
8         color: #228b22;
9     }
```

Quellcode-Listing 4-3: Style für BeispielMarkup

Die neue Sprache kann nun von *Prism* auf die gleiche Weise hervorgehoben werden wie im obigen Beispiel mit Java. Es ist auch möglich, bestehende Sprachen zu erweitern oder aus bestehenden neue aufzubauen. Dazu wird die Funktion *extend* verwendet, wie im Quellcode-Listing 4-4 zu sehen ist.

```
 1    <script>
 2    Prism.languages.BeispielMarkup = Prism.languages.extend('markup', {
 3          'comment': {
 4                pattern: /\/\*[\w\W]*?\*\//
 5          },
 6          'tag': {
 7                pattern: /\|\w*\||\|\/\w*\|/g
 8          }
 9    });
10    </script>
```

Quellcode-Listing 4-4: Funktion extend für BeispielMarkup

Der erste Parameter steht hierbei für die Sprache, die als Ausgangssprache verwendet wird. Der zweite Parameter übergibt die zusätzlichen Werte. *Prism* bietet noch weitere Möglichkeiten an, um auch kompliziertere Ausdrücke zu markieren. Zusätzlich erlaubt es Rekursionen. Da bisher jedoch nur Java-Projekte vom Generator unterstützt werden und *Prism* die Sprache Java schon beherrscht, soll hier nicht weiter auf die Erstellung einer neuen Sprache eingegangen werden.

Eine weitere Möglichkeit, *Prism* zu erweitern, besteht darin, Plugins zu entwickeln. An bestimmten Stellen der Bearbeitung, beispielsweise kurz vor dem Einfärben eines Schlüsselworts, ruft *Prism* alle Funktionen der Plugins auf, die sich für diese Stelle registriert haben. Eine solche Stelle wird *hook* genannt. Die Registrierung erfolgt durch folgenden Ausdruck: *Prism.hooks.add(hookname, callback).* Soll das Plugin Funktionen vor dem Hervorheben eines *pre-Tags* aufrufen, wird *before-highlight* als *hook* verwendet. Dementsprechend wird für Funktionen nach dem Hervorheben *after-highlight* angegeben. Wurde ein Schlüsselwort erkannt, kann mit *wrap* direkt an dieser Stelle eingegriffen werden, wie das Quellcode-Listing 4-5 zeigt.

Dieses Plugin ergänzt die Schlüsselwörter *private*, *public* und *protected* zusätzlich um eine weitere Klasse, welche genutzt werden kann, um diese Schlüsselwörter besonders hervorzuheben.

```
1    <script>
2    Prism.hooks.add('wrap', function(env){
3          if( env.type == "keyword" && (
4                env.content == "private" || env.content == "public" ||
5                env.content == "protected" )
6          ){
7              env.classes.push("accessModifier");
8          }
9    });
10   </script>
```

Quellcode-Listing 4-5: Plugin für Prism

5 Erweiterung des Prototyps

Für die Darstellung des Quellcodes in der Oberfläche wurde diese erweitert, was nachstehend vorgestellt wird. Das zweite Unterkapitel beschreibt die Architektur der Oberfläche und erläutert, wie die Erweiterungen in diese integriert und umgesetzt wurden.

5.1 Funktionsweise

Für das Anzeigen des Quellcodes in der vorgestellten Oberfläche, ist vorerst ein geeigneter Bereich innerhalb dieser zu identifizieren. Unter der Verwendung der *Recursive Disc Metaphor* erfolgt die Visualisierung der Hauptbestandteile von Software, das heißt Packages, Klassen, Methoden und Attribute im *Overview*-Bereich. Dagegen ist für die Darstellung zusätzlicher Informationen, wie beispielsweise Name, Typ und Sichtbarkeit eines einzelnen Elements, der *Details-on-demand*-Bereich geeignet. Dabei kann der Quellcode ebenfalls als zusätzliche Information zu einem Element angesehen werden. Daher bietet sich dieser Bereich für das Anzeigen des Quellcodes an. Abbildung 5-1 verdeutlicht dies an einem Beispiel.

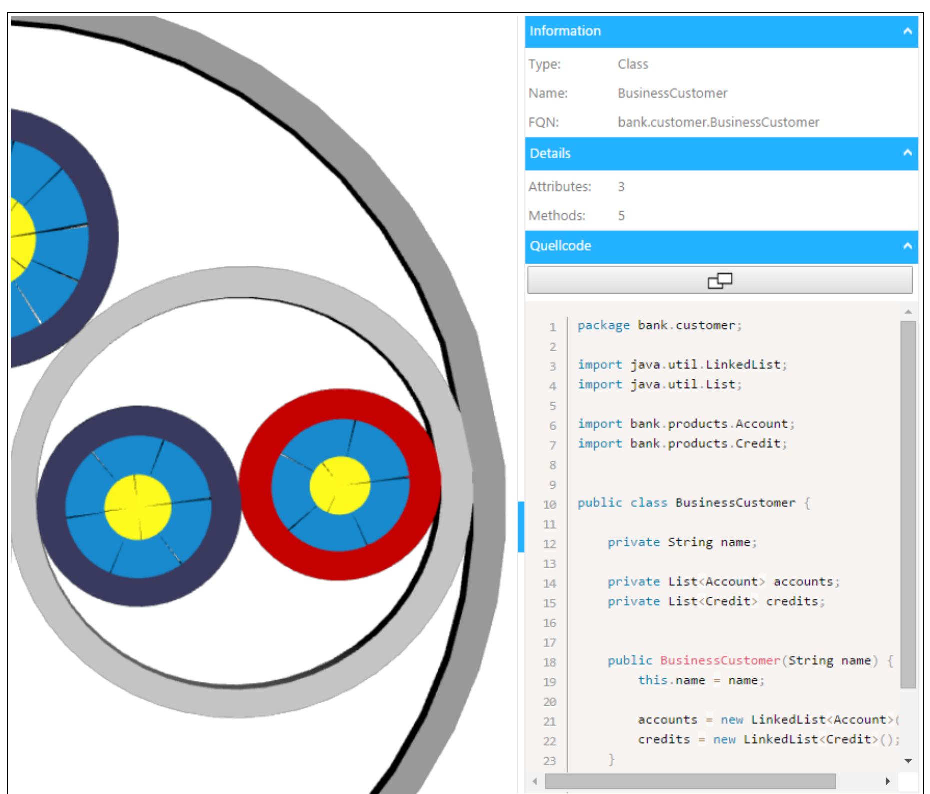

Abbildung 5-1: Darstellung des Quellcodes

Wie an den Scrollbalken obiger Abbildung zu erkennen, passt nicht immer der gesamte Quellcode in den *Detalis-on-demand*-Bereich. Zusätzlich zur Höhe ist dieser Bereich auch in der Breite beschränkt. Aufgrund dessen hat der Benutzer durch den Button über der Anzeige des Quellcodes die Möglichkeit, sich den Quellcode in einem extra Fenster anzuzeigen zu lassen. Das Fenster ist in seiner Größe beliebig skalierbar und kann verschieden positioniert werden, beispielsweise auf einem zweiten Bildschirm.

In der Visualisierung wurde eine Klasse selektiert. Daher wird ihr entsprechender Quellcode auch rechts im *Details-on-demand*-Bereich angezeigt. *Prism* hat diesen schon automatisch mit einem Syntax-Highlighting versehen. Damit sind die Ziele */Z10/* sowie */Z20/* erfüllt. Durch die Selektion eines Attributs erfolgt ebenfalls die Anzeige des gesamten Quellcodes der zugehörigen Klasse. Zusätzlich wird dabei jedes weitere Vorkommen des Attributs im Quellcode besonders hervorgehoben. Ersichtlich ist dies im Beispiel in Abbildung 5-2. Dort wurde das Attribut *name* selektiert. Ebenso verhält es sich bei der Auswahl einer Methode. Einzige Ausnahme bilden Packages. Einem solchen Element kann kein Quellcode direkt zugeordnet werden, deshalb wird hierbei auch nichts angezeigt. Die Selektion eines Elements verhält sich somit wie im Ziel */Z30/* beschrieben.

```
10  public class BusinessCustomer {
11
12      private String name;
13
14      private List<Account> accounts;
15      private List<Credit> credits;
16
17
18      public BusinessCustomer(String name) {
19          this.name = name;
20
21          accounts = new LinkedList<Account>();
22          credits = new LinkedList<Credit>();
23      }
24      public void setName(String name) {
25          this.name = name;
26      }
27
28      public String getName() {
29          return name;
30      }
```

Abbildung 5-2: Hervorhebung eines Attributs

Das Ziel */Z40/* ergänzt die Visualisierung um eine Interaktionsmöglichkeit mit dem Quellcode. Erfolgt ein Klick mit der Maus auf eine Attribut- oder Methodendefinition im Quelltext, wird das entsprechende Element in der Visualisierung selektiert. Diese Art der Selektion verhält sich dabei ebenso wie die direkte Auswahl eines Elements in der Visualisierung. Dadurch werden die gleichen Auswirkungen wie im Ziel */Z30/* hervorgerufen. Abbildung 5-3 zeigt, dass eine Definition besonders hervorgehoben wird, sobald der Benutzer mit der Maus über diese fährt. Damit ist ihm ersichtlich, dass das Element selektiert werden kann.

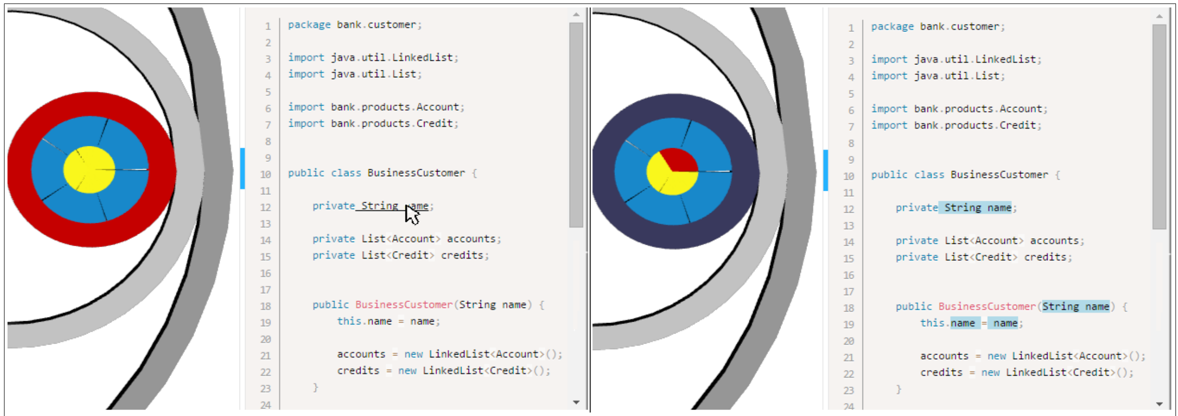

Abbildung 5-3: Elementselektion im Quellcode

5.2 Implementierung

Nach Vorstellung der neuen Funktionen der Oberfläche, folgen nun Erläuterungen zu ihren Implementierungen. Zuerst wird der Quellcode der Oberfläche angepasst. Wie in Kapitel 4.3 beschrieben, müssen die Skript-Dateien von *Prism* geladen sowie das *pre-* und *code-Tag* hinzugefügt werden. Um mit JavaScript auf das *code-Tag* referenzieren zu können, erhält es das Attribut *id* mit dem Wert *„codeTag"*.

5.2.1 Architektur der Oberfläche

Durch die Entwicklung einer neuen Controller-Klasse ist es möglich, die Oberfläche um neue Funktionen zu ergänzen. Alle Controller-Klassen sind dabei mit Plugins vergleichbar. Der Aufbau eines solchen wird im Folgenden am Beispiel des Plugins zur Darstellung des Quellcodes gezeigt.

```
1    var codeController = (function(){
2          function initialize() {
3                subscribeOnEntitySelected(onEntitySelected);
4          }
5
6          function reset() {
7                // fuehre reset aus
8          }
9
10         function onEntitySelected(event, entity) {
11               // reagiere auf das Event
12         }
13
14         return {
15               initialize: initialize,
16               reset: reset
17         };
18   }());
```

Quellcode-Listing 5-1: Plugin zum Anzeigen des Quellcodes

Das Quellcode-Listing 5-1 wird in der Datei *CodeController.js* im Ordner *scripts* gespeichert und auf die gleiche Weise wie *Prism* in die Oberfläche geladen. In Zeile 1 wird die globale Variable *codeController* erstellt und ihr dabei eine spezielle Form einer Funktion zugewiesen, ein sogenanntes *Closure*. Wie viele andere Sprachen nutzt JavaScript *lexical scoping* (vgl. [Flanagan 2011]). Dadurch ermöglicht sich das Folgende: Im Körper der Funktion ab Zeile 1 können weitere Funktionen und Variablen definiert werden. Diese haben die besondere Eigenschaft, dass sie nur innerhalb der umfassenden Funktion verfügbar sind. Die Funktionen *initialize*, *reset* und *onEntitySelected* werden daher auch *inner functions* genannt.

Die *inner function initialize* kann aufgrund des *lexical scoping* auf die beiden anderen Funktionen zugreifen, da sie ebenfalls innerhalb der äußeren Funktion definiert wurde. Damit ist es auch möglich, Aspekte der objektorientierten Programmierung umzusetzen. Die äußere Funktion ist somit vergleichbar mit einer Klasse, die über weitere Funktionen, den *inner functions*, verfügt. Variablen innerhalb dieser Klasse sind damit in allen *inner functions* sichtbar, nicht aber außerhalb der Klasse. Sollen Funktionen oder Variablen auch nach außen sichtbar sein, so müssen sie mit *return* (siehe Zeile 14) zurückgegeben werden. Bei *codeController* handelt es sich damit um ein Objekt, auf dessen Funktionen mithilfe des Punktoperators zugegriffen werden kann. *codeController.initialize();* führt die Funktion *initialize* aus, *codeController.onEntitySelected()* liefert dagegen einen Fehler, weil diese Funktion nach außen nicht sichtbar ist.

Eine weitere Besonderheit ist das Eventsystem. Wird beispielsweise im Package Explorer auf ein Element geklickt, löst das ein Event aus. Alle Controller-Klassen, die auf dieses Event reagieren sollen, müssen sich zuvor mit einer Funktion dafür registrieren. Durch die Funktion *subscribeOnEntitySelected(onEntitySelected)* in Zeile 3 registriert sich der *Code-Controller* für das Event *onEntitySelected* mit der gleichnamigen Funktion. Das ist der Ansatzpunkt für das Laden des Quellcodes, da der Funktion *onEntitySelected(event, entity)* als Parameter das selektierte Element mit übergeben wird. Nimmt diese Controller-Klasse Änderungen an einem Element vor oder selektiert ein anderes, muss das allen anderen Controller-Klassen mitgeteilt werden. Eine einfache Möglichkeit dabei ist, über die Funktion *publishOnEntitySelected(id)* selbst ein Event auszulösen. Dabei wird die ID des Elements übergeben und alle anderen Controller-Klassen können ihrerseits wieder darauf reagieren.

5.2.2 Laden und Anzeigen des Quellcodes

Zur Erfüllung des Ziels */Z10/ Anzeigen des Quellcodes des zugrundeliegenden Softwareprojektes* wird die Funktion *onEntitySelected(event, entity)* erweitert. Der zweite Parameter steht dabei für das selektierte Element, welches als Objekt mit weiteren Attributen repräsentiert wird. Das Attribut *type* kann die vier Werte *„FAMIX.Namespace“*, *„FAMIX.Class“*, *„FAMIX.Method“* und *„FAMIX.Attribute“* annehmen. *„FAMIX.Namespace“* steht dabei für ein Package. Ein weiteres Attribut ist *qualifiedName*, dessen Wert vom Attribut *type* abhängt. *qualifiedName* entspricht der Struktur, wie sie im Package Explorer abgebildet ist. Besitzt dieses Attribut den Wert *„bank.customer.PrivateCustomer.credits“*, lässt sich ableiten, dass es sich hierbei um das Attribut *credits* handelt. Dieses ist in der Klasse *PrivateCustomer* definiert. Diese befindet sich innerhalb des Packages *customer*, welches selbst wiederum innerhalb des Packages *bank* eingebunden ist. Dieser Aufbau entspricht der Struktur der Verzeichnisse, in denen die Quellcodedateien liegen. Verzeichnisse entsprechen dabei den Packages. Eine Quellcodedatei in dieser Ordnerstruktur entspricht einer Klasse, die in Java den selben Namen trägt. Aufgrund dessen können *type* und *qualifiedName* dazu verwendet werden, die Quellcodedatei zu ermitteln, in der das selektierte Element definiert ist.

```
1    function onEntitySelected(event, entity) {
2          var javaCodeFile = null;
3          if (entity.type === "FAMIX.Class"){
4                javaCodeFile = entity.qualifiedName.replace(/\./g, "/")
5                                  + ".java";
6          } else if (entity.type === "FAMIX.Method" ||
7                     entity.type === "FAMIX.Attribute"){
8                var fileSplit = entity.qualifiedName.split(".");
9                fileSplit.pop();
10               javaCodeFile = fileSplit.join('/') + ".java";
11         } else{
12               // Namespace / Package
13               codeController.reset();
14         }
15         displayCode(javaCodeFile, entity);
16   }
```

Quellcode-Listing 5-2: onEntitySelected

Handelt es sich bei dem selektierten Element um eine Klasse, so müssen nur alle Punkte im Attribut *qualifiedName* durch Slashes ersetzt und am Ende durch *„.java“* ergänzt werden. Quellcode-Listing 5-2 zeigt dies in den Zeilen 3 bis 5. Handelt es sich um eine Methode oder ein Attribut, muss vor Ergänzung des Dateityps *„.java“* noch der Name der Methode oder des Attributs abgeschnitten werden, wie in den Zeilen 8 und 9 gezeigt. Die resultierende Zeichenkette beinhaltet nun den Pfad zur Quellcodedatei. Da bei einem Package nichts angezeigt werden soll, wird mit *codeController.reset()* in Zeile 13 die Anzeige gelöscht. Der Dateiname steht nun im Attribut *javaCodeFile* und wird an die Funktion *displayCode,* siehe Quellcode-Listing 5-3, übergeben.

In dieser Funktion wird in Zeile 3 ein *XMLHttpRequest-Objekt* erzeugt, welches das Laden der Quellcodedatei übernimmt. An dieser Stelle kann das schon beschriebene Problem der *Same Origin Policy* auftreten, weshalb alle Dateien auf einem Server liegen müssen. Weitere Probleme könnten unter anderem das Fehlen von Dateien sein. Wegen der Übersichtlichkeit wird hier allerdings auf die Darstellung der Fehlerbehandlung im Quellcode-Listing verzichtet. Sobald eine Datei vollständig geladen ist, wird ihr Inhalt, sichtbar in Zeile 6, dem *code-Tag* angefügt. Dieses wurde zuvor in Zeile 5 über dessen ID *codeTag* ermittelt. Anschließend übernimmt *Prism* das Syntax-Highlighting. Damit sind die Ziele */Z10/* sowie */Z20/* erfüllt. Für die folgenden Ziele */Z30/* und */Z40/* erfolgt ab Zeile 8 der Aufruf zusätzlicher Funktionen.

```
 1    function displayCode(file, entity){
 2          var xmlhttp=new XMLHttpRequest();
 3          xmlhttp.open("GET",'data/model/src/'+file, false);
 4          xmlhttp.send();
 5          var codeTag = document.getElementById("codeTag");
 6          codeTag.innerHTML = xmlhttp.responseText;
 7          Prism.highlightElement(codeTag, false, function(){
 8                textNodesToSpan();
 9                highlightSelectedElement(entity);
10                addInteraction(entity);
11          });
12    }
```

Quellcode-Listing 5-3: displayCode

5.2.3 Hervorheben des selektierten Elements

Prism umschließt alle Schlüsselwörter mit einem *span-Tag* und übergibt diesen jeweils eine Klasse. In Abhängigkeit der Klasse wird ihnen eine Farbe zugewiesen. Das Ziel */Z30/* beschreibt zusätzlich die Aufgabe, alle Vorkommen des selektierten Elements im Quellcode besonders hervorzuheben. Allerdings stellen beispielsweise die Namen der Attribute keine solchen Schlüsselwörter dar. Sie erhalten daher auch kein umschließendes *span-Tag*. Um diese dennoch mit einem bestimmten Style versehen zu können, ist es notwendig, die Ausgabe von *Prism* anzupassen. Diese Aufgabe übernimmt die Funktion *textNodesToSpan,* dargestellt in Quellcode-Listing 5-4.

```
 1    function textNodesToSpan(){
 2          var codeTag = document.getElementById("codeTag");
 3          var codeTagChilds = codeTag.childNodes;
 4          for (var i=0; i<codeTagChilds.length; i++){
 5                if (codeTagChilds[i].nodeName == "#text"){
 6                      var span = document.createElement("span");
 7                      var textNode = document.createTextNode(
 8                                        codeTagChilds[i].textContent);
 9                      span.appendChild(textNode);
10                      codeTag.replaceChild(span, codeTagChilds[i]);
11                }
12          }
13    }
```

Quellcode-Listing 5-4: textNodesToSpan

In Zeile 3 werden alle enthaltenen Elemente des *code-Tags* in der Variablen *codeTagChilds* gespeichert. Im Anschluss wird über alle Elemente iteriert. Das Attribut *nodeName* in Zeile 5 gibt an, um welchen *Tag* es sich dabei handelt. Hier können nur die beiden Fälle *SPAN* und *#text* auftreten. Ist das betreffende Element ein Textknoten, generiert die Funktion ab Zeile 6 ein neues *span*-Element und hängt diesem den Inhalt des Textknoten an. Zeile 10 ersetzt nun das alte Element durch das neue.

Danach wird das selektierte Element hervorgehoben. Diese Aufgabe übernimmt die Funktion *highlightSelectedElement.* Wurde ein Attribut selektiert, werden alle Elemente des *code-Tags* durchlaufen und deren Inhalt mit dem Namen des Attributs verglichen. Ein regulärer Ausdruck überprüft, ob es sich dabei um ein einzelnes Wort handelt. Dadurch werden alle Vorkommen ausgeschlossen, in denen der Name des Attributs nur Teil einer anderen Zeichenkette ist. Quellcode-Listing 5-5 zeigt diesen Ausdruck in Zeile 5. Weil der Ausdruck *\b* bereits im Kapitel 3.4 beschrieben wurde, soll hier nicht näher darauf eingegangen werden. Liegt jedoch eine Übereinstimmung vor, wird das

umschließende *span-Tag* in Zeile 8 um die Klasse *codeControllerHighlightAttribute* erweitert.

```
 1    function highlightSelectedElement(entity){
 2       if (entity.type === "FAMIX.Attribute"){
 3          var codeTag = document.getElementById("codeTag");
 4          var codeTagChilds = codeTag.childNodes;
 5          var reg = new RegExp('\\b' + entity.name + '\\b', 'g');
 6          for(var i=0; i < codeTagChilds.length; i++){
 7                if(codeTagChilds[i].textContent.search(reg) >= 0){
 8                      codeTagChilds[i].className +=
 9                            "codeControllerHighlightAttribute";
10                }
11          }
12       } else if (entity.type === "FAMIX.Method"){
13          [...]
14       }
15    }
```

Quellcode-Listing 5-5: highlighSelectedElement

Das Vorgehen ist für Methoden ähnlich. Das *span*-Element erhält dabei allerdings die Klasse *codeControllerHighlightMethod*. Durch die Angabe unterschiedlicher Klassen für Attribute und Methoden ist es möglich, diese im Quellcode unterschiedlich hervorzuheben.

5.2.4 Interaktion mit dem Quellcode

Die zuvor beschriebene Interaktion wird durch eine Selektion in der Visualisierung ausgelöst. Eine weitere Interaktion soll dem Benutzer innerhalb des Quellcodes zur Verfügung stehen. Daher wird in diesem Abschnitt die Umsetzung des Ziels */Z40/* näher betrachtet. Im Quellcode kann der Benutzer auf die Definition einer Methode oder eines Attributs klicken und dabei dieses Element in der Visualisierung selektieren. Daraufhin erfolgt die Anzeige genauerer Informationen zu diesem Element im *Details-on-demand*-Bereich.

Um dieses Verhalten umzusetzen, müssen alle Attribute sowie Methoden des Quellcodes den entsprechenden Elementen in der Visualisierung zugeordnet werden. Hierbei ist es notwendig alle Elemente der Visualisierung aufzulisten, die zur gerade angezeigten Klasse gehören. Dazu wird die ID dieser Klasse benötigt. Diese kann über das selektierte Element ermittelt werden, bei Klassen entspricht es dem Attribut *id*, bei Attributen und Methoden dem Attribut *belongsTo*. Nun können innerhalb einer Schleife alle Elemente der Visualisierung überprüft werden, ob ihr Attribut *belongsTo* mit dieser ID übereinstimmt. Ist das der Fall, wird das Element abhängig vom *type* in die Liste der Methoden beziehungsweise in die Liste der Attribute eingefügt.

Der Quellcode selbst muss ebenfalls darauf untersucht werden, welche Methoden- und Attributdefinitionen in ihm auftreten. Attributdefinitionen finden sich dabei in Java wie folgt: Eingeleitet werden sie mit einem Zugriffsmodifikator (engl. *access modifier)* wie *public*, *private* oder *protected.* Darauf folgt ein Datentyp und der Name des Attributs beziehungsweise eine Aufzählung verschiedener Attribute zum gleichen Datentyp. Abgeschlossen wird die Definition durch ein Semikolon. Bei Methoden muss zusätzlich darauf geachtet werden, dass nur abstrakte Methoden auf ein Semikolon enden, alle anderen jedoch mit einer öffnenden, geschweiften Klammer.

Das nachstehende Quellcode-Listing 5-6 zeigt, wie diese Struktur verwendet werden kann, um Methoden und Attribute im Quellcode zu identifizieren. Dabei werden in Zeile 2 alle Elemente mit der Klasse *accessModifier* aufgelistet. Diese Klasse wird standardmäßig nicht von *Prism* vergeben, deshalb wurde es um das Plugin aus Quellcode-Listing 4-5 erweitert. Alle Elemente werden in der Schleife ab Zeile 3 überprüft und zusammen mit ihren direkten Nachfolgern zu einem Array zusammengefasst. Die Bedingung der Schleife in Zeile 7 und 8 stoppt das Hinzufügen, sobald ein Semikolon oder eine öffnende, geschweifte Klammer auftritt. Enthält eines dieser Elemente die Klasse *function*, erfolgt die Zuordnung des Arrays zur Liste der Methoden in Zeile 16. *Prism* verwendet die Klasse *function* zur Kennzeichnung von Methodennamen. Tritt diese Klasse nicht auf, wird das Array der Liste der Attribute zugeordnet.

```
 1    function addInteraction(classId){
 2          var accessModifier = getElementsWithClass("accessModifier")
 3          for(var line=0; line < accessModifier.length; line++){
 4                var element = accessModifier[line];
 5                var tmpField = [element];
 6                var isMethod = false;
 7                while( ! (element.nextSibling.textContent == "{"
 8                       || element.nextSibling.textContent == ";" ) ){
 9                      if (element.className.includes("function")){
10                            isMethod = true;
11                      }
12                      element = element.nextSibling;
13                      tmpField.push(element);
14                }
15                if( isMethod ){
16                      methods.push(tmpField);
17                } else{
18                      attributes.push(tmpField);
19                }
20          }
21    }
```

Quellcode-Listing 5-6: addInteraction

Im letzten Schritt werden die im Quellcode gefunden Methoden beziehungsweise Attribute mit den korrekten IDs der Methoden und Attribute der Visualisierung versehen. Dazu werden die jeweiligen Listen miteinander abgeglichen. Zu beachten ist, dass Methodennamen nicht einzigartig sein müssen. Durch Methodenüberladung können zwei oder mehr Methoden den gleichen Namen tragen und sich lediglich in den Datentypen der Parameter und deren Anzahl unterscheiden. Die Visualisierung hält dabei für Methoden das Attribut *signature* bereit. Dieses umfasst den Namen der Methode und eine Liste der Datentypen ihrer Parameter, jedoch nicht die verwendeten Parameternamen. Da diese aber im Quellcode vorhanden sind, ist kein direkter Vergleich möglich. Durch den Einsatz von regulären Ausdrücken, können die Methoden dennoch korrekt zugeordnet werden. Die Ersetzung von Sonderzeichen wie Kommas, Klammern und Leerzeichen im Attribut *signature* durch den Ausdruck .* erzeugt dafür passende Ausdrücke. Der Punkt symbolisiert alle beliebigen Zeichen und der Stern eine beliebig große Anzahl. Dieser Ausdruck wird nun verwendet, um die Definition im Quellcode zu finden. Das allein löst das Problem der überladenen Methoden aber noch nicht ganz, wie das Beispiel in Tabelle 5-1 demonstriert.

Signature	regulärer Ausdruck
int testFunction(int,float)	/int.*testFunction.*int .*float .*/
int testFunction(int,float,float)	/int.*testFunction.*int .*float .*float .*/

Tabelle 5-1: Reguläre Ausdrücke aus der Signatur einer Methode

In Spalte 1 sind die Signaturen zweier Methoden mit dem gleichen Namen angegeben. Dazu wurden in Spalte 2 die regulären Ausdrücke erzeugt. Allerdings wird die Methode *int testFunction (int anzahl, float zahl, float basis)* von beiden Ausdrücken erkannt. Denn der erste Ausdruck erwartet nach dem Datentyp *float* beliebig viele Zeichen, was bei dieser Methode auch der Fall ist. Die Methode *int testFunction (int anzahl, float zahl)* wird dagegen nur vom ersten Ausdruck erkannt, nicht aber vom Zweiten. Das Problem löst sich erst dadurch, wenn die Ausdrücke nach ihrer Länge sortiert werden. Die Länge wird hierbei durch die Anzahl des Symbols .* in ihnen beschrieben. Dies geschieht in der Funktion *addInteractionForMethods,* zu sehen im Quellcode-Listing 5-7 in den Zeilen 2 und 3. In den beiden Schleifen ab Zeile 5 werden alle Ausdrücke mit allen Methoden aus dem Quelltext verglichen. Zuerst wird der längste Ausdruck überprüft. Wenn keine weiteren Übereinstimmung gefunden werden, wird der nächstkleinere Ausdruck zum Vergleich herangezogen.

Bei Übereinstimmung wird in Zeile 11 die ID der Methode aus der Visualisierung an das *span-Tag* der Methode im Quellcode angefügt. Zusätzlich wird eine weitere Klasse vergeben, um beispielsweise das Element beim Überfahren mit der Maus hervorzuheben. Gleichfalls wird eine *onclick*-Funktion definiert, welche die schon beschriebene Funktion *publishOnEntitySelected* in Zeile 14 mit der vergebenen ID aufruft. Das Eventsystem der Oberfläche reagiert darauf und teilt dies allen anderen Controller-Klassen mit. Für Attribute ist das Vorgehen ähnlich. Weil es dort jedoch keine Überladungen gibt, ist der Einsatz regulärer Ausdrücke nicht notwendig.

```
 1    function addInteractionForMethods(classMethods, methods){
 2        var patterns = createPatternsFromSignature();
 3        patterns = patternsSortByLength();
 4
 5        for(var i=0; i<patterns.length; i++){
 6            for(var line=0; line < methods.length; line++){
 7                var quelltextZeile = methods[line];
 8                if( quelltextZeile.match(patterns[i])  ){
 9                    var method = quelltextZeile.getSpanTag();
10                      if (method.id == null){
11                          method.id = classMethods[patterns[i]].id;
12                          method.className += " codeControllerHover";
13                          method.onclick = function(){
14                              publishOnEntitySelected(this.id);
15                          }
16                      }
17                  }
18              }
19          }
20      }
```

Quellcode-Listing 5-7: addInteractionForMethods

5.2.5 Darstellung im Extra-Fenster

Nachdem die vorhergegangenen Schritte implementiert wurden, wird der Quellcode mit allen Hervorhebungen und Interaktionsmöglichkeiten im unteren Abschnitt des *Details-on-demand*-Bereichs der Oberfläche dargestellt. Dieser Bereich ist aber relativ klein, weshalb der Benutzer viel scrollen muss, um die für ihn interessanten Stellen sehen zu können. Um die Benutzerfreundlichkeit zu erhöhen, kann der Quellcode auch in einem Extra-Fenster angezeigt werden. Die Anzeige des Quellcodes enthält dazu einen Button, der die Funktion *openWindow* im *CodeController* aufruft. Diese öffnet über *window.open(url)* ein neues Fenster und lädt eine entsprechende Seite, die ebenfalls ein *code-Tag* enthält. Zusätzlich wird eine Referenz auf die Seite gehalten, wodurch der Zugriff auf deren *code-Tag* möglich ist. Beim Laden einer Quellcodedatei übergibt der *CodeController* der neuen Seite den Inhalt der Datei.

Damit die Selektion eines Elements in diesem Fenster auch das entsprechende Element in der Visualisierung im Hauptfenster hervorhebt, benötigt das Extra-Fenster einen Bezug zur Oberfläche des Hauptfensters. Der Bezug wird durch das Objekt *opener* hergestellt, über welches das Extra-Fenster standardmäßig verfügt. Das Objekt referenziert die aufrufende Seite und repräsentiert damit die gesamte Oberfläche. Durch die Selektion eines Elements wird die Funktion *publishOnEntitySelected* aufgerufen. Da allerdings das Eventsystem in diesem Fenster nicht vorhanden ist, muss der Aufruf an das Hauptfenster weitergeleitet werden. Dies übernimmt die im Quellcode-Listing 5-8 ersichtliche Funktion. Über das Objekt *opener* erfolgt der Aufruf der eigentlichen Funktion im Hauptfenster.

```
1     <script>
2           function publishOnEntitySelected(entity){
3                 opener.publishOnEntitySelected(entity);
4           }
5     </script>
```

Quellcode-Listing 5-8: Mapperfunktion im Extra-Fenster

6 Fazit und Ausblick

Softwarevisualisierung bietet verschiedene Möglichkeiten des Informationsgewinns über eine Software. Der von Müller entwickelte Prototyp erstellt drei dimensionale Visualisierungen. Als Grundlage dienen ihm die Quellcodedateien der Software. Jedoch wird der Quellcode selbst nicht in seiner ursprünglichen Form dargestellt, wodurch es schwieriger ist, die Elemente aus der Visualisierung direkt mit ihrem zugehörigen Quellcode zu vergleichen. Mit der vorliegenden Arbeit wurde diese Schwierigkeit beseitigt. Dazu wurden zunächst vier Teilziele definiert.

Neben dem Laden und der einfachen Darstellung des Quellcodes, sollte einem Benutzer auch die Interaktion mit diesem ermöglicht werden. Der erste Schritt dabei war, die Lesbarkeit des Quellcodes und damit die Orientierung in ihm zu verbessern. Diese Aufgabe übernimmt ein Framework, das den Quellcode mit einem Syntax-Highlighting versieht. Die Evaluation verschiedener Frameworks ergab, dass dazu das Framework *Prism* am besten geeignet ist. Die Darstellung weiterer Informationen eines Elements erfolgt beispielsweise durch dessen Selektion in der Visualisierung.

Die Einführung dieser Art der Interaktion mit dem Quellcode wurde im zweiten Schritt durch die Entwicklung eines Plugins für *Prism* sowie einer weiteren Controller-Klasse für die Oberfläche umgesetzt. Damit kann der Benutzer mit der Maus auf eine Definition einer Methode oder eines Attributs im Quellcode klicken. Dieses Ereignis verhält sich dabei exakt wie eine Selektion in der Visualisierung. Der direkte Bezug zwischen der Visualisierung und dem zugrundeliegenden Quellcode, der von einem eher abstrakten Charakter geprägt ist, wird nun ersichtlicher.

Wenngleich alle genannten Ziele erfüllt wurden, bestehen jedoch weitere Ansatzmöglichkeiten für zukünftige Entwicklungen. So wurde bisher nicht betrachtet, dass mit Java innere Klassen definiert werden können. Das sind Klassen, die sich innerhalb einer anderen Klasse und damit in der selben Quellcodedatei befinden. Jede dieser Klassen kann unabhängig von anderen ihre eigenen Methoden und Attribute definieren. Beispielsweise ist möglich, dass eine Klasse eine Methode definiert und eine innere Klasse besitzt, die wiederum selbst eine Methode unter gleichem Namen definiert. Ein ähnliches Problem stellten die überladenen Methoden innerhalb einer einzigen Klasse dar. Dieses konnte jedoch durch die Verwendung von regulären Ausdrücken umgangen werden. Allerdings ist es keine triviale Aufgabe, verschiedene Elemente mit gleichen Namen innerhalb einer Quellcodedatei verschiedenen Klassen zuzuordnen.

Die Lösung dieses Problems wäre ein interessantes Thema einer weiterführenden Arbeit. Einen nächsten möglichen Ansatzpunkt stellen *„Selektions-Chroniken“* dar. Diese können die Benutzerfreundlichkeit erhöhen, indem sie speichern, in welcher Reihenfolge welche Elemente selektiert wurden. So kann es von Nutzen sein, über möglichst wenige Schritte zur vorherigen oder zur nächsten Selektion zu springen, um die Informationen verschiedener Elemente vergleichend zu analysieren.

Weiterhin wäre zu beachten, dass die Visualisierung nicht auf Java-Projekte beschränkt sein soll. Zum Auffinden von Methoden- und Attributdefinitionen wurden jedoch Annahmen getroffen, die nur für Java spezifisch sind. Dazu wurde *Prism* um ein Plugin erweitert, welches jedem Schlüsselwort für die Sichtbarkeit eines Element eine bestimmte Klasse zuordnet. Diese wird zur Identifikation von Definitionen herangezogen. In anderen Sprachen müssen dabei jedoch nicht die gleichen Schlüsselwörter auftreten. Ebenfalls können einfache Formatierungen, wie beispielsweise Zeileneinzüge oder Absätze, eine Definition einleiten. *Prism* kann hierzu um ein weiteres Plugin erweitert werden, um die Interaktionen für eine andere Sprache zu unterstützen sowie den Entwicklungsaufwand gering zu halten.

Zum Abschluss dieser Arbeit wird noch einmal auf einen Punkt der Motivation zurückgegriffen. So wurde einbezogen, dass nicht alle Stakeholder eines Softwareprojektes die gleichen Kenntnisse über eine Software besitzen. Die Visualisierung dient dabei als ein Mittel der Kommunikation, unter anderem zwischen Entwicklern aus verschiedenen Bereichen. Das Mittel der Visualisierung wurde mit dieser Arbeit erweitert. Jetzt kann die Software über den direkten Vergleich zwischen der Darstellung der Elemente in der Visualisierung und der tatsächlichen Umsetzung innerhalb des Quellcodes analysiert werden.

Literaturverzeichnis

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

[Balzert 2009b] Balzert, H., UML 2 in 5 Tagen: Der schnelle Einstieg in die Objektorientierung. 2. Auflage. W3l GmbH, Herdecke, 2009.

[Berners-Lee 1999] Berners-Lee, T., Der Web-Report. Econ, München, 1999.

[BOC 2014] Nagel, F., beautyofcode, URL: https://github.com/fnagel/beautyofcode, gelesen am 28.09.2015.

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

[DH 2015] Bazon, M., JavaScript Syntax Highlighting Engine, URL: http://mihai.bazon.net/projects/javascript-syntax-highlighting-engine, gelesen am 28.09.2015.

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

[ECMA 1961] ECMA International (Hrsg.), What is Ecma International, URL: http://www.ecma-international.org/memento/index.html, gelesen am 28.09.2015.

[ECMA 2015] ECMA International (Hrsg.), ECMA 2015 Language Specification – ECMA-262 6th Edition, URL: http://www.ecma-international.org/ecma-262/6.0/index.html#sec-scope, gelesen am 28.09.2015.

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

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

[Hesse et al. 1984] Hesse, W., Keutgen, H., Luft, A.L., Rombach, H.D., Ein Begriffssystem für die Softwaretechnik, in: Informatik-Spektrum 7, Springer-Verlag, Berlin, 1984, S. 200-213.

[HIGHLIGHT 2015] Sagalaev, I., highlight.js, URL: https://highlightjs.org/, gelesen am 28.09.2015.

[Hopcroft et al. 2011] Hopcroft, J.E., Motwani, R., Ullman, J.D., Einführung in Automatentheorie, Formale Sprachen und Berechenbarkeit. 3. Auflage. Pearson Studium, München, 2011.

[Knight/Munro 1999] Knight, C., Munro, M., Comprehension with[in] virtual environment visualisations, in: International Conference on Program Comprehension, IEEE, 1999, S. 4-11.

[Kovacs 2010] Kovacs, P., Ansätze zur Interaktion mit dreidimensional visualisierten Softwaremodellen. Diplomarbeit, Universität Leipzig, Leipzig, 2010.

[Lohse 1993] Lohse, G., A cognitive model for understanding graphical perception, in: Human-Computer Interaction Volume 8, L. Erlbaum Associates Inc., Hillsdale, 1993, S. 353-388.

[MDN 2015] Mozilla Foundation (Hrsg.), Reguläre Ausdrücke JavaScript | MDN, URL: https://developer.mozilla.org/de/docs/Web/JavaScript/Guide/Regular_Expressions, gelesen am 28.09.2015.

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

[Müller/Zeckzer 2015] Müller, R., Zeckzer, D., The Recursive Disc Metaphor: A Glyph-based Approach for Software Visualization, Universität Leipzig, Leipzig, 2015.

[OMG 2015] Object Management Group (Hrsg.), About OMG, URL: http://www.omg.org/gettingstarted/gettingstarted index.htm, gelesen am 28.09.2015.

[PRISM 2015] Verou, L., Prism, URL: http://prismjs.com/, gelesen am 28.09.2015.

[RAINBOW 2013] Campbell, C., Rainbow – Javascript Code Syntax Highlighting, URL: https://craig.is/making/rainbows, gelesen am 28.09.2015.

[Reijers et al. 2011] Reijers, H.A., Reytag, T., Mendling, J., Eckleder, A., Syntax highlighting in business process models, in Decision Support Systems Volume 51, Elsevier, 2011, S. 339-349.

[Robertson et al. 1991] Robertson, G.G., Mackinlay, J.D., Card, S.K., Cone trees: animated 3D visualizations of hierarchical information, in: CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems, ACM, New York, 1991, S. 189-194.

[SH 2011] Gorbatchev, A., SyntaxHighlighter, URL: http://alexgorbatchev.com/SyntaxHighlighter/, gelesen am 28.09.2015.

[SHJS 2008] o.V., SHJS – Syntax Highlight in JavaScript, URL: http://shjs.sourceforge.net/, gelesen am 28.09.2015.

[SL 2013] Montgomery, T., sunlight, URL: https://github.com/tmont/sunlight, gelesen am 28.09.2015.

[W3C 2014] W3C (Hrsg.), 4.4 Grouping content – HTML5, URL: http://www.w3.org/TR/html5/grouping-content.html#the-pre-element, gelesen am 28.09.2015.

[W3C 2015] W3C (Hrsg.), CSS Arbeitsgruppe, URL: https://www.w3.org/Style/CSS/members, gelesen am 28.09.2015.

[Web3D 2015a] Web3D Consortium (Hrsg.), X3D Standards for Version all, URL: http://www.web3d.org/standards/version/all, gelesen am 28.09.2015.

[Web3D 2015b] Web3D Consortium (Hrsg.), What is X3D, URL: http://www.web3d.org/x3d/what-x3d, gelesen am 28.09.2015.

[Web3D 2015c] Web3D Consortium (Hrsg.), X3D Resources, URL: http://www.web3d.org/x3d/content/examples/X3dResources.html, gelesen am 28.09.2015.

[WIMP 2014] Wikipedia (Hrsg.), WIMP (Benutzerschnittstelle), URL: https://de.wikipedia.org/wiki/WIMP_(Benutzerschnittstelle), gelesen am 28.09.2015.

[X3DOM 2015] Fraunhofer-Gesellschaft, -x3dom.org, URL: http://www.x3dom.org/, gelesen am 28.09.2015.

Ehrenwörtliche Erklärung

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

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

Yves Annanias

Leipzig, 06. Oktober 2015