View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Qucosa - Publikationsserver der Universität Leipzig

Ansatz zur Interaktion mit dreidimensional visualisierten Softwaremodellen

Dipl.-Wirt.-Inf. Pascal Kovacs
kovacs@wifa.uni-leipzig.de
Universität Leipzig
Wirtschaftswissenschaftliche Fakultät
Institut für Wirtschaftsinformatik

Kurzzusammenfassung

Softwaresysteme sind komplexe immaterielle Systeme mit einer Vielzahl von Bestandteilen und Beziehungen. Um den Aufbau, die Funktionsweise und die Entwicklung von Softwaresystemen besser zu verstehen, eignen sich Softwarevisualisierungen, welche die abstrakte Datengrundlage in eine visuelle Repräsentation übertragen. Auf Grund der Masse und der Komplexität der in der Visualisierung enthaltenen Informationen, kommt es schnell zur Unübersichtlichkeit, was sich negativ auf den Prozess des Verstehens auswirkt. Zur Beherrschung der Komplexität muss der Betrachter daher die Gesamtheit zuerst in mehrere Perspektiven unterteilen, um diese anschließend gezielt nach verschiedenen Aspekten untersuchen zu können.

Die dafür benötigten Interaktionsmöglichkeiten sind Gegenstand der Untersuchungen in dieser Arbeit, wobei im Wesentlichen Visualisierungen der Struktur von Software als Ausgangspunkt genutzt werden. Insbesondere wird der Frage nachgegangen, wie die Interaktion gestaltet werden kann, damit der Benutzer ein möglichst umfassendes Verständnis der Struktur erlangt.

Zur Umsetzung der theoretischen Erkenntnisse wird ein Prototyp vorgestellt, der automatisiert aus den Strukturinformationen eines Ecore-Modells eine interaktive dreidimensionale Softwarevisualisierung der Struktur im freien standardisierten Format Extensible 3D generiert. Der Prozess der Visualisierung wird dabei durch Werkzeuge des openArchitectureWare-Frameworks realisiert. Zur Integration in den Entwicklungsprozess ist der Prototyp in ein Plugin für Eclipse eingebettet.

Schlagwörter

Softwarevisualisierung, 3D, Interaktion, Extensible 3D, openArchitectureWare, Eclipse

Gliederung

Gliederung ... I

Abbildungsverzeichnis ... III

Tabellenverzeichnis ... V

Verzeichnis der Listings ... VI

Abkürzungsverzeichnis ... VII

1 Einleitung ... 1

1.1 Motivation ... 1

1.2 Ziele ... 2

1.3 Aufbau ... 3

2 Softwarevisualisierung ... 5

2.1 Visualisierung ... 5

2.1.1 Grundlagen ... 5

2.1.2 Aufgaben und Ziele ... 6

2.1.3 Teilgebiete ... 8

2.2 Definition ... 8

2.3 Ziele und Aufgaben ... 10

2.4 Visualisierungspipeline ... 11

2.5 Visualisierungstechniken ... 13

2.5.1 Graphen ... 13

2.5.2 Visuelle Metaphern ... 17

2.6 Taxonomie für Softwarevisualisierungen ... 19

3 Interaktion mit einer Softwarevisualisierung ... 22

3.1 Einordnung ... 22

3.2 Definition ... 23

3.3 Ziele ... 24

3.4 Benutzungsschnittstelle ... 27

3.5 Aufbau der Benutzungsschnittstelle ... 29

3.5.1 Anwendungsschnittstelle ... 30

3.5.2 Dialogschnittstelle ... 30

3.5.3 Ein-/Ausgabenschnittstelle ... 31

3.6 Interaktionstechnik ... 32

3.7 Taxonomie der Interaktionstechniken ... 34

3.8 Konzept zur Interaktion mit einer Softwarevisualisierung ... 38

4 Technische Grundlagen des Prototyps ... 42

4.1 Eclipse ... 42

4.2 Das openArchitectureWare-Framework....44
4.2.1 Modellgetriebene Softwareentwicklung....44
4.2.2 Aufbau des Frameworks....46
4.2.3 oAW-Workflow....47
4.2.4 Ecore....49
4.2.5 Xtend Modell-zu-Modell-Transformationen....50
4.3 Extensible 3D....55
4.3.1 Grundlagen....55
4.3.2 Szenegraph....57
4.3.3 Ereignismodell....59
4.3.4 X3D-Prototypen....62
5 Basisprototyp....65
5.1 Funktionsweise....65
5.2 Einordnung....68
5.3 Visualisierungsprozess des Generators....69
5.3.1 Modelltransformation von Ecore zu Graph....70
5.3.2 Modellmodifikation des Graphen....71
5.3.3 Modelltransformation von Graph zu X3D....72
5.4 Ansatzpunkte der Erweiterung....74
6 Erweiterung des Prototyps für die Interaktion....75
6.1 Benutzungsschnittstelle....75
6.1.1 Direkte Manipulation....77
6.1.2 Manipulation nach Elementtyp....78
6.1.3 Navigation durch Pakethierarchie und Klassengraph....80
6.1.4 Identifikation nach Bezeichner und Tooltip....81
6.2 Architektur des interaktiven X3D-Modells....82
6.3 Erweiterung des Generators....85
6.3.1 Anpassung der Transformationen des Basisprototyps....85
6.3.2 Modelltransformation in ein interaktives X3D-Modell....87
6.4 Integration in das Eclipse-Plugin....90
7 Fazit und Ausblick....92
Literaturverzeichnis....VIII
Anhang....XII
A.1 X3D-Architektur: ManipulationLayer....XII
A.2 X3D-Architektur: SelectionLayer....XII
A.3 X3D-Architektur: NavigationLayer....XIII

Abbildungsverzeichnis

Abbildung 1: Stufen der Informationen: Beispiel 5
Abbildung 2: Visualisierungspipeline 12
Abbildung 3: Cone-Tree 15
Abbildung 4: Modifizierte Heatmap ohne Färbung 16
Abbildung 5: Information cube 16
Abbildung 6: Kugelbaum 17
Abbildung 7: Information-Landscape-Metapher 18
Abbildung 8: Abstrakte Graphen-Metapher 18
Abbildung 9: City-Metapher: Variante 1 18
Abbildung 10: City-Metapher: Variante 2 18
Abbildung 11: Interaktion: Überblick 24
Abbildung 12: Benutzungsschnittstelle: Überblick 27
Abbildung 13: Benutzungsschnittstelle: Aufbau 29
Abbildung 14: Interaktionstechnik: Übersicht 33
Abbildung 15: Interaktionstechnik: Beispiel 33
Abbildung 16: Eclipse-Oberfläche 42
Abbildung 17: Ecore-Klassenmodell: Auszug 49
Abbildung 18: X3D-Beispiel: Hello World! 59
Abbildung 19: X3D-Szenegraph: Baumstruktur 59
Abbildung 20: X3D-Ereignismodell: Überblick 61
Abbildung 21: X3D-Ereignismodell: Beispiel Farbwechsel nach Zeit 61
Abbildung 22: X3D-Ereignismodell: Beispiel Überblick 62
Abbildung 23: X3D-Prototyp: Beispiel Farbwechsel nach Berührung 63
Abbildung 24: X3D-Prototyp: Beispiel Knoteüberblick 63
Abbildung 25: Klassendiagramm: Beispiel familytree 67
Abbildung 26: Eclipse-Integration: Basisprototyp 67
Abbildung 27: X3D-Modell: Beispiel familytree 68
Abbildung 28: Visualisierungsprozess Basisprototyp 70
Abbildung 29: Graphenmodell: Aufbau 71
Abbildung 30: GUI: Überblick 77
Abbildung 31: GUI: Selektion direkt 79
Abbildung 32: GUI: Positionierung direkt 79
Abbildung 33: GUI: Filter Typ 80
Abbildung 34: GUI: Selektion Typ 80
Abbildung 35: GUI: Navigation Pakethierarchie 81

Abbildung 36: GUI: Navigation Klassengraph..82
Abbildung 37: GUI: Selektion Name, Tooltip...83
Abbildung 38: X3D-Architektur: Schichtenmodell..84
Abbildung 39: X3D-Architektur: Ablauf..85
Abbildung 40: X3D-Architektur: Steuerelement..86
Abbildung 41: Eclipse-Einstellungsseite: Interaktion.......................................91
Abbildung 42: X3D-Architektur: ManipulationLayer.....................................XII
Abbildung 43: X3D-Architektur: SelectionLayer...XII
Abbildung 44: X3D-Architektur: NavigationLayer.......................................XIII
Abbildung 45: X3D-Architektur: NavigationLayer OneWayRouter........................XIII

Tabellenverzeichnis

Tabelle 1: Aspekte Softwarevisualisierung....9

Tabelle 2: Gegenüberstellung der Taxonomien 19

Tabelle 3: Ziele Interaktion mit Softwarevisualisierung der Struktur....24

Tabelle 4: Eigenschaften Gestaltung Ein- und Ausgabe nach ISO 9241-12....32

Tabelle 5: Ziele der Interaktion und realisierende Interaktionstechniken....41

Tabelle 6: Modellebenen MDA....46

Tabelle 7: X3D-Komponenten: Beispiele 56

Tabelle 8: X3D-Felder: Zugriffsarten....64

Verzeichnis der Listings

Listing 1: oAW-Workflow: Beispiel.....48
Listing 2: Xtend-Ausdruckssprache: Kontrollstrukturen.....51
Listing 3: Xtend-Ausdruckssprache: Mengen und Mengenoperationen.....52
Listing 4: Xtend-Ausdruckssprache: Verketteter Ausdruck.....52
Listing 5: Xtend-Erweiterungen: Definition und Aufruf.....53
Listing 6: Xtend-Erweiterungen: Erstellen von Objekten.....54
Listing 7: X3D-Dateistruktur.....58
Listing 8: X3D-Szenegraph: Hello World!.....59
Listing 9: X3D-Ereignismodell: Beispiel Szenegraph.....62
Listing 10: X3D-Prototyp: Beispiel Definition.....64
Listing 11: X3D-Prototyp: Beispiel ECMAScript.....65
Listing 12: Ecore-zu-Graphenmodell: Auszug Xtend-Transformation.....72
Listing 13: Graphenmodell: Beispiel Cluster.....72
Listing 14: Graphenmodel-Modifikation: Beispiel Cluster.....73
Listing 15: Graphenmodell-zu-X3D: Auszug Xtend-Transformation.....74
Listing 16: Graphenmodell-zu-X3D: Auszug Ergebnis.....74
Listing 17: Generator-Anpassung: Vollqualifizierter Name.....87
Listing 18: Generator-Anpassung: Auszug statisches X3D-Modell.....88
Listing 19: Modellmodifikation zur Interaktion: Workflow-Aufruf.....89
Listing 20: Modellmodifikation zur Interaktion: Xtend-Auszug.....90

Abkürzungsverzeichnis

DIN	Deutsches Institut für Normung
DSL	Domain Specific Language
ECMA	European Computer Manufacturers Association
EMF	Eclipse Modeling Framework
EN	Europäische Norm
EPL	Eclipse Public License
GUI	Graphical User Interface
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
LOC	Lines of Code
M2C	model-to-code transformation
M2M	model-to-model transformation
MCI	Mensch-Computer-Interaktion
MDA	Model Driven Architecture
MDSD	Model Driven Software Development
MOF	Meta Object Facility
oAW	OpenArchitectureWare
OMG	Object Management Group
UML	Unified Modeling Language
VRML	Virtual Reality Modeling Language
X3D	Extensible 3D
XML	Extensible Markup Language

1 Einleitung

Software durchdringt immer mehr Bereiche des persönlichen und gesellschaftlichen Lebens, was durch einen anhaltenden technischen Fortschritt noch verstärkt wird. Dies erzeugt auf der einen Seite eine hohe Nachfrage nach Softwarelösungen und bringt auf der anderen Seite viele neue, meist sehr komplexe Anforderungen mit sich. Diese Situation wirkt sich nachhaltig auf bestehende Softwareentwicklungsprozesse aus, die zusammen mit ihrem Entwicklungsgegenstand noch einmal an Komplexität und Umfang zulegen.

Neben der Entwicklung neuer Softwaresysteme sorgt auch der Umgang mit Altsystemen für Herausforderungen. So müssen nicht nur die Fragen der Wartung und der Weiterentwicklung alter Systeme geklärt werden, sondern auch, wie sich diese in Teilen beziehungsweise als Ganzes austauschen lassen. Besonders problematisch wird es, wenn die bestehenden Systeme ungenügend dokumentiert sind oder wenn es an fachkundigem Personal mit Systemkenntnis fehlt. In diesem Fall ist nicht die Implementierung sondern der Prozess des Verstehens der Software die eigentliche Herausforderung.

Die Beantwortung der Frage nach dem Umgang mit den gestiegenen Anforderungen, sei es bei der Neuentwicklung oder beim Umgang mit Altsystemen, wird noch durch eine entscheidende Eigenschaft von Software verschärft. Software ist immateriell, was das Verständnis von ihrer Struktur, ihrem Ablauf und ihrer Entwicklungshistorie deutlich erschwert. Der Quellcode, häufig die einzige Grundlage für eine Analyse, enthält zudem nicht alle für das Verständnis relevanten Informationen, wie Anforderungen oder Entwurfsentscheidungen. Auch erreichen die Artefakte schnell eine kritische Größe in Anzahl und Umfang, was den Überblick erschwert und sich somit negativ auf das Verständnis auswirkt.

1.1 Motivation

Einen Beitrag zur Lösung dieser Problematik bietet die Softwarevisualisierung. Die aus verschiedenen Artefakten stammende Datengrundlage wird in eine visuelle Form überführt. Auf diese Weise soll der Betrachter einerseits Zusammenhänge zwischen einzelnen Konstrukten erkennen und somit einen Überblick über das System erlangen. Andererseits sollen aber auch Konzepte und Ideen, die den konkreten Daten zu Grunde liegen, identifiziert werden können.

In der Literatur wurden bereits viele Grundlagen und Ansätze zur Softwarevisualisierung erarbeitet und beschrieben. Auch existieren bereits mehrere Systeme zur Visualisierung von Software, die allerdings zum Teil mit erheblichen Akzeptanzproblemen behaftet sind. Die Studie von Bassil & Keller zeigt, dass die ungenügende Integration in den

Entwicklungsprozess und fehlende standardisierte Formate für Im- und Export die Hauptkritikpunkte darstellen (Bassil & Keller 2001). Speziell für Softwarevisualisierungen in 3D stellt der Automatisierungsgrad einen weiteren Kritikpunkt dar. Ist der manuelle Aufwand zur Erzeugung zu hoch, kann dies sogar zur Ablehnung der Visualisierung insgesamt führen (Stasko & Patterson 1991).

Müller setzt an diesen Kritikpunkten an und konzipiert einen Generator zur Softwarevisualisierung in 3D, den er durch die Implementierung eines Prototypen realisiert (R. Müller 2009). Dieser generiert ausgehend von den Strukturinformationen der Software automatisiert eine dreidimensionale Visualisierung der Struktur. Der Generator ist dabei in eine Entwicklungsumgebung integriert und nutzt standardisierte, portierbare Formate. Im Folgenden wird auf die Erkenntnisse von Müller und seinem als Prototyp bezeichneten Generator aufgebaut.

Der Grad des Verständnisses, der mit einer Softwarevisualisierung erlangt werden kann, ist von mehreren Faktoren abhängig. So kann eine Darstellung, die zu viele Informationen gleichzeitig visualisiert, sogar hinderlich für das Verständnis sein. Besonders bei komplexen Softwaresystemen stößt jede Repräsentationsform schnell an eine Grenze, in der selbst unter Einbezug der dritten Dimension der Überblick verloren geht. Auch die von Müller gewählte Repräsentationsform bildet hier keine Ausnahme. Zudem kommt hinzu, dass sich Beziehungen durch bloße Betrachtung nur schwer erfassen lassen. Gerade Software besitzt jedoch eine große Anzahl unterschiedlicher Beziehungen, deren Erfassen grundlegend für das Gesamtverständnis ist. Des Weiteren lassen sich durch bloße Darstellung der Datengrundlage die dahinter liegenden Ideen und Konzepte nicht sofort erkennen. Oftmals werden sie nur indirekt in eine visuelle Form übertragen oder sind durch andere Einflüsse überlagert.

Eine Möglichkeit diese Defizite zu überwinden, stellt die Interaktion dar, die in Literatur und Praxis aber oftmals zugunsten der Darstellung vernachlässigt wird. Dabei findet Interaktion selbst bei einer statischen Abbildung statt. Der Betrachter kann die Abbildung drehen, einen Ausschnitt näher betrachten oder eine Stelle markieren. Interaktion ist somit ein elementarer Bestandteil im Prozess der Erkenntniserlangung über die zugrunde liegende Software, da der Benutzer die Darstellung seinen Erfordernissen anpassen kann.

1.2 Ziele

Aufbauend auf den Ergebnissen von Müller soll im Folgenden untersucht werden, wie durch die Interaktion mit einer Softwarevisualisierung der Prozess des Verstehens von Software unterstützt werden kann. Zunächst müssen dazu die Ziele identifiziert werden,

die ein Benutzer bei der Interaktion mit einer Softwarevisualisierung verfolgt. Der Fokus der Untersuchung liegt dabei auf dem Verständnis des Aufbaus von Software und seiner Struktur. Beispiele sind das Nachvollziehen der Vererbungsbeziehungen und das Identifizieren der *get*- und *set*-Methoden von Klassen. Während im ersten Beispiel der Zusammenhang zwischen einzelnen Elementen der Software erfasst werden soll, werden im zweiten Beispiel Eigenschaften von Elementen identifiziert, die wiederum Rückschlüsse auf deren Verwendung zulassen.

Aufbauend auf den Zielen der Interaktion sollen die Gestaltungsmöglichkeiten der Interaktion mit dreidimensionalen Softwaremodellen untersucht werden, die es dem Benutzer ermöglichen, diese Ziele zu erreichen. Die Betrachtung erfolgt dabei aber nicht nur nach technischen Gesichtspunkten, sondern berücksichtigt auch den Einfluss der kognitiven Bedürfnisse des Benutzers sowie das eigentliche Ziel, welches der Benutzer mit einer bestimmten Form der Interaktion verfolgt.

Die Umsetzung der Erkenntnisse soll durch die Erweiterung des von Müller entwickelten Prototyps erfolgen. Dafür müssen Erweiterungspunkte identifiziert werden, die es unter den technischen Gegebenheiten des Prototyps erlauben, ein interaktives dreidimensionalen Softwaremodell zu generieren, das den Benutzer beim Verständnis des Aufbaus und der Struktur der Software unterstützt. Die Erweiterungen müssen dabei so gestaltet werden, dass zukünftige Arbeiten darauf aufbauen können, um beispielsweise Interaktionsmöglichkeiten für den Ablauf von Software ergänzen zu können.

1.3 Aufbau

Die Arbeit gliedert sich in sieben Kapitel einschließlich der Einleitung. Im zweiten Kapitel werden die theoretischen Grundlagen der Softwarevisualisierung erarbeitet, die für das weitere Verständnis der Arbeit von wesentlicher Bedeutung sind. Nach einem kurzen Abriss zur Visualisierung wird eine Arbeitsdefinition hergeleitet und die Aufgabengebiete und Ziele der Softwarevisualisierung werden vorgestellt. Dem folgen die grundlegenden Konzepte sowie konkrete Techniken zur Visualisierung. Das Kapitel schließt mit einer Taxonomie zur Klassifizierung von Visualisierungen.

Die Interaktion im Kontext der Softwarevisualisierung wird im dritten Kapitel behandelt. Neben einer Abhandlung zu den Zielen wird auch eine Definition gegeben, die auf Grundlage der Mensch-Computer-Interaktion verfasst ist. Des Weiteren wird der grundlegende Aufbau der Interaktion mit einem System behandelt und es werden Einblicke in die Gestaltungsmöglichkeiten gegeben, die im Rahmen dieser Arbeit relevant sind. Anschließend wird eine Taxonomie vorgestellt, die das Klassifizieren einer

Interaktionsmöglichkeit im Kontext ihrer Verwendung erlaubt. Im letzten Teil des Kapitels werden die erarbeiteten Erkenntnisse zusammengefasst und ein Konzept einer Benutzungsschnittstelle für die Interaktion mit einer Softwarevisualisierung der Struktur vorgestellt.

Die technischen Grundlagen für die Umsetzung der Benutzungsschnittstelle bilden das vierte Kapitel. Dabei werden die Entwicklungsumgebung, in die der Generator integriert ist, die Werkzeuge, die für seine Implementierung verwendet werden, und das Format der Visualisierung vorgestellt.

Die Ausführungen zur Umsetzung der Interaktionsmöglichkeiten beginnt mit der Beschreibung des Prototyps von Müller im fünften Kapitel. Neben dem Ausgangspunkt und dem Ergebnis wird auch der Ablauf des Visualisierungsprozesses in Auszügen behandelt. Zum Ende des Kapitels werden dann die Ansatzpunkte zur Erweiterung aufgezeigt.

Das sechste Kapitel stellt den um Interaktionsmöglichkeiten erweiterten Prototypen im Detail vor. Zuerst werden die implementierten Möglichkeiten zur Interaktion und den damit zu realisierenden Zielen vorgestellt. Anschließend werden Einblicke in die Architektur der Umsetzung gegeben sowie Auszüge aus relevanten Bereichen des Implementierung vorgestellt.

Das Fazit der Arbeit sowie ein Ausblick auf weitere Arbeiten zur Interaktion mit einer Softwarevisualisierung bilden im siebten Kapitel den Abschluss der Arbeit.

2 Softwarevisualisierung

Dieses Kapitel legt das theoretische Fundament für die weiteren Betrachtungen. Da Softwarevisualisierung als Teilgebiet der Visualisierung gilt, erfolgt im ersten Unterkapitel ein kurzer Abriss zur Visualisierung. Im Weiterem wird eine Definition für Softwarevisualisierung hergeleitet und anschließend die Aufgaben und Ziele vorgestellt. Dem folgt die Betrachtung der Visualisierungspipeline im vierten Unterkapitel, welche den Prozess der Visualisierung repräsentiert. Im fünften Unterkapitel werden zwei grundlegende Techniken der Visualisierung, nämlich Graphen und Metaphern, vorgestellt und im sechsten Unterkapitel werden die Möglichkeiten zur Klassifizierung von Softwarevisualisierungen beschrieben.

2.1 Visualisierung

2.1.1 Grundlagen

Card et al. definieren Visualisierung als „*the use of computer-supported, interactive, visual representations of data to amplify cognition*“ (Card et al. 1999). Eine Visualisierung dient somit dem Erlangen oder dem Erweitern von Erkenntnissen über eine zugrunde liegende Menge von Daten. Die in der Arbeit folgenden Ausführungen zur Visualisierung nehmen diese Definition als Grundlage.

Bertin & Scharfe unterscheiden drei Stufen von Informationen, welche in einer Abbildung dargestellt werden können (Bertin & Scharfe 1982). Die erste Stufe umfasst die direkte Abbildung der Daten aus dem Datenbestand, womit jede Information eine entsprechende Repräsentation in der Abbildung besitzt. Beispielsweise könnte die Anzahl der Zeilen (engl. lines of code, LOC) pro Klasse wie in Abbildung 1 in einem Koordinatensystem abgetragen werden. Jeder Punkt ist somit ein Datensatz aus LOC und Klasse. In der zweiten Stufe wird bereits von den zugrunde liegenden Daten abstrahiert.

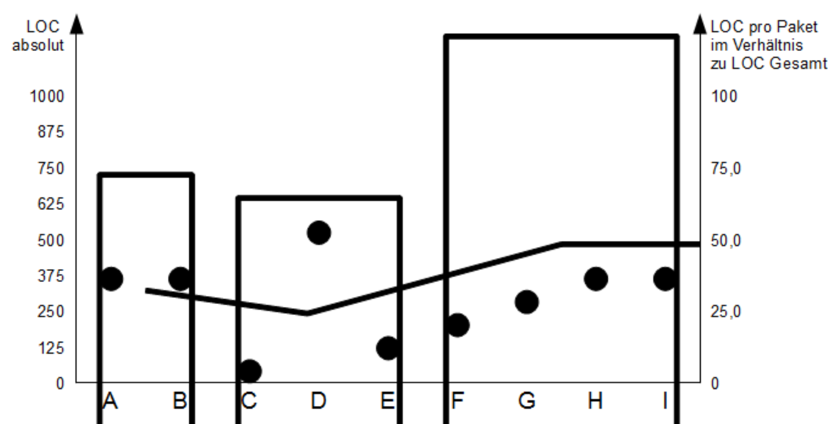

Abbildung 1: Stufen der Informationen: Beispiel

Wesentliche Zusammenhänge und Beziehungen der Struktur können somit vom Betrachter nachvollzogen werden. Im Beispiel der LOC könnten die Klassen nach ihren Paketen gruppiert und die LOC der einzelnen Klassen aufsummiert werden, was in Abbildung 1 durch drei große Rechtecke dargestellt ist. Somit lassen sich auf einem Blick die einzelnen Pakete vergleichen, während gleichzeitig die LOC der einzelnen Klassen enthalten sind. Die letzte Stufe visualisiert alle in den Daten enthaltenen Informationen und wird als „Gesamtheit der Informationen" bezeichnet, womit sie folglich auch die Informationen der beiden ersten Stufen enthält. Entsprechend wurde die Abbildung um eine Kurve ergänzt, welche die LOC der Pakete im Verhältnis zur gesamten Zahl der LOC zeigt.

2.1.2 Aufgaben und Ziele

Schumann & Müller identifizieren die Erkenntniserlangung und die Kommunikation als die zwei grundlegenden Anwendungsfälle für eine Visualisierung (Schumann & W. Müller 1999). Im ersten Fall soll mit Hilfe einer visuellen Repräsentation eine Vorstellung über die Daten erlangt beziehungsweise entwickelt werden. Enthält die Abbildung nur Informationen der ersten Stufe, so kann lediglich ein grober Überblick über die vorhandene Datenbasis erlangt werden. Die Anwendungsmöglichkeiten sind dem entsprechend stark begrenzt. Im Beispiel der LOC könnte also lediglich festgestellt werden, welche Klasse wie viele LOC besitzt, und dass sich diese Anzahl von Klasse zu Klasse unterscheidet. Da aber auch ein abstrakteres Verständnis der zugrunde liegenden Ideen und Konzepte gefördert werden soll, muss die Visualisierung auch Informationen der zweiten Stufe enthalten. Somit können auch implizit enthaltene Zusammenhänge erkannt werden, welche mit der ausschließlichen Betrachtung der Datengrundlage nicht oder nur schwer nachvollzogen werden können. So kann aus Abbildung 1 entnommen werden, dass die Klassen F bis I für sich genommen recht klein sind, ihr Paket aber fast die Hälfte des Programms ausmacht. Für die Ableitung von Entscheidungen muss die Abbildung jedoch die dritte Stufe der Informationen besitzen, also über die Gesamtheit der Informationen verfügen. Ist der Großteil der LOC zum Beispiel auf ein Paket verteilt, stellt sich die Frage, ob das System gut strukturiert ist. Ausgehend von der Visualisierung könnte somit die Entscheidung getroffen werden, das Paket in mehrere kleinere aufzuteilen.

Gemäß dem Sprichwort „Ein Bild sagt mehr als 1000 Worte" stellt die Kommunikation den zweiten Anwendungsfall dar. Dabei unterstützt die Abbildung die Kommunikation zwischen mehreren Personen, zum Beispiel, um eine Idee oder Vorstellung zu erläutern. Im Folgenden ist dieser Anwendungsfall für sich genommen jedoch nicht Gegenstand der

Betrachtung, obwohl die Kommunikation auch zu Erkenntnissen führen kann und Erkenntnisse andererseits auch kommuniziert werden müssen.
Das wichtigste Ziel der Visualisierung für die weiteren Betrachtungen ist somit die optimale Unterstützung des Prozesses der Erkenntniserlangung, da nur durch diesen die Zusammenhänge und Beziehungen erfasst und somit auch für Entscheidungen genutzt werden können. Das Erreichen dieses Ziels hängt im wesentlichen von der Qualität der Visualisierung und den entsprechenden Einflussfaktoren ab. Bei schlechter Qualität droht sogar die Gefahr der Fehlinterpretation der Daten und des Ableitens falscher Entscheidungen (Schumann & W. Müller 1999). Die quantitative und objektive Erfassung der Qualität ist allerdings ein schwieriges und komplexes Vorhaben.
Nach Schumann & Müller ist die Qualität einer visuellen Repräsentation definiert als das Maß, „*in dem der Betrachter fähig ist, den Kontext der realen Welt aus der Abbildung zu rekonstruieren, die wahrgenommenen Strukturen in der Abbildung mit tatsächlich existierenden Korrelationen zwischen Parametern bzw. Störungen in den Ergebnisdaten in Verbindung zu bringen und die richtigen Schlüsse zu ziehen*“ (Schumann & W. Müller 1999). Demzufolge ist die Qualität eng mit dem Prozess der Erkenntniserlangung verbunden und somit auch stark abhängig vom jeweiligen Betrachter. Die objektive Bewertung der Qualität wird somit zu einer herausfordernden Aufgabe, da die visuellen Fähigkeiten und Vorlieben, genauso wie das Vorwissen eines Betrachters, entscheidenden Einfluss darauf besitzen. Weitere Einflussfaktoren ergeben sich zudem aus der Art und Struktur der Daten, dem Ziel der Visualisierung, den üblichen Metaphern des Anwendungsgebietes sowie typischen Eigenschaften des Darstellungsmediums (Schumann & W. Müller 1999).
Zudem existieren die drei allgemeine Anforderungen *Expressivität, Effektivität und Angemessenheit*, welche sich teilweise mit den Qualitätsanforderung von Schumann & Müller überschneiden. Die wichtigste Anforderung an eine Visualisierung ist dabei die *Expressivität*. Die zugrunde liegenden Daten müssen unverfälscht wiedergegeben werden und dürfen keine Informationen suggerieren, die nicht in der Datenbasis vorhanden sind. Die Expressivität oder auch *Ausdrucksmächtigkeit* (Mackinlay 1986) ist somit die Grundvoraussetzung für Qualität und im Wesentlichen abhängig von der Art und der Struktur der Datenbasis (Schumann & W. Müller 1999).
Die *Effektivität* bezeichnet hingegen die Eignung einer Visualisierungsmethode zur Unterstützung des Prozesses der Erkenntniserlangung (Mackinlay 1986). Neben der reinen Datengrundlage müssen dabei die visuellen Vorlieben und Fähigkeiten des Betrachters, die Eigenschaften des Darstellungsmedium sowie der Anwendungskontext und die Zielsetzung

berücksichtigt werden. Diese Anforderung entspricht im Wesentlichen dem von Schumann & Müller definierten Qualitätsbegriff.
Das Verhältnis zwischen Aufwand und Kosten auf der einen Seite und dem Nutzen auf der anderen wird durch die Angemessenheit beschrieben (Schumann & W. Müller 1999). Sind der Aufwand oder die Kosten einer Visualisierung im Vergleich zum Nutzen zu hoch, sollte die gesamte Visualisierung in Frage gestellt und gegebenenfalls verworfen werden.

2.1.3 Teilgebiete

Card et al. unterscheidet anhand der zugrunde liegenden Daten zwei Teilgebiete (Card et al. 1999). Die wissenschaftliche Visualisierung stellt Daten und Simulationsergebnisse aus der wissenschaftlichen Forschung dar, denen direkt natürliche Prozesse zugeordnet werden können. Die Informationsvisualisierung zeigt hingegen Daten, welche nicht direkt einem Prozess der Natur zugeordnet werden können. Sie visualisiert somit abstrakte Daten wie Geldflüsse oder Geschäftsprozesse, während die wissenschaftliche Visualisierung konkrete Daten wie physikalische Messwerte abbildet.
Bei der Software als Datengrundlage der Visualisierung ist eine Eigenschaft entscheidend für die Einordnung in eines der beiden Teilgebiet. Software ist immateriell, womit sie als Datengrundlage einen abstrakten Charakter aufweist und nicht direkt einem natürlichen Prozess zugeordnet werden kann. Anhand dieses Kriteriums lässt sich die Softwarevisualisierung dem Teilgebiet der Informationsvisualisierung zuordnen. Als Folge dieser Zuordnung lassen sich Konzepte und Verfahren der Informationsvisualisierung auf die Softwarevisualisierung übertragen (Diehl 2007).

2.2 Definition

In der Literatur findet sich eine Vielzahl von Definitionen zur Softwarevisualisierung. So geben beispielsweise Knight & Munro folgende Definition:

> „*Software visualisation is a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration.*“ (Knight & Munro 1999)

Wie auch andere Autoren (Price et al. 1993; Roman & Cox 1993; Myers 1990) beziehen sich Knight & Munro auf die Visualisierung eines konkret existierenden Systems mittels Darstellung von Struktur und Verhalten. Es hat sich allerdings gezeigt, dass diese Definition zu eng gefasst ist und eine Menge von Anwendungsfällen der Visualisierung ausschließt. Diehl definiert Softwarevisualisierung daher in einem weiteren Sinne als „[...] *the visualization of artifacts related to software and its development process* […]“ (Diehl 2007). Dadurch erfasst er neben dem Quellcode auch Dokumente über Anforderungen,

Design und Planung genauso wie Änderungen im Quellcode oder Fehlerberichte. Diese weit gefasste Definition von Softwarevisualisierung dient im Folgenden als Arbeitsdefinition. Konkret wird Softwarevisualisierung als Visualisierung von Artefakten der Struktur, des Verhaltens und der Evolution von Software aufgefasst. Jedes Artefakt der Softwareentwicklung kann dabei mindestens einem dieser drei Aspekte zugeordnet werden (Diehl 2007).

Artefakte mit statischen Informationen – wie Quellcode, Datenstrukturen oder der Aufteilung in Teilsysteme – bilden die Struktur eines Systems ab. Neben den für die Ausführung der Software relevanten Artefakten zählen aber auch Diagramme, wie Klassendiagramme oder Paketdiagramme, zur Struktur. Die Gemeinsamkeit aller Strukturartefakte ist, dass sie ohne Ausführung der Software entstehen beziehungsweise verfügbar sind. Die vorliegende Arbeit wird sich auf diesen Aspekt der Software fokussieren, während die anderen beiden Aspekte nur der Vollständigkeit halber aufgezeigt werden sollen. Das konkrete Verhalten eines Systems wird durch dynamische Informationen zur Laufzeit ermittelt. Die so erzeugten Artefakte können beispielsweise zeitlich aufeinander folgende Zustände der Software mit Quellcode und Daten zum jeweiligen Zeitpunkt enthalten. Sie können aber auch, abhängig von der verwendeten Programmiersprache, abstraktere Informationen wie Funktionsaufruffolgen oder Objektkommunikation enthalten. Die dritte Art von Informationen bezieht sich auf die Veränderung von Software im Laufe der Zeit. Zum einen muss sie erst einmal entwickelt werden, was in der Regel ein längerfristiger und komplexer Prozess ist. Zum anderen müssen an der entwickelten Software Änderungen vorgenommen werden, um Fehler zu beseitigen oder neue Funktionen zu integrieren. Diese historischen Informationen bilden den Aspekt der Evolution von Software ab und können in verschiedensten Artefakten, wie beispielsweise einem Fehlerbericht enthalten sein oder aus einem Versionsverwaltungssystemen generiert werden. Tabelle 1 fasst die drei Aspekte der Softwarevisualisierung in einer Übersicht zusammen.

Softwareaspekt	Informationen	Artefakte	Verfügbarkeit
Struktur	statisch	Quellcode, Diagramme	ohne Ausführung des Systems
Verhalten	dynamisch	Aufrufgraphen, Zustandsaufnahmen	nur nach Ausführung des Systems
Evolution	historisch	Änderungsberichte, Versionsverläufe	protokollierte Änderungen des Systems

Tabelle 1: Aspekte Softwarevisualisierung

2.3 Ziele und Aufgaben

Aus dem Ziel der Visualisierung (Kapitel 2.1.2) und der Defintion der Softwarevisualisierung (Kapitel 2.2) lässt sich das Ziel der Softwarevisualisierung folgendermaßen beschreiben: Sie soll es dem Betrachter ermöglichen, die Struktur, das Verhalten und die Evolution von Software zu verstehen.

Die besondere Herausforderung besteht dabei in der Immaterialität der Software, wodurch bereits die Datengrundlage abstrakt und losgelöst von natürlichen Prozessen ist. Aus dieser Zielstellung lassen sich entlang des Softwarelebenszyklus gemäß Bohnet & Döllner folgende Aufgabengebiete ableiten (Bohnet & Döllner 2005):

- Planung, Entwicklung und Testen neuer Softwaresysteme
- Wartung, Erweiterung und Wiederverwendung existierender Systeme
- Management und Monitoring von Softwareprojekten

Während der Entwicklung entsteht eine Vielzahl von Artefakten mit statischen, dynamischen und historischen Informationen über die Software. Der Fokus liegt dabei auf der Entwicklung der Software, die den gestellten Anforderungen genügt. Dementsprechend dienen Visualisierungen in diesem Anwendungsgebiet hauptsächlich zum Planen, Koordinieren und Überprüfen der Implementierung. Durch dieses vorwärts gerichtete Vorgehen ist es nicht zwingend, dass nach der Implementierung die Bindung zur Visualisierung erhalten bleibt. Viele Konzepte und Ideen der Planung sind – wenn überhaupt – nur noch implizit im Code enthalten. Eine übliche Form für die Visualisierung in diesem Bereich sind Diagramme, welche Modelle des Systems oder Teilmodelle unter verschiedenen Aspekten abbilden. Einen leistungsstarken Ansatz für die Modellierung und Visualisierung bietet die Unified Modeling Language[1] (UML) mit einer Vielzahl von standardisierten Diagrammen, wie beispielsweise dem Klassendiagramm oder dem Sequenzdiagramm. Noch einen Schritt weiter geht die modellgetriebene Softwareentwicklung, bei der aus den Modellen lauffähiger Quellcode generiert wird. Auf dieses Paradigma wird im Kapitel 4.2.1 noch näher eingegangen.

„Die Softwaresysteme von heute sind die Altsysteme von morgen“. Dieser Satz beschreibt eine wichtige Tatsache in der Softwareentwicklung. Die Wartung, Erweiterung und Wiederverwendung alter Systeme, vollständig oder in Teilen, stellt eine zunehmende Herausforderung dar. Oftmals ist nur der Quellcode als Grundlage vorhanden, während Dokumentation, Diagramme oder Experten für das System, manchmal sogar für die Programmiersprache, fehlen. Wie bereits erwähnt, sind aber die zu Grunde liegenden Konzepte, Modelle und Ideen der Planung nur implizit im Code enthalten, wenn sie sich

1 http://www.uml.org/

denn überhaupt identifizieren lassen. Ein nicht zu unterschätzender Aufwand für das Extrahieren dieser Information ist die Folge. Softwarevisualisierung kann diese Aufgabe unterstützen, indem weitestgehend automatisiert, zeit- und kosteneffektiv Informationen über Struktur und Verhalten erzeugt und dem Betrachter in aufbereiteter Form vermittelt werden. Dabei können auch kritische Stellen, wie etwa besonders komplexer Code, aufgedeckt werden.

Softwarevisualisierung kann aber auch eingesetzt werden, um das Management über den Stand und den Fortschritt eines Softwareprojektes zu informieren. Dazu werden neben den statischen und dynamischen Informationen aus dem Quellcode die historischen Informationen der Entwicklung betrachtet und miteinander verknüpft. Entsprechende Visualisierungstechniken und -methoden ermöglichen ein Monitoring des Projekts und können als Unterstützung für Entscheidungen herangezogen werden.

Losgelöst von diesen drei eher praktischen Anwendungsgebieten ist der Einsatz für die Lehre. Insbesondere in den Anfängen der Softwarevisualisierung wurden Visualisierungen von Algorithmen oder Datenflüssen eingesetzt, um die teilweise komplexen und umfangreichen Zusammenhänge besser erklären zu können (Maletic et al. 2002).

2.4 Visualisierungspipeline

Die Visualisierungspipeline ist der Prozess, der aus der Datengrundlage die visuelle Repräsentation erzeugt. Haber & McNabb haben die elementare Variante dieses Prozesses definiert (Haber & McNabb 1990). Sie besteht aus den aufeinander folgenden Schritten Datenaufbereitung (engl. *filtering*), Abbildung auf ein geometrisches Modell (engl. *mapping*) und dem Generieren des Bildes für den Betrachter (engl. *rendering*).

DosSantos & Brodlie ergänzen diese konventionelle Visualisierungspipeline um den vorgelagerten Schritt der Datenanalyse (engl. *data analysis*) (dos Santos & Brodlie 2004). Durch diese Analyse lassen sich multivariate und multidimensionale Aufgabenstellungen besser handhaben, da die Datenaufbereitung und die Datenauswahl für die Visualisierung getrennt voneinander betrachtet werden.

Eine weitere Ergänzungsmöglichkeit entsteht durch die Interaktion des Betrachters mit der Repräsentation. Ist der Prozess in seiner ursprünglichen Variante eine geradlinige Abfolge von Schritten, so kann nun der Betrachter Einfluss auf die Schritte *filtering* und *mapping* nehmen. Dieser Einfluss auf die Visualisierung wird als *computational steering* (Mulder et al. 1999) beziehungsweise *visual steering* (Johnson et al. 1999) bezeichnet. Durch die Interaktion sollen die Expressivität und die Effektivität der Visualisierung gesteigert werden, indem der Betrachter die Repräsentation individuell auf seine Bedürfnisse

ausrichten kann. Ziel ist es, dass die Individualität des Betrachters einen möglichst geringen Einfluss auf die Qualität der Softwarevisualisierung hat. Die entsprechend adaptierte und im Folgenden verwendete Variante der Visualisierungspipeline wird in Abbildung 2 dargestellt.

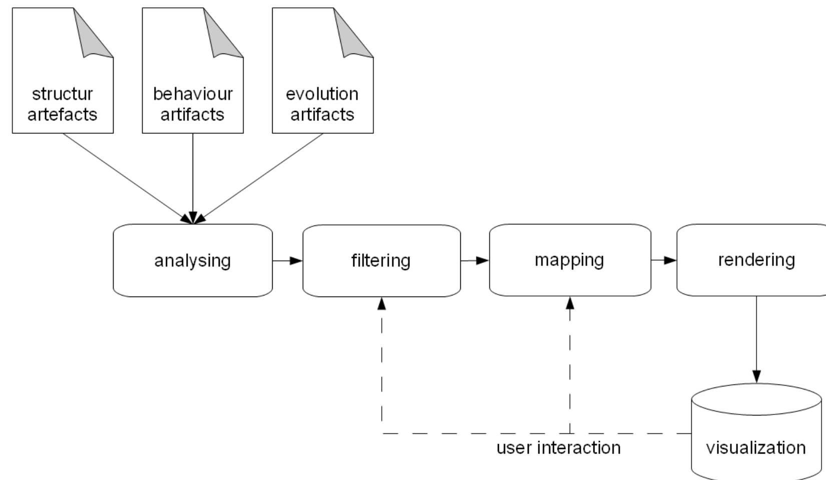

Abbildung 2: Visualisierungspipeline
(in Anlehnung an dos Santos & Brodlie 2004)

Im ersten Schritt, *analysing,* werden die Artefakte mit ihren statischen, dynamischen und historischen Informationen analysiert und zusammengeführt. Ziel ist es, eine konsistente Datengrundlage für die folgenden Schritte der Pipeline zu schaffen. Dafür können fehlende Daten durch Interpolation ergänzt, fehlerhafte Daten korrigiert beziehungsweise ausgeschlossen oder Daten in ein anderes Format transformiert werden. Insbesondere bei großen Datenmengen kann auch eine Reduzierung der Daten, zum Beispiel durch Glättung, sinnvoll sein (Schumann & W. Müller 1999). Dieser erste Schritt ist eine rechnergestützte Daten-zu-Daten-Transformation ohne Einflussmöglichkeiten durch Interaktion (dos Santos & Brodlie 2004). Allerdings können verschiedene Parameter vom Benutzer konfiguriert werden.

Ist eine konsistente Datengrundlage geschaffen, können im nächsten Schritt, dem *filtering,* die interessierenden Teilmengen entlang verschiedener Kriterien und Dimensionen selektiert werden. Dank der Interaktionsmöglichkeit kann der Betrachter die zu visualisierenden Daten selbst bestimmen und somit seinem individuellen Wissensstand und aktuellen Bedürfnissen anpassen. Die Effektivität der Visualisierung kann dadurch erheblich gesteigert werden.

Nachdem die zu visualisierende Datenmenge bestimmt wurde, erfolgt im nächsten Schritt, dem *mapping,* die Abbildung der Daten auf geometrische Primitive, wie zum Beispiel Bildpunkte, Kurven oder geometrische Figuren, inklusive ihrer Attribute, wie Farbe, Größe oder Anordnung. Diese Daten-zu-Geometrie-Abbildung ist das Herzstück der gesamten

Visualisierungspipeline, da die Expressivität und die Effektivität durch die Wahl der geometrischen Primitive und ihrer zugehörigen Attribute in besonders großem Maße beeinflusst wird (Schumann & W. Müller 1999). Durch die Interaktion kann der Betrachter Einfluss auf die Abbildung der Daten nehmen, womit seinen individuellen visuellen Vorlieben, Fähigkeiten und Bedürfnissen entsprochen werden kann.

Der letzte Schritt in der Visualisierungspipeline ist das *rendering*. Dabei werden die geometrischen Daten in Bilder entsprechend dem Medium der Visualisierung umgewandelt. Da dieser Vorgang stark vom jeweiligen Medium abhängig ist und überwiegend automatisiert durch ein entsprechendes Grafik-Framework erledigt wird, finden in dieser Arbeit keine weiteren Ausführungen zu diesem Schritt statt.

2.5 Visualisierungstechniken

In Kapitel 2.4 wurde das *mapping* als das Herzstück des Visualisierungsprozesses mit kritischem Einfluss auf die Expressivität und die Effektivität identifiziert. Die zentrale Problemstellung ist die Wahl der Darstellungsform, der graphischen Elemente sowie deren Attribute (Schumann & W. Müller 1999). Die Komplexität des Problems zeigt sich in der Anzahl der Kombinationsmöglichkeiten. Es muss für jede Kombination separat untersucht werden, wie sie sich auf die Qualität der Visualisierung auswirkt. Aus diesem Grund sollen in dieser Arbeit nur zwei grundlegende Techniken, nämlich Graphen und Metaphern, vorgestellt werden. Für eine detaillierte Betrachtungen des *mappings* wird an dieser Stelle auf die Ausführungen von (Schumann & W. Müller 1999) verwiesen.

2.5.1 Graphen

Graphen sind, nicht nur in der Informatik, eine weit verbreite Form zur Visualisierung von Informationen. So lassen sich viele Sachverhalte, wie beispielsweise Landkarten, Stücklisten oder Schaltkreise, mittels Graphen expressiv und effektiv visualisieren. In der Softwareentwicklung werden sie in vielen Diagrammen, wie dem Klassendiagramm, dem Datenflussdiagramm, dem Zustandsdiagramm oder dem Entity-Relationship-Diagramm, eingesetzt. Nach der Graphentheorie besteht ein Graph aus zwei Mengen von Knoten und Kanten (Diestel 2000). Die Knoten bilden die Grundlage des Graphen und können über Kanten miteinander verbunden sein. Darauf aufbauend lassen sich verschiedene Arten unterscheiden. Die bekannteste Art ist ein *Baum*, bei dem es sich um einen zusammenhängenden, azyklischen Graphen handelt. In einem Baum ist also jeder Knoten mit mindestens einem anderen über eine Kante verbunden. Des Weiteren existiert immer nur ein Weg von einem Knoten zum anderen, so dass kein Zyklus entstehen kann.

Wird den Kanten des Baums eine Richtung zugewiesen, trägt er die Bezeichnung *gewurzelter Baum* oder *Wurzelbaum*, da immer ein Knoten als Wurzel identifiziert werden kann.

Die Graphentheorie macht keine Vorschrift darüber, wie ein Graph dargestellt werden muss (Diestel 2000) .Die Visualisierung muss lediglich über geometrische Elemente für die beiden Mengen Knoten und Kanten verfügen. Üblicherweise wird eine zweidimensionale Darstellung mit Punkten oder Kreisen für die Knoten und geraden, durchgezogenen Linien für die Kanten verwendet.

Für das Zeichnen eines Graphen existieren verschiedene Verfahren, die von den Eigenschaften des Graphen und den gestellten Anforderungen, wie zum Beispiel Ästhetik oder Platzverbrauch, abhängen. Im Folgenden werden Verfahren für ungerichtete Graphen, Wurzelbäume und gemischte Graphen vorgestellt.

Verfahren für ungerichtete Graphen

Für ungerichtete Graphen eignen sich kräftebasierte (engl. *force-directed*) Verfahren (Battista 1999). Der Ansatz besteht darin, dass auf jeden Knoten anziehende oder abstoßende Kräfte einwirken. Durch die Ermittlung der Gesamtkraft auf die einzelnen Knoten kann deren Anordnung bestimmt werden. Welche Kräfte wie auf die Knoten einwirken, ist von der konkreten Implementierung abhängig. Zu den bekanntesten Vertretern zählen *spring-embedder* (Eades 1984), der *Kamada-Kawai*-Algorithmus (Kamada & Kawai 1989) und das *force-directed placement* (Fruchterman & Reingold 1990).

Verfahren für Wurzelbäume

Für Bäume, insbesondere Wurzelbäume, sind hingegen hierarchische (engl. *hierarchical*) Verfahren besser geeignet (Battista 1999). Die grundlegende Idee zur Visualisierung geht auf Sugiyama et al. zurück und gliedert sich in die folgenden drei Schritte (Sugiyama et al. 1981).

Zuerst werden alle Knoten einer Ebene (engl. *layer*) zugeordnet. Dies geschieht nach ihrer Rangordnung in der Hierarchie. Alle Knoten einer Ebene erhalten damit die gleiche Positionierung in einer Dimension, beispielsweise der y-Dimension. Im darauf folgenden Schritt werden die Knoten einer Ebene so angeordnet, dass möglichst wenige Überschneidungen der Kanten entstehen. Der dritte Schritt ordnet jedem Knoten entsprechend seiner Ebene die finale Position zu. Dies kann nach unterschiedlichen Kriterien, wie beispielsweise Symmetrie oder minimalem Platzverbrauch, geschehen.

Jeder der drei Schritte ist eine eigene Problemstellung mit mehreren Ansätzen zur Lösung. Ausführungen mit Beispielen zur Implementierung für *saubere* Bäume mit minimalem

Platzverbrauch und unter Einhaltung ästhetischer Grundsätze liefern Wetherell & Shannon (Wetherell & Shannon 1979) sowie Reingold & Tilford (Reingold & Tilford 1981).

Eine weiteres Verfahren für Wurzelbäume stellen die von Robertson et al. beschriebenen *cone trees* beziehungsweise *cam trees* dar (Robertson et al. 1991). Unter Einbeziehung der dritten Dimension ordnen sie alle Knoten einer Ebene als Kegel unterhalb des übergeordneten Knotens an. Auf diese Weise wird eine maximale Übersicht bei minimalem Platzverbrauch erreicht. Abbildung 3 zeigt ein Beispiel für einen *cone tree*.

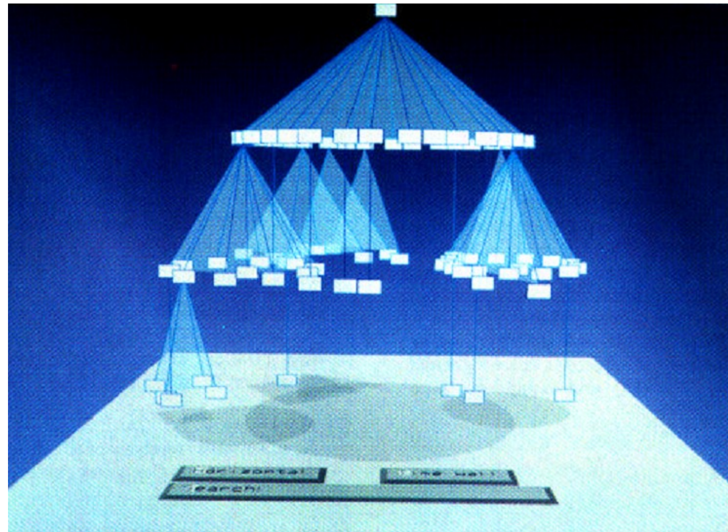

Abbildung 3: Cone-Tree
(Robertson et al. 1991)

Verfahren für gemischte Graphen

Als gemischte Graphen werden im Folgenden Graphen bezeichnet, die der Theorie nach keine Bäume sind, allerdings über eine Hierarchie verfügen. Es handelt sich also um zusammenhängende, gerichtete, zyklische Graphen. Für diese Art eignen sich geschachtelte Visualisierungen in zwei oder drei Dimensionen.

Eine Variante der Schachtelung in 2D stellt die sogenannte Wärmekarte (engl. *heatmap*) dar. Alle Knoten eines Graphen werden dabei als Rechtecke dargestellt, wobei die Rechtecke der untergeordnete Knoten in die Rechtecke der übergeordneten Knoten geschachtelt werden. Die Wärme wird durch einen Farbverlauf von blau (kalt) nach rot (warm) visualisiert. Somit können auf einen Blick interessante Stellen erkannt werden. Die Abbildung eines gemischten Graphen in einer Wärmekarte erfolgt zum einen über die beschriebene Schachtelung und zum anderen über zusätzliche Linien zwischen den Rechtecken für diejenigen Kanten, die nicht zur Hierarchie zählen. In Abbildung 4 ist eine Variante einer solchen zweidimensionalen Schachtelung abgebildet. Auf die Darstellung der Wärme wurde dagegen verzichtet, da die Färbung für das Zeichnen eines gemischten Graphen nicht relevant ist.

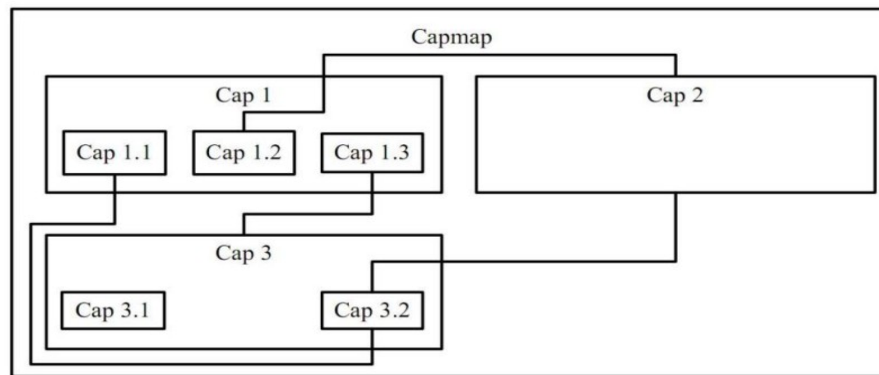

Abbildung 4: Modifizierte Heatmap ohne Färbung (Klinkmüller 2009, S.53)

Wie bereits Abbildung 4 andeutet, kann diese Variante nicht für jede beliebige Größenordnung eines Graphen sinnvoll sein. Abhilfe könnte das Ausweichen in die dritte Dimension leisten. Rekimoto und Green beschreiben dazu einen *information cube* (Rekimoto & Green 1993) wie in Abbildung 5 dargestellt, der dreidimensionale Würfel anstelle von Rechtecken schachtelt. Parallel zur Wärmekarte wird auch hier die Hierarchie durch die Schachtelung der untergeordneten Knoten in die übergeordneten Knoten abgebildet. Zusätzliche Kanten können ebenfalls als Linien zwischen den Würfeln eingezeichnet werden.

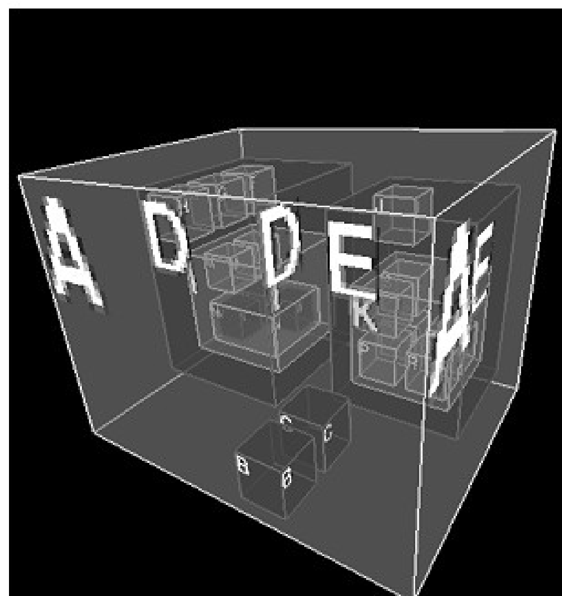

Abbildung 5: Information cube (Rekimoto & Green 1993)

Einen weiteren interessanten Ansatz für die Visualisierung gemischter Graphen in 3D liefert (Klinkmüller 2009). Er bildet die Hierarchie als Wurzelbaum auf der Oberfläche einer Kugel ab. Zusätzliche Kanten verlaufen dann im Inneren der Kugel zwischen den Knoten. Abbildung 6 zeigt ein Beispiel dieser Visualisierungstechnik aus zwei verschiedenen Perspektiven. Dabei kennzeichnen die roten Linien die Kanten der Hierarchie, während die grünen Linien die zusätzlichen Kanten abbilden.

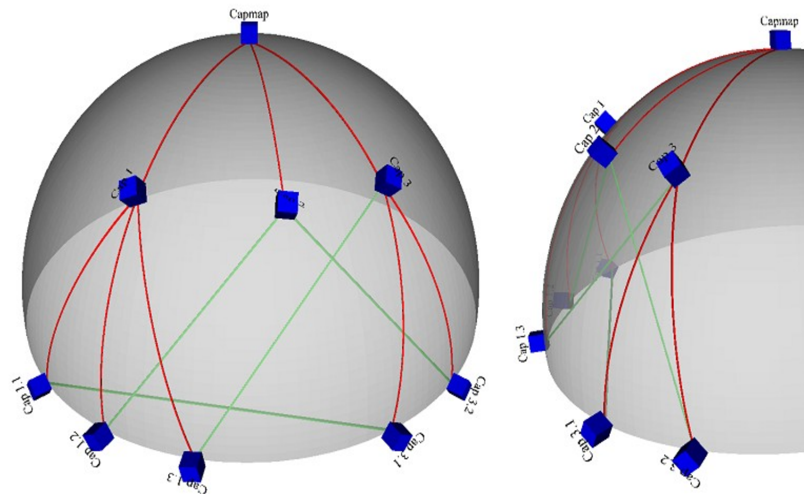

Abbildung 6: Kugelbaum
(Klinkmüller 2009, S.56)

2.5.2 Visuelle Metaphern

Eine weit verbreitete Definition des Metaphern-Begriffs stammt von Lakoff und Johnson: „The essence of metaphor is understanding and experiencing one kind of thing in terms of another.“ (Lakoff & M. Johnson 1980) Übertragen auf die Visualisierung soll es eine visuelle Metapher dem Betrachter ermöglichen, den zu Grunde liegenden Sachverhalt einfacher zu verstehen, indem ihm bereits vertraute Konzepte genutzt werden. Die bekanntesten Vertreter lassen sich in jedem Betriebssystem wiederfinden. So symbolisiert eine Lupe beispielsweise die Suchfunktion, ein Fragezeichen steht für Hilfe und eine Sanduhr steht für die Wartezeit, bis eine Aktion ausgeführt wurde. Entsprechend den verwendeten Elementen und Konzepten können prinzipiell abstrakte und natürliche visuelle Metaphern unterschieden werden (Gračanin et al. 2005). Abstrakte visuelle Metaphern setzen sich aus einfachen geometrischen Figuren ohne direkten Bezug zur Realität zusammen. Die beiden Abbildungen auf der folgenden Seite zeigen zwei Beispiele für abstrakte visuelle Metaphern.

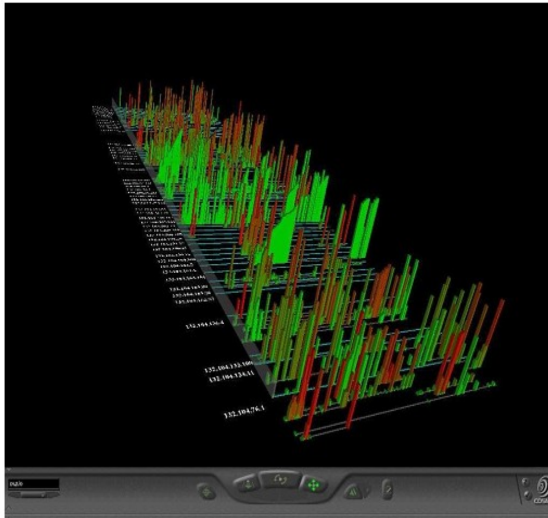

Abbildung 7: Information-Landscape-Metapher (Santos et al. 200)

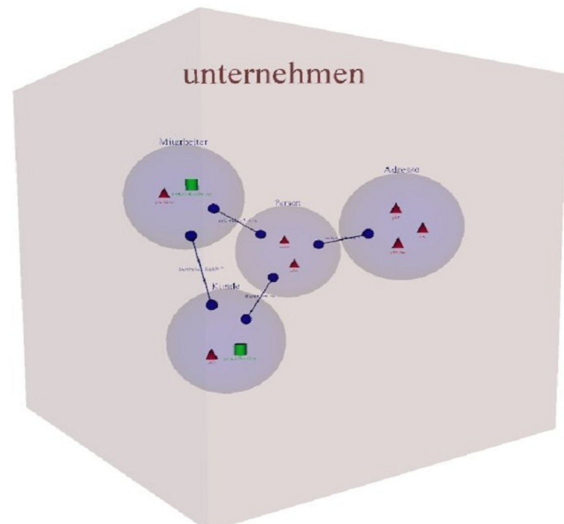

Abbildung 8: Abstrakte Graphen-Metapher (R. Müller 2009)

Abbildung 7 zeigt das Beispiel einer Information-Landscape-Metapher zur Visualisierung von Netzwerkdaten (Santos et al. 2000). Graphen können ebenfalls als abstrakte Metaphern aufgefasst werden, wie Abbildung 8 mit der Visualisierung der Strukturinformationen von Software zeigt (R. Müller 2009).

Eine natürliche Metapher bedient sich hingegen Formen, die aus der realen Welt bekannt sind. Diese Art der Metapher dürfte für die meisten Betrachter die größere Erleichterung für das Verständnis erzielen. Dabei muss allerdings beachtet werden, dass die Komplexität der visuellen Metapher nicht die Expressivität der Visualisierung beeinträchtigen darf. Es sollte nicht möglich sein, Informationen entsprechend der realen Welt zu interpretieren, die nicht in der Datengrundlage enthalten sind. Abbildung 9 und Abbildung 10 zeigen zwei Varianten der City-Metapher nach (Santos et al. 2000) und (Panas et al 2007).

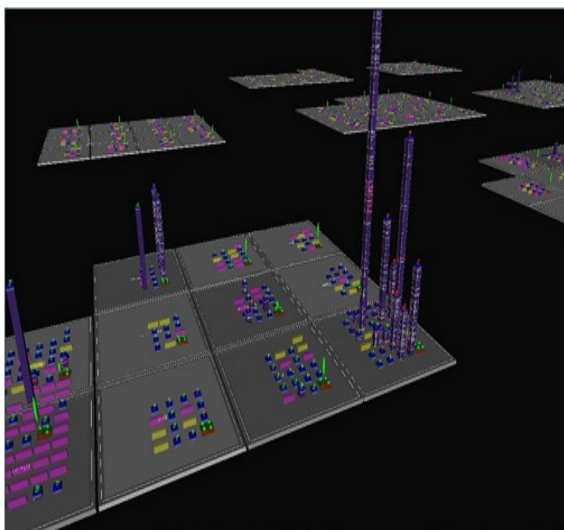

Abbildung 9: City-Metapher: Variante 1 (Santos et al. 2000)

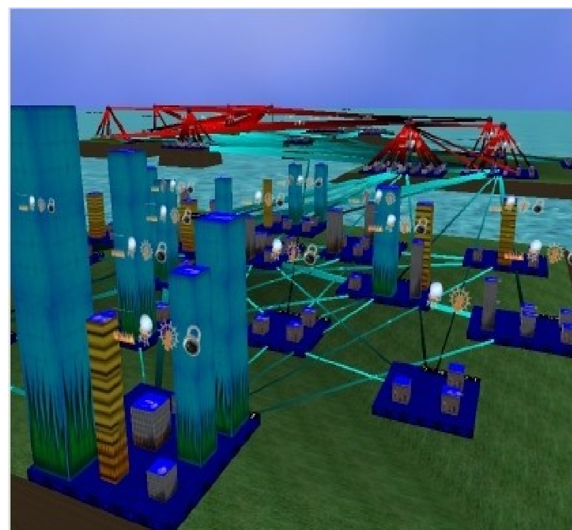

Abbildung 10: City-Metapher: Variante 2 (Panas et al. 2007)

Nach Gračanin et al. müssen beim Einsatz einer visuellen Metapher zwei Kriterien gewährleistet sein, um die Effektivität und Expressivität der Visualisierung nicht zu gefährden (Gračanin et al. 2005):

- Die Konsistenz verlangt, dass eine Metapher konsistent für die gesamte Visualisierung eingesetzt werden muss. Verschiedene Arten von Artefakten dürfen nicht auf die gleichen Elemente der Metapher abgebildet werden und eine Art von Artefakten darf nicht durch mehrere Elemente der Metapher repräsentiert werden.
- Die semantische Reichhaltigkeit (engl. *semantic richness*) besagt, dass die gewählte Metapher genügend Elemente enthalten muss, um alle Aspekte der Software, die visualisiert werden sollen, abbilden zu können. Gleichzeitig muss aber auf den Umfang und die Komplexität der Metapher geachtet werden, da ansonsten der Nutzen beeinträchtigt wird oder es zu Fehlinterpretationen kommen kann.

2.6 Taxonomie für Softwarevisualisierungen

In den bisherigen Ausführungen wurde bereits deutlich gemacht, dass Softwarevisualisierung ein komplexes und umfangreiches Themengebiet mit einer Vielzahl von Gestaltungsmöglichkeiten ist. In diesem Abschnitt soll nun eine Taxonomie zur Unterscheidung der verschiedenen Softwarevisualisierungen entwickelt werden.

In der frühen Literatur existieren zwei Taxonomien, nämlich (Roman & Cox 1993) und (Price et al. 1993), mit zum Teil deutlichen Redundanzen. Maletic et al. stellen diese gegenüber, kombinieren sie in einer eigenen Taxonomie und ergänzen sie um weitere Punkte (Maletic et al. 2002). Tabelle 2 zeigt das Ergebnis.

Maletic et al.	Roman & Cox	Price et al.
Aufgabe (engl. *task*)	-	Zweck
Zielgruppe (engl. *audience*)	-	Zweck
Ausrichtung (engl. *target*)	Bereich Abstraktionsgrad	Bereich Inhalt
Darstellung (engl. *representation*)	Spezifikationsmethode Benutzungsschnittstelle Präsentation	Form Methode Interaktion Effektivität
Medium	-	Form

Tabelle 2: Gegenüberstellung der Taxonomien von Maletic et al. , Roman & Cox sowie Price et al. (Maletic et al. 2002)

Wie in der Tabelle 2 gezeigt, klassifizieren Maletic et al. Softwarevisualisierung entlang der fünf Dimensionen Aufgabe, Zielgruppe, Ausrichtung, Darstellung und Medium.

Die Einordnung nach der Aufgabe erfolgt gemäß den in Kapitel 2.3 vorgestellten Aufgabengebieten Entwicklung neuer Softwaresysteme, Wartung und Integration bestehender Systeme, Management von Softwareprojekten sowie der allgemeinen Lehre der Funktionsweise von Software.

Maletic et al. unterscheiden anhand der Aufgabendimension die Zielgruppen der Visualisierung in Bildung und Industrie. Ersterer werden Personen der Lehre und Forschung zugeordnet, die ein allgemeines Verständnis über Software erlernen oder vermitteln wollen. Zur zweiten Gruppe werden Personen im Kontext von Softwareprojekten, wie beispielsweise Entwickler, Qualitätssicherer oder Teammanager, gezählt. Zudem können die Zielgruppen nach verschiedenen Kriterien weiter unterteilt werden. Beispielsweise nach dem Stand der Fähigkeiten, Anfänger versus Experte oder Entwickler versus Manager oder nach der benötigten Einarbeitungszeit in die Visualisierungstechnik.

Die Ausrichtung gibt den zu visualisierenden Aspekt der Software und die zu Grunde liegenden Artefakte an. Wie in Kapitel 2.2 dargelegt, wird nach den Aspekten Struktur, Verhalten und Evolution unterschieden. Diese werden wiederum durch Artefakte mit statischen, dynamischen und historischen Informationen beschrieben.

Mit der Darstellung wird klassifiziert, wie die zu Grunde liegenden Daten abgebildet werden. Diese Dimension hängt von allen anderen Dimensionen einschließlich dem Medium ab. Der Prozess der Abbildung der abstrakten Datengrundlage auf eine visuelle geometrische Repräsentation wurde in Kapitel 2.4 bereits näher beschrieben und in Kapitel 2.5 wurden grundlegende Techniken dafür vorgestellt. Die Dimension der Darstellung erfasst auch die Interaktion mit einer Softwarevisualisierung, da die Möglichkeiten zur Interaktion stark von der jeweiligen Repräsentation abhängig sind.

Mit dem Medium wird die Grundlage zur Darstellung charakterisiert. Die Unterteilung reicht von Zettel und Stift über verschiedene elektronische Anzeigegeräte bis hin zu immersiven virtuellen Realitäten. Die Wahl des Mediums sollte an die konkret gestellten Anforderungen der Visualisierung durch die bisher vorgestellten Dimensionen der Aufgabe, der Zielgruppe sowie der Ausrichtung gekoppelt sein.

Neben diesen fünf von Maletic et al. definierten Dimensionen, stellen Stasko & Patterson eine weitere relevante Eigenschaft einer Softwarevisualisierung vor, nämlich den Automatisierungsgrad (Stasko & Patterson 1991). Dieser beschreibt das Verhältnis zwischen maschinellem und manuellem Aufwand beim Erstellen und Ausführen einer

Visualisierung. Insbesondere für die Anforderung an die Angemessenheit ist dies eine kritische Eigenschaft (vgl. Kapitel 2.1.2).

Mit Hilfe dieser sechs vorgestellten Klassifizierungskriterien – Aufgabe, Zielgruppe, Ausrichtung, Darstellung, Medium und Automatisierungsgrad – werden in der vorliegenden Arbeit Softwarevisualisierungen unterschieden. Damit sind die theoretischen Ausführungen zur Softwarevisualisierung abgeschlossen, womit im nächsten Kapitel die Interaktion mit einer Softwarevisualisierung ausführlich behandelt werden kann.

3 Interaktion mit einer Softwarevisualisierung

In diesem Kapitel erfolgt die theoretische Betrachtung der Interaktion mit einer Softwarevisualisierung. Zunächst wird dazu im ersten Unterkapitel die Interaktion im Kontext der Softwarevisualisierung eingeordnet und deren Bedeutung herausgestellt. Im zweiten Unterkapitel folgt dann die Definition, während das dritte Unterkapitel die Ziele vorstellt, die der Benutzer bei der Interaktion mit einer Softwarevisualisierung verfolgt. Die Benutzungsschnittstelle als zentrales Konzept der Interaktion zwischen System und Benutzer wird im vierten Unterkapitel behandelt. Die Beschreibung des Aufbaus erfolgt anschließend in Kapitel 3.5. Das sechste Unterkapitel stellt die Interaktionstechnik als ein weiteres Konzept der Interaktion vor, welche die Art und Weise der Verwendung einer Benutzungsschnittstelle zu einem bestimmten Zweck beschreibt. Für einen besseren Überblick über die verschiedenen Interaktionstechniken wird im darauf folgenden Kapitel 3.7 eine Taxonomie zur Klassifizierung eingeführt. Im letzten Unterkapitel wird aufbauend auf den Erkenntnissen der vorangegangen Kapitel ein Konzept zur Interaktion mit einer Softwarevisualisierung erarbeitet.

3.1 Einordnung

Bereits aus der Visualisierungspipeline (Kapitel 2.4) geht hervor, dass eine Softwarevisualisierung zwei Aspekte aufweist, nämlich Repräsentation und Interaktion. Die Repräsentation, deren Basis die Computergrafik ist, enthält die Überführung der zugrunde liegenden Daten in eine geometrische Form und die Darstellung dieser auf einem Ausgabegerät. Die Interaktion dagegen realisiert die Kommunikation zwischen Softwarevisualisierung und Benutzer. Auf diese Weise kann der Benutzer die Repräsentation seinen Anforderungen und Bedürfnissen anpassen. Die Grundlagen dafür liefert das Forschungsfeld der Mensch-Computer-Interaktion (MCI).
Die Aspekte Repräsentation und Interaktion können nicht völlig getrennt voneinander betrachtet werden. So schafft die Repräsentation erst die Voraussetzungen für die Interaktion, während diese wiederum Einfluss auf die Repräsentation nimmt. Eine isolierte Betrachtung der Repräsentation ohne Interaktion ist auch deshalb kaum möglich, da ein Benutzer selbst mit einer statischen Abbildung interagieren kann, was bereits in Kapitel 1.1 beschrieben wurde.
Obwohl beide Aspekte somit entscheidenden Einfluss auf die Qualität einer Softwarevisualisierung besitzen, beschäftigt sich ein Großteil der Forschung zur Visualisierung ausschließlich mit der Repräsentation. Dabei sind einer Softwarevisualisierung ohne Interaktionsmöglichkeiten schnell Grenzen gesetzt. Das

Dilemma besteht in der begrenzten Menge an Informationen, die übersichtlich in einer Softwarevisualisierung dargestellt sein kann. Selbst bei einer guten Darstellungsform wird sie ab einer gewissen Menge unübersichtlich und die Erkenntniserlangung des Betrachters wird erschwert oder gar verhindert. Bei einer geringen Menge enthaltener Informationen kann der Betrachter allerdings auch nur eine begrenzte Menge von Erkenntnissen erlangen. Die Möglichkeit, mehrere Visualisierungen für verschiedene Aspekte zu verwenden, kann dieses Problem nur teilweise lösen, da durch die Aufteilung wiederum der Gesamtüberblick verloren geht und somit grundlegende Beziehungen nur schwer nachvollzogen werden können. Die Interaktion dagegen erlaubt es, dieses Dilemma zu durchbrechen, indem sie das Platzproblem durch eine Veränderung der Repräsentation im Zeitverlauf auflöst. In einer interaktiven Softwarevisualisierung kann somit eine erheblich größere Menge von Information enthalten sein, die andernfalls nur in einer sehr großen oder in vielen kleinen statischen Repräsentationen untergebracht werden könnte.

Eine interaktive Softwarevisualisierung ermöglicht dem Benutzer zudem eine viel intensivere Erforschung der Repräsentation, als dies mit einer statischen Abbildung überhaupt möglich wäre. Durch die direkte und gezielte Manipulation der Repräsentation können insbesondere Zusammenhänge deutlich besser erfasst werden. Ein Ansatzpunkt zur Untersuchung oder eine Idee lassen sich dadurch viel genauer verfolgen und bringen somit eher Erkenntnisse hervor, als dies durch einfaches Betrachten der Fall wäre.

3.2 Definition

Eine präzise und aussagekräftige Definition für die Interaktion mit einer Softwarevisualisierung zu formulieren ist keine leichte Aufgabe, da sie zum einen von der Repräsentation abhängig ist und zum anderen kein direkt greifbares Konzept darstellt. Die MCI definiert Interaktion allgemein als *die Kommunikation zwischen einem Benutzer und einem interaktiven System in Form eines Dialogs* (Dix et al. 2004). Das Ziel dieses Dialogs ist es, dass der Benutzer den Ablauf des Programms entsprechend einer Aufgabenstellung beeinflussen kann. Abbildung 11 zeigt die Kommunikation zwischen Benutzer und System im Kontext der Aufgabe.

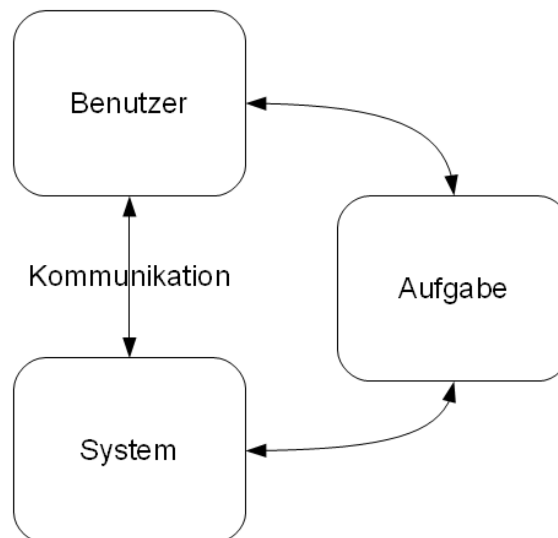

Abbildung 11: Interaktion: Überblick

Übertragen auf die Interaktion mit einer Softwarevisualisierung führt der Benutzer also einen Dialog mit dem Ziel, die Repräsentation der Visualisierung so zu verändern, dass er ein möglichst gutes Verständnis der zugrunde liegenden Software entwickeln kann. Als Besonderheit gegenüber anderen Systemen dienen die Eingaben des Benutzers dabei in erster Linie der Änderung der Ausgabe und nicht, um neue Daten in das System einzugeben.

3.3 Ziele

Ausgehend von dem allgemeinen Ziel der Softwarevisualisierung – Verständnis von Aufbau, Verhalten und Entwicklung der Software zu erlangen – lassen sich fünf Ziele speziell für die Interaktion mit einer Visualisierung der Struktur von Software identifizieren, die in Tabelle 3 mit ihrer Bezeichnung und einer Kurzbeschreibung aufgeführt sind.

Ziel	Beschreibung
Überblick & Ausschnitt	Betrachten der Struktur als Gesamtheit oder in Ausschnitten
Beziehungen erfassen	Erfassen von Beziehungen zwischen Elementen
Grad der Details ändern	Ändern der Detailmenge in der Repräsentation
Elemente vergleichen	Vergleichen von Elemente nach verschiedenen Kriterien
Elemente identifizieren	Identifizieren von Elementen nach verschiedenen Kriterien

Tabelle 3: Ziele Interaktion mit Softwarevisualisierung der Struktur

Diese Ziele wurden unter Berücksichtigung ihrer Bedeutung für den Prozess des Verstehens der Struktur von Software extrahiert. Sie sind nicht vollständig diskret, so dass beim Verfolgen eines Ziels gleichzeitig ein anderes erreicht werden kann. Im Folgenden wird jedes Ziel in einem eigenen Absatz vorgestellt.

Überblick & Ausschnitt

Die Betrachtung von Ausschnitten ist eine wichtige Grundlage für die Entwicklung von Verständnis, da die Informationsmenge, die ein Benutzer gleichzeitig wahrnehmen und verarbeiten kann, begrenzt ist. Somit genügt es nicht, einfach nur alle Bestandteile gleichzeitig in einer Repräsentation zu betrachten. Sondern es muss auch ein Überblick über die Gesamtheit der Struktur möglich sein, um den Zusammenhang der einzelnen Ausschnitte nachvollziehen zu können. Mit der Interaktion wird genau dieses Ziel verfolgt. Der Benutzer kann beliebige Ausschnitte aus der visualisierten Datengrundlage auswählen, um sie unter verschiedenen Gesichtspunkten näher zu betrachten. Dabei hat er gleichzeitig die Möglichkeit, die einzelnen Ausschnitte in den Kontext der gesamten Struktur einzuordnen, da er zu jeder Zeit einen Überblick über das gesamte System erhalten kann. Ein Ansatzpunkt für eine solche Unterteilung bietet die Paketstruktur, bei der Pakete oder Klassen als Ausschnitte betrachtet werden können.

Beziehungen erfassen

Einen Großteil der Struktur der Software stellen die zum Teil komplexen Beziehungen zwischen den Elementen dar. Das Erfassen dieser ist somit ein grundlegender Bestandteil für das Verständnis des gesamten Aufbaus. Beziehungen lassen sich nach verschiedenen Arten, wie Vererbung oder Assoziation, unterscheiden. Alle Beziehungsarten gleichzeitig in einer Repräsentation darzustellen führt jedoch bereits bei kleinen Softwaresystemen schnell zur Unübersichtlichkeit. Durch das Interagieren mit der Repräsentation soll der Benutzer dagegen selbst bestimmen können, welche Beziehungsarten dargestellt werden und unter welchen Umständen dies geschieht. Dazu wählt er entweder die Beziehungsarten aus oder bestimmt direkt die Elemente, deren Beziehungen angezeigt werden sollen. Mit Interaktion kann der Benutzer auch Beziehungen erfassen, die keine direkte visuelle Darstellung, wie zum Beispiel eine Linie, besitzen. Die mit einem Element verbundenen Elemente werden dann zum Beispiel visuell hervorgehoben oder räumlich um das Ausgangselement angeordnet. Auf diese Weise sind der Verarbeitung von Beziehungen keine Grenzen mehr durch eine konkrete visuelle Repräsentation gesetzt.

Grad der Details ändern

Auch ein Ausschnitt kann noch zu viele Informationen für die Verarbeitung enthalten, so dass dieser weiter verkleinert werden müsste. Aus verschiedenen Gründen ist dies aber nicht immer möglich oder sinnvoll, zum Beispiel wenn ein ganzes Paket im Fokus des Interesses steht. Um dennoch den Überblick zu behalten und weiterhin gezielt verschiedene Aspekte untersuchen zu können, muss demnach die Anzahl der Details reduziert werden. Das Ausblenden verschiedener Elemente durch Interaktion kann die

Konzentration auf bestimmte Aspekte unterstützen. Sollen nur Klassen und ihre Beziehungen betrachtet werden, kann auf die Darstellung von Attributen, Methoden und eventuell sogar Paketen verzichtet werden. Oftmals werden aber in einem Ausschnitt auch zusätzliche Informationen benötigt, um einen Aspekt näher untersuchen zu können, wie zum Beispiel die Verteilung der LOC auf die einzelnen Methoden einer Klasse. Das Einblenden weiterer Details, wie Metriken oder aber auch Beziehungen, ermöglicht es dem Benutzer, die Datengrundlage individuell nach seinen Bedürfnissen und den jeweiligen Anforderungen zu erforschen.

Elemente vergleichen

Obwohl ein Benutzer bereits in einer statischen Repräsentation verschiedene Elemente miteinander vergleichen kann, sind ihm dabei gewisse Grenzen gesetzt. Zum einen können nicht alle Informationen für jede Vergleichsmöglichkeit in der Repräsentation enthalten sein, da dies schnell zur Unübersichtlichkeit führen würde. Zum anderen lassen sich die enthaltenen Informationen nicht immer exakt aus der Repräsentation extrahieren, da sie beispielsweise nur implizit in der Darstellung eines Elements kodiert sind. Darüber hinaus erschwert eine größere räumliche Entfernung das Vergleichen, selbst wenn alle relevanten Informationen vorliegen. Das Ziel der Interaktion ist es demnach, dass der Benutzer beliebige Elemente nach verschiedenen Kriterien miteinander vergleichen kann. Dafür kann er sich die benötigten Informationen nach Bedarf zusammenstellen und die Anordnung der Elemente so verändern, dass ein direkter Vergleich durch einfaches Betrachten ermöglicht wird.

Elemente identifizieren

Das Identifizieren von Elementen in einer statischen Repräsentation kann ein aufwändiges Unterfangen sein. Interaktion ermöglicht es dem Benutzer, jedes Element mit vergleichsweise einfachen Mitteln schnell und eindeutig zu identifizieren. Durch das Identifizieren einer Menge von Elementen anhand bestimmter Kriterien können auch verschiedene Muster und Konzepte der Struktur erkannt werden. Das Identifizieren aller get- und set-Methoden beispielsweise lässt einen ersten Rückschluss auf die Aufgabe einer Klasse zu. Auch Aspekte der Architektur können ein Kriterium zur Identifikation darstellen, wie beispielsweise die Zugehörigkeit zu einer der drei Schichten Frontend, Logik oder Backend. Dabei zeigt sich das Potential der Interaktion, auch die Ideen und Konzepte, die hinter der eigentlichen Implementierung liegen, zu identifizieren.

3.4 Benutzungsschnittstelle

Die Kommunikation zwischen Benutzer und System erfolgt über die Benutzungsschnittstelle des interaktiven Systems. Eine Benutzungsschnittstelle, auch Benutzerschnittstelle oder Benutzeroberfläche, ist nach DIN EN ISO 9241-110 wie folgt definiert:

> *„Alle Bestandteile eines interaktiven Systems (Software oder Hardware), die Informationen und Steuerelemente zur Verfügung stellen, die für den Benutzer notwendig sind, um eine bestimmte Arbeitsaufgabe mit dem interaktiven System zu erledigen."* (DIN EN ISO 9241-110)

Abbildung 12 stellt die Benutzungsschnittstelle entsprechend der Definition mit ihren Beziehungen zu System, Benutzer und Aufgabe dar.

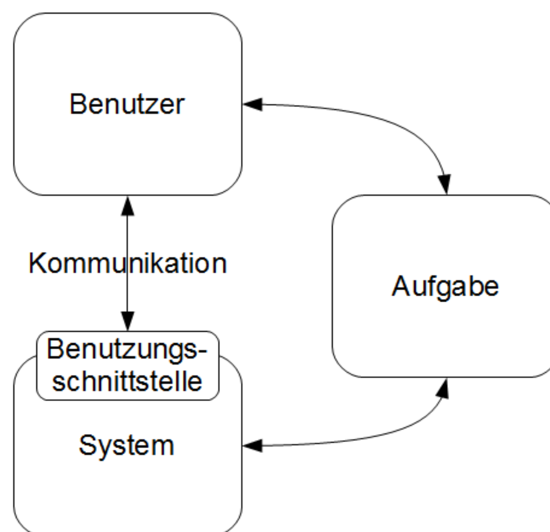

Abbildung 12: Benutzungsschnittstelle: Überblick

Wie aus der Abbildung hervorgeht, bilden die technischen Gegebenheiten des Systems die Ausgangslage für die Gestaltung der Benutzungsschnittstelle. Auf der anderen Seite muss der Benutzer als direkter Kommunikationspartner des Systems berücksichtigt werden. Zudem besitzt auch die Aufgabe, die mit dem System realisiert wird, einen Einfluss auf die Gestaltung, da die Interaktion ja zum Zweck der Aufgabenerfüllung erfolgt. Im Folgenden werden nun diese drei Einflüsse auf die Gestaltung der Benutzungsschnittstelle kurz umrissen. Eine detailliertere Betrachtung erfolgt dagegen in (Heinecke 2004).

System

Die technischen Gestaltungsmöglichkeiten der Kommunikation werden durch die Ein- und Ausgabegeräte des Systems definiert. Im Verlauf der Zeit wurden, angefangen bei Schaltern und Lämpchen über Tastatur und Farbbildschirm bis hin zu immersiven virtuellen Realitäten, immer mächtigere Geräte für die Interaktion entwickelt. Konnten zu Beginn beispielsweise nur einfache Eingaben über Schalter oder Tastatur erfolgen, kann in

einer virtuellen Realität selbst die Bewegung des Kopfes als Eingabe dienen. Auch die Ausgabemöglichkeiten haben sich von einfachen Schwarzweißbildern bis zu komplexen dreidimensional wirkenden Farbbildern stetig weiterentwickelt. Zusätzlich zu visuellen Ausgabemöglichkeiten wurden auch Ausgaben für andere Sinne, wie zum Beispiel Hören oder Tasten, entwickelt. Für die im Kontext dieser Arbeit betrachteten Softwarevisualisierungen werden Maus und Tastatur als Eingabegeräte und ein Farbbildschirm zur Ausgabe verwendet.

Benutzer

Da der Benutzer der direkte Kommunikationspartner des Systems ist, spielt der Einfluss der menschlichen Kognition und Wahrnehmungspsychologie auf die Interaktion eine entscheidende Rolle bei der Gestaltung der Benutzungsschnittstelle. Heinecke vergleicht den Menschen mit einem informationsverarbeitenden System, wobei Sinnesorgane und zentrales Nervensystem die Hardware darstellen und die geistige Wahrnehmung und Verarbeitung die Software repräsentieren (Heinecke 2004). Als informationsverarbeitendes System steht der Mensch mit der Umwelt in Wechselwirkung, indem er Informationen aufnimmt (Wahrnehmung) und auf die Umwelt physisch einwirkt (Motorik). Bei einer Softwarevisualisierung liegt der Schwerpunkt allerdings auf der Wahrnehmung, während die Motorik sich auf das gezielte Ändern der gelieferten Informationen durch die Eingabegeräte beschränkt. Der Sehsinn ist dabei der wichtigsten Sinn der Wahrnehmung.

Entscheidend für die visuelle Wahrnehmung ist aber nicht nur die physiologische Beschaffenheit des Auges, sondern vielmehr die Verarbeitung der aufgenommenen Informationen. Durch Gewohnheiten oder den Wunsch, etwas Bestimmtes sehen zu wollen, sieht der Mensch oft nicht exakt das, was eigentlich vorhanden ist. Beispielsweise läuft die Strukturierung relativ gleichförmiger visueller Informationen nach bestimmten Regeln ab, die als Gestaltgesetze bezeichnet werden. Diese beschreiben, wie mittels Form, Farbe und Anordnung einer Menge von Objekten ein Zusammenhang suggeriert oder vermieden werden kann. Bei der Gestaltung einer Benutzungsschnittstelle sollten diese Erfahrungsregeln berücksichtigt werden, damit der Benutzer einerseits zusammengehörige Darstellungsinformationen auch als zusammenhängend wahrnimmt und andererseits keine Zusammenhänge erkennt, die nicht vorhanden sind. Durch das Gesetz der Nähe beispielsweise werden räumlich oder zeitlich benachbarte Objekte als zusammenhängend wahrgenommen. Den gleichen Effekt erzielt das Gesetz der Ähnlichkeit/Gleichheit, indem beispielsweise ähnliche/gleiche Farben oder Formen verwendet werden. Die beiden Gesetze können auch kombiniert werden, um den Effekt zu verstärken oder abzuschwächen.

Aufgabe

Das allgemeine Ziel der Softwarevisualisierung inklusive der Anwendungsfälle wurde bereits in Kapitel 2.3 ausführlich vorgestellt. Die speziellen Ziele dagegen, die der Benutzer bei der Interaktion mit einer Softwarevisualisierung der Struktur verfolgt, sind in Kapitel 3.3 identifiziert und beschrieben worden. Entsprechend dieser Ziele muss die Benutzungsschnittstelle ausschließlich die Anpassung der Repräsentation zur Erfüllung dieser Ziele realisieren. Es werden im Gegensatz zu anderen interaktiven Systemen keine neuen Daten in das System eingegeben, womit folglich auch keine Verarbeitung von Daten, außer für die Darstellung der Datengrundlage, stattfindet.

Nachdem nun die Einflüsse von System, Benutzer und Aufgabe auf die Benutzungsschnittstelle beschrieben wurden, kann im nächsten Kapitel der Aufbau der Schnittstelle untersucht werden.

3.5 Aufbau der Benutzungsschnittstelle

Für die detaillierte Betrachtung der Gestaltungsmöglichkeiten einer Benutzungsschnittstelle wird diese in Anwendungs-, Dialog- und Ein-/Ausgabeschnittstelle unterteilt (Heinecke 2004). In Abbildung 13 ist diese Unterteilung im Kontext der Definition einer Benutzungsschnittstelle dargestellt.

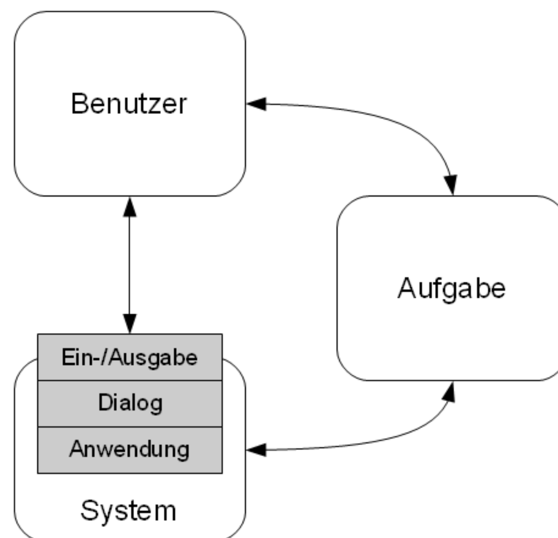

Abbildung 13: Benutzungsschnittstelle: Aufbau

Die Anwendungsschnittstelle, auch als Werkzeugschnittstelle bezeichnet, stellt die gesamte Funktionalität des Systems der Dialogschnittstelle zur Verfügung und setzt somit direkt auf dem System auf. Der Aufruf einer Funktion des Systems erfolgt dann über einen Dialog, der in der Dialogschnittstelle definiert wird. Die Ein- und Ausgaben des Dialogs werden über die Ein-/Ausgabeschnittstelle realisiert, die als einzige der drei Schnittstellen direkt mit dem Benutzer kommuniziert.

Das Ziel der Gestaltung einer Benutzungsschnittstelle ist es, eine ergonomische Schnittstelle für die Arbeit des Benutzers zu schaffen, die optimal auf seine Bedürfnisse und Fähigkeiten abgestimmt ist. Eine ausführliche Beschreibung der Gestaltungsmöglichkeiten der Benutzungsschnittstelle inklusive Beispielen gibt (Heinecke 2004). In den folgenden drei Unterkapiteln soll dagegen nur ein kurzer Überblick über die Möglichkeiten der Gestaltung der einzelnen Schnittstellen gegeben werden.

3.5.1 Anwendungsschnittstelle

Die Anwendungsschnittstelle stellt die Funktionalität eines Systems in *Benutzungsschritten* zur Verfügung. Ein Benutzungsschritt beinhaltet das Ausführen einer Funktion des Systems sowie die Rückgabe des Ergebnisses an die Dialogschnittstelle. Das Speichern eines Dokuments stellt ein Beispiel für einen Benutzungsschritt dar. Die Gestaltung der Anwendungsschnittstelle ist aufgrund der direkten Arbeit auf dem System stark von diesem abhängig. Im Falle einer Softwarevisualisierung beschränkt sich die Funktionalität dieser Schnittstelle auf die Manipulation der Repräsentation.

3.5.2 Dialogschnittstelle

Auf der Anwendungsschnittstelle setzt die Dialogschnittstelle, auch Steuerungsschnittstelle genannt, auf. Diese enthält alle Dialoge, die zum Aufruf der einzelnen Benutzungsschritte geführt werden können, wobei der gleiche Schritt von verschiedenen Dialogen aufgerufen werden kann. Zum Beispiel kann das Speichern eines Dokuments mit einem Dialog ausgeführt werden, der nach dem Ort und dem Namen für das Dokument fragt. Die gleiche Funktion des Systems kann aber auch über einen Dialog aufgerufen werden, bei dem Ort und Name unverändert bleiben und lediglich die Frage des Überschreibens gestellt wird.

Ein Dialog kann anhand seines Interaktionsstils als funktionsorientierte oder objektorientierte Interaktion gestaltet sein. Im ersten Fall wird erst eine Funktion bestimmt und anschließend werden diejenigen Objekte ausgewählt, auf die sie angewendet werden soll. Ein bekannter Vertreter der funktionsorientierten Interaktion ist die Kommandozeile. Beim Interaktionsstil der objektorientierten Interaktion werden erst die Objekte und anschließend ein oder mehrere Funktion ausgewählt, die auf die Objekte angewendet werden sollen. Beispielsweise wird bei der Formatierung von Text zuerst die entsprechende Stelle markiert und anschließend die Formatierungsfunktion angewendet.

Entsprechend der Ein- und Ausgabegeräte können zusätzlich verschiedene Dialogarten unterschieden werden. Wie die Geräte haben sich auch die Arten der Dialogführung im Laufe der Zeit immer weiter entwickelt, angefangen bei Kommando- und Menüdialogen über Dialoge mit Masken und Formularen bis hin zu Dialogen mit Fenstersystemen. Eine

besondere Dialogart stellt die direkte Manipulation dar, bei der die relevanten visuellen Objekte direkt mit einem entsprechendem Eingabegerät, wie einer Maus, manipuliert werden können. Die Auswirkungen einer solchen Änderung, beispielsweise das Verschieben von Objekten, ist noch während der Ausführung für den Benutzer sichtbar. Shneiderman et al. definieren folgende Charakteristiken für eine direkte Manipulation (Shneiderman et al. 1983):

- Zu manipulierende Objekte sind stets graphisch dargestellt
- Manipulation erfolgt direkt an den Objekten (nicht über Kommandos oder Menüs)
- Auswirkungen der Aktionen sind unmittelbar sichtbar
- Manipulation durch einstufige, umkehrbare und schnelle/kurze Änderungen

Die direkte Manipulation besitzt im Gegensatz zu anderen Dialogarten eine deutlich höhere Akzeptanz durch die Benutzer. Ein unerfahrener Benutzer kann die Grundfunktionen des Systems einfach und schnell durch Ausprobieren erlernen. Unerwünschte Auswirkungen können leicht wieder rückgängig gemacht werden. Erfahrene Benutzer hingegen können ihre Aufgaben sehr zielgerichtet und zügig erledigen, da die Effekte einer Manipulation bereits bei der Ausführung nachvollzogen werden können.

3.5.3 Ein-/Ausgabenschnittstelle

Die Ein- und Ausgabenschnittstelle nimmt alle Eingaben des Benutzers entgegen und leitet diese an die Dialogschnittstelle weiter. Umgekehrt gibt sie alle für die Interaktion relevanten Darstellungsinformationen an den Benutzer weiter. Jede Kombination aus Ein- und Ausgabe wird als Interaktionsschritt bezeichnet. Im Beispiel des Dialogs zum Speichern eines Dokuments mit Name und Ort stellen die Eingabe des Ortes und des Namens zwei Interaktionsschritte dar.

Für die Ein- und Ausgaben werden spezielle Elemente – so genannte Steuerelemente – in die Repräsentation eingefügt, die in ihrer Gesamtheit als graphische Benutzungsschnittstelle (engl. *graphical user interface*, GUI) bezeichnet werden. Bei der direkten Manipulation sind die Elemente der Repräsentation gleichzeitig auch Steuerelemente. Es werden verschiedene Arten von Steuerelementen unterschieden, wie zum Beispiel Schaltflächen (engl. *button*) oder Auswahlkästen (engl. *checkbox*).

Bei der Gestaltung der Ein- und Ausgabenschnittstelle, speziell den Steuerelementen, müssen unterschiedliche Gestaltungskriterien berücksichtigt werden, die sich aus den Kenntnissen über die Physiologie und Psychologie des Menschen ableiten lassen. Die ISO 9241-12 nennt dazu sieben charakteristische Eigenschaften, die eine Ausgabe für die

Informationsdarstellung besitzen sollte (DIN EN ISO 9241-12). Diese sind in Tabelle 4 mit Bezeichnung und Beschreibung aufgeführt.

Eigenschaft	**Beschreibung**
Klarheit	Der Informationsinhalt wird schnell und zutreffend vermittelt.
Unterscheidbarkeit	Die angezeigte Information kann genau unterschieden werden.
Kompaktheit	Dem Benutzer werden nur die Informationen gegeben, die er für die Bewältigung seiner Aufgabe benötigt.
Konsistenz	Die gleiche Information wird innerhalb der Anwendung entsprechend den Benutzererwartungen stets auf gleiche Art dargestellt.
Erkennbarkeit	Die Aufmerksamkeit des Benutzers wird zur benötigten Information gelenkt.
Lesbarkeit	Die Information ist leicht lesbar.
Verständlichkeit	Die Bedeutung ist leicht verständlich, eindeutig, interpretierbar und erkennbar.

Tabelle 4: Eigenschaften Gestaltung Ein- und Ausgabe nach ISO 9241-12

Aus diesen Eigenschaften sowie den Kenntnissen über die menschliche Physiologie und Psychologie lassen sich konkrete Gestaltungsregeln ableiten, die bei der Gestaltung der Ein-/Ausgabenschnittstelle angewendet werden sollten. Beispielsweise reicht das Färben von Elementen in rot und grün für sich allein genommen nicht aus, um die Eigenschaft der Unterscheidbarkeit zu realisieren. Menschen können eine Rot-Grün-Sehschwäche besitzen und somit die Unterschiede nicht erkennen, womit die Unterscheidbarkeit nicht gegeben wäre. In der (DIN EN ISO 9241-12) sind weitere Gestaltungsregeln inklusive Beispielen enthalten.

3.6 Interaktionstechnik

Aufbauend auf den Gestaltungsmöglichkeiten der Benutzungsschnittstelle können verschiedene Interaktionstechniken für Visualisierungen identifiziert werden, mit denen der Benutzer die Interaktion für einen bestimmten Anwendungsfall realisiert. Foley et al. definieren Interaktionstechnik im Kontext der MCI als die Art und Weise der Benutzung von Ein- und Ausgabegeräten, um eine typische Aufgabe im Mensch-Computer-Dialog zu erfüllen (Foley et al. 1995). Eine Interaktionstechnik stellt somit einen Ausschnitt des Dialogs zwischen System und Benutzer dar, der im Kontext einer Teilaufgabe über die Benutzungsschnittstelle ausgeführt wird. Abbildung 14 verdeutlicht diesen Zusammenhang.

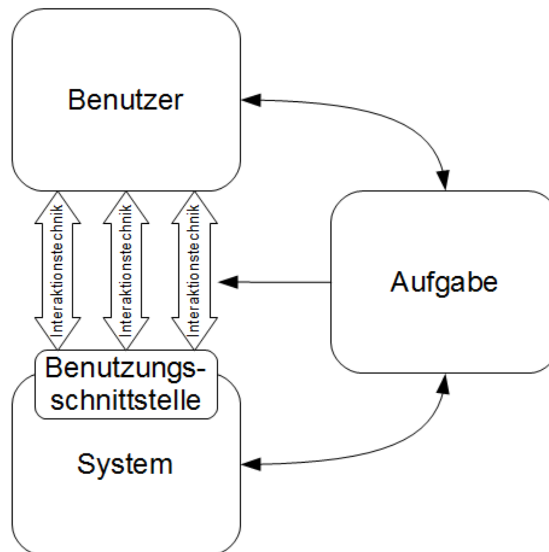

Abbildung 14: Interaktionstechnik: Übersicht

Für die Übertragung auf die Informationsvisualisierung erweitern Yi et al. diese Definition und definieren Interaktionstechnik mit *„the features that provide users with the ability to directly or indirectly manipulate and interpret representations“* (Yi et al. 2007). Diese Definition dient den folgenden Betrachtungen als Grundlage.

Ein Beispiel für die Anwendung einer Interaktionstechnik zeigt Abbildung 15, in der verschiedene Motoren mit Zylindern, Hubraum, Pferdestärke und Gewicht als Tabelle visualisiert sind.

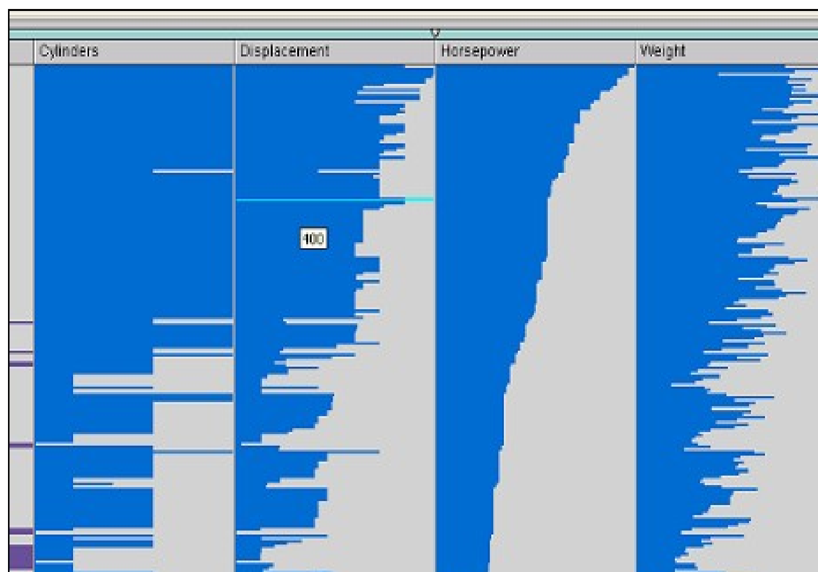

Abbildung 15: Interaktionstechnik: Beispiel
(Yi et al. 2007)

Das Sortieren der Einträge nach den Werten einer Spalte stellt ein Beispiel für eine Interaktionstechnik dar. Das Ziel dieser Technik ist es, Korrelationen zwischen der Eigenschaft einer Spalte und der einer anderen aufzudecken. Im Beispiel werden die Motoren nach Pferdestärke sortiert, womit Verbindungen zu Zylinderanzahl und Hubraum

ersichtlich werden. Eine weitere Interaktionstechnik kann darin bestehen, die Reihenfolge der Tabellenspalten zu ändern. Auf diese Weise können die Verbindungen zwischen den Spalten besser untersucht werden, was insbesondere bei einer großen Anzahl von Spalten hilfreich sein kann.

Für einen besseren Überblick über die verschieden Interaktionstechniken werden diese in eine Taxonomie eingeordnet. Yi et al. haben mehrere Taxonomien aus der Literatur untersucht und dabei festgestellt, dass sie zwar gemeinsame Punkte besitzen, sich allerdings in der Granularität deutlich unterscheiden (Yi et al. 2007). Einige nutzen Kategorien für sehr einfache Formen von Interaktionstechniken, wie beispielsweise *overview*, *zoom* und *filter* (Shneiderman 1996). Andere verwenden Dimensionen für die Einordnung, wie zum Beispiel *continuous*, *stepped*, *passiv* und *composite* (Spence 2007). Amar et al. dagegen klassifizieren die einzelnen Interaktionstechniken nach den Zielen, die ein Benutzer mit ihnen verfolgt, wie das Identifizieren extremer Bereiche oder Anomalien (Amar et al. 2005).

Obwohl jede der untersuchten Taxonomien sicherlich hilfreich für das Verständnis von Interaktionstechniken ist, identifizieren Yi et al. einen entscheidenden Nachteil, den alle untersuchten Taxonomien aufweisen. Jede betrachtet eine Interaktionstechnik entweder aus der Sicht des Systems oder aus der Sicht des Benutzers. Somit erfolgt die Klassifizierung einmal nach technischen Aspekten der Umsetzung und einmal nach den Zielen, die ein Benutzer verfolgt, ohne dabei direkt auf die Interaktion einzugehen. Da die Interaktion aber der Dialog zwischen Benutzer und System ist, muss auch der Zusammenhang zwischen den vom Benutzer verfolgten Zielen und der technischen Umsetzung berücksichtigt werden. Um diesem Zusammenhang mehr Gewicht zu verleihen, schlagen Yi et al. eine eigene Taxonomie vor, die im nächsten Kapitel vorgestellt wird und den weiteren Betrachtungen als Ausgangspunkt dient.

3.7 Taxonomie der Interaktionstechniken

Zur Erarbeitung einer Taxonomie, die den Zusammenhang zwischen einer Interaktionstechnik und dem Zweck des Einsatzes berücksichtigt, stellen Yi et al. in einem ersten Schritt eine Liste mit Interaktionstechniken aus einer Reihe kommerzieller Informationsvisualisierungen zusammen. In einem zweiten Schritt aggregieren sie diese nach verschiedenen Kriterien und stellen dabei fest, dass viele verschiedene Interaktionstechniken zur Erfüllung einer relativ kleinen Menge von Benutzerzielen dient. Davon ausgehend identifizieren sie sieben aussagekräftige Kategorien, die im Folgenden mit ihrer englischen Bezeichnung und dem Einsatzzweck aufgelistet sind:

- *select* - markiere etwas als interessant
- *explore* - zeige etwas anderes
- *reconfigure* - zeige eine andere Anordnung
- *encode* - zeige eine andere Repräsentation
- *abstract/elaborate* - zeige mehr oder weniger Details
- *filter* - zeige nur etwas Bestimmtes
- *connect* - zeige verbundene Elemente

Allerdings lässt sich nicht jede Interaktionstechnik eindeutig nur einer Kategorie zuordnen. Zudem gibt es Interaktionstechniken, die sich keiner der vorgeschlagenen Kategorien zuordnen lassen. Dennoch stellen die von Yi et al. vorgeschlagenen Kategorien eine aussagekräftige Taxonomie für die Einordnung von Interaktionstechniken dar, was sie durch die Anwendung auf eine Menge von Vertretern aus kommerziellen Informationsvisualisierungen belegen. In den nächsten sieben Abschnitten wird jede Kategorie kurz vorgestellt. Ausführliche Beispiele inklusive Abbildungen sind dagegen (Yi et al. 2007) zu entnehmen.

Select

Mit *select*-Interaktionstechniken kann der Benutzer für ihn interessante Elemente in der Visualisierung markieren. Auf diese Weise kann er diese auch bei großen Datenmengen oder bei Änderungen der Anordnung im Auge behalten. Ein Beispiel ist das Markieren einer interessanten Stelle mit einem speziellen Symbol oder die farbliche Hervorhebung eines bestimmten Elements. Eine *select*-Interaktionstechnik kann auch der erste Schritt einer Interaktionstechnik einer anderer Kategorien sein. So wird beispielsweise ein Element zuerst markiert, um anschließend dessen verbundene Elemente anzeigen zu lassen (Interaktionstechnik der Kategorie *connect*).

Explore

Die Menge an Daten, die in einer Visualisierung gleichzeitig dargestellt werden kann, ist begrenzt. Sie wird von der gewählten Darstellungsform, der Größe der Repräsentationsfläche sowie den Grenzen der menschlichen Wahrnehmung bestimmt. Oftmals möchte der Benutzer auch nur einen bestimmten Ausschnitt näher untersuchen und anschließend andere Ausschnitte betrachten. *Explore*-Interaktionstechniken ermöglichen genau diese Vorgehensweise, indem neue Elemente in das Blickfeld gelangen können und andere dieses verlassen, ohne dass dabei die grundlegende Anordnung verändert wird. Eine der wichtigsten Interaktionstechniken dieser Kategorie ist das Schwenken beziehungsweise Verschieben der Kamera (engl. *panning*), die den Ausschnitt

der Repräsentation bestimmt. Dies kann beispielsweise über eine direkte Steuerung erfolgen, wobei der aktuelle Ausschnitt mit der Maus *gegriffen* und anschließend verschoben wird. Alternativ kann dies auch über Steuerelemente, wie zum Beispiel einer *Scrollbar* geschehen. Eine weitere Interaktionstechnik dieser Kategorie ist der *Direct-Walk*. Dabei wechselt der Fokus der Betrachtung *sanft* von einer Position zu einer anderen. Eine Umsetzung kann zum Beispiel durch Hyperlinks realisiert sein, die durch einen Klick mit der Maus den entsprechenden Positionswechsel ausführen.

Reconfigure

Reconfigure-Interaktionstechniken erlauben die Änderung der räumlichen Anordnung der Elemente. Der Benutzer erhält dadurch eine andere Perspektive und kann so vorher verdeckte Eigenschaften oder Beziehungen identifizieren. Ist die Darstellungsform beispielsweise eine Tabelle, so kann eine *reconfigure*-Interaktionstechnik darin bestehen, die Einträge nach den Werten einer Spalte zu sortieren. Anschließend kann die Eigenschaft dieser Spalte mit der Eigenschaft einer anderen Spalte verglichen werden, um einen Zusammenhang zu identifizieren. Das Beispiel für eine Interaktionstechnik aus Kapitel 3.6, bei dem die Motoren nach Pferdestärke sortiert wurden, lässt sich demnach in diese Kategorie einordnen. Übertragen auf die Softwarevisualisierung könnten beispielsweise Klassen mit verschiedenen Codemetriken, wie LOC oder Methodenanzahl als Tabelle dargestellt werden. Durch das Sortieren nach LOC lässt sich eine direkt proportionale Beziehung zu Methodenanzahl nachvollziehen. Besitzt eine Klasse hingegen viele Zeilen Code, aber nur wenige Methoden kann dies ein Hinweis auf schlechten Code sein, was zum Beispiel durch Komplexitätsmetriken bestätigt werden kann. *Reconfigure*-Interaktionstechniken können aber auch genutzt werden, um das Verdecken von Elementen zu reduzieren, was besonders bei großen Datenmengen oft der Fall ist. Das Drehen der Ansicht in einer dreidimensionalen Repräsentationen ermöglicht es zum Beispiel, entfernte Elemente, die durch ein vorgelagertes verdeckt werden, zu erkennen. An dieser Stelle zeigt sich allerdings, dass es nicht immer einfach ist, eine Interaktionstechnik nur exakt einer Kategorie zuzuordnen. Das Drehen der Ansicht könnte ebenso als *explore*-Interaktionstechnik aufgefasst werden.

Encode

Zur Änderung des grundlegenden Erscheinungsbildes der Repräsentation werden *encode*-Interaktionstechniken verwendet. Diese Änderungen können entscheidenden Einfluss auf die Qualität der Visualisierung haben, da die Interpretation durch den Benutzer direkt vom Erscheinungsbild abhängig ist. Das Ändern der Visualisierungstechnik stellt eine Interaktionstechnik dieser Kategorie dar, die in vielen Visualisierungswerkzeugen realisiert

ist. Mit dem Wechsel, beispielsweise von einer abstrakten zu einer natürlichen Metapher, sollen neue Aspekte oder Beziehungen erkannt werden, die ein Benutzer vorher nicht erkennen konnte.

Eine andere, weit verbreitete Interaktionstechnik dieser Kategorie ist das Färben der Elemente nach einer bestimmten Farbkodierung. Diese kann zum Beispiel als Abbildung einer Variable oder Kennzahl auf einen Farbverlauf realisiert sein. Ein klassisches Beispiel ist die Höheneinfärbung einer Landkarte von grün (flach) über gelb bis zu braun (hoch). In einer Softwarevisualisierung können dagegen verschiedene Metriken, wie LOC oder Testabdeckung, in eine entsprechende Farbkodierung übertragen werden. Neben der Färbung können aber auch Änderungen der Größe oder der Form realisiert werden.

Abstract/Elaborate

Mit Abstract-/Elaborate-Interaktionstechniken erhält der Benutzer die Möglichkeit, die Anzahl der Details einer Repräsentation zu bestimmen. Neben zusätzlichen Informationen zu den Elementen kann er so auch den Abstraktionsgrad beeinflussen. Eine einfache Technik dieser Kategorie stellt der *Tooltip* dar. Fährt der Benutzer mit der Maus über ein Element, werden ihm zusätzliche Details angezeigt. *Zoomen* ist weiteres Beispiel dieser Kategorie. Dabei ändert der Benutzer den Maßstab der Repräsentation, um sich entweder einen Überblick über eine große Datenmenge zu verschaffen (*zoom out*) oder eine kleine Datenmenge detailliert zu betrachten (*zoom in*). Der Unterschied zur Kategorie *encode* ist, dass die Repräsentation beim Zoomen nicht grundlegend verändert wird. Details, die bereits enthalten waren, werden lediglich deutlicher sichtbar oder verschwinden dagegen im Kontext.

Filter

Soll nur eine ausgewählte Menge von Elementen betrachtet werden, so kann dies mit *filter*-Interaktionstechniken realisiert werden. Dazu werden verschiedene Bedingungen definiert, so dass der Benutzer nur diejenigen Elemente angezeigen lassen kann, die diesen Kriterien genügen. Alle übrigen Elemente werden dagegen ausgeblendet. Alternativ kann auch die Darstellung der Elemente verändert werden, indem entweder die ausgewählten Elemente hervorgehoben werden oder die übrigen Elemente in den Hintergrund treten. Änderungen, die mit *filter*-Interaktionstechniken erzeugt wurden, können zu jedem Zeitpunkt wieder rückgängig gemacht werden, womit alle vorübergehend ausgeblendeten oder anders dargestellten Elemente zu ihrer ursprünglichen Darstellung zurückkehren. Ein verbreiteter Vertreter dieser Kategorie sind *Dynamic Query Controls*, mit denen der Benutzer einen bestimmten Wert oder Wertebereich auswählen kann. Dazu wählt er beispielsweise eine *Checkbox* aus (Wert) oder verschiebt einen *Slider* (Wertebereich), woraufhin die

ausgewählte Elementmenge unmittelbar in der entsprechenden Art und Weise visualisiert wird.

Connect

Connect-Interaktionstechniken werden zum einen dafür verwendet, Beziehungen zwischen Elementen hervorzuheben, die bereits in der Repräsentation enthalten sind. Der Benutzer soll somit die oftmals komplexen Zusammenhänge erkennen und nachvollziehen können. Besteht die Repräsentation zum Beispiel aus einem Graph mit Knoten und Kanten, so können beim Auswählen eines Knotens die verbundenen Knoten ebenfalls markiert werden.

Zum anderen können mit *connect*-Interaktionstechniken auch zusätzliche Elemente angezeigt werden, die nicht direkt in Verbindung mit dem Element dargestellt werden, aber eine Relevanz für dieses besitzen. Dies können zum Beispiel weit entfernte Elemente sein, bei denen eine direkte visuelle Abbildung der Beziehung aufgrund der Entfernung nicht nachvollzogen werden könnte. Eine weiterer Anwendungsfall ist immer dann gegeben, wenn die Elemente eine zu große Anzahl von Beziehungen besitzen, so dass deren Darstellung die Erkenntniserlangung des Benutzers behindern würden. Mit einer entsprechenden *connect*-Interaktionstechnik kann er stattdessen nur eine Auswahl der Beziehungen bestimmen, die angezeigt werden sollen.

3.8 Konzept zur Interaktion mit einer Softwarevisualisierung

In diesem Kapitel soll ein Konzept für die Interaktion mit einer Softwarevisualisierung der Struktur erarbeitet werden. Dabei werden allerdings keine konkreten Vorschläge für die Gestaltung einer Benutzungsschnittstelle gemacht, da diese zum Teil bereits in den Kapiteln 3.4 und 3.5 gegeben wurden und des Weiteren stark von den jeweiligen technischen Gegebenheiten der Softwarevisualisierung abhängen. Vielmehr soll in dem Konzept die Funktionalität beschrieben werden, die eine Benutzungsschnittstelle für die Interaktion mit einer Softwarevisualisierung der Struktur in Form von Interaktionstechniken zur Verfügung stellen sollte. Dazu werden in den folgenden Abschnitten diejenigen Kategorien der Interaktionstechniken identifiziert, die für die Realisierung der einzelnen Ziele aus Kapitel 3.3 eingesetzt werden können. Im letzten Abschnitt werden die Erkenntnisse zusammengefasst und ein Fazit für die Interaktion mit einer Softwarevisualisierung der Struktur verfasst.

Überblick & Ausschnitt

Für das Betrachten und Auswählen von Ausschnitten aus der Gesamtheit bieten sich Interaktionstechniken der Kategorie *explore* an. Durch das Bewegen der Kamera

beispielsweise kann der Benutzer direkt den Ausschnitt der Betrachtung bestimmen. Mittels *abstract/elaborate*-Interaktionstechniken, wie zum Beispiel zoomen, kann er zudem den Grad der Abstraktion ändern, so dass bei Bedarf zwischen Überblick (zoom out) und Ausschnitt (zoom in) gewechselt werden kann. Das individuelle Arrangieren von Elementen durch Techniken der Kategorie *reconfigure* und das Ausblenden von Elementen durch *filter*-Interaktionstechniken stellen weitere Möglichkeiten dar, einen Ausschnitt zu bestimmen. Des Weiteren können auch *select*-Interaktionstechniken einen Beitrag leisten, indem sie durch das Markieren von Elementen Orientierungshilfen erstellen.

Beziehungen Erfassen

Für die Unterstützung des Erkennens und des Verstehens von Beziehungen zwischen den Elementen eignen sich die Interaktionstechniken der Kategorie *connect*. Mit ihnen ist es möglich, aktuell interessierende Beziehungen visuell hervorzuheben und auch Beziehungen zu erfassen, die keine direkte visuelle Repräsentation besitzen. Für das Bestimmen der Elemente, für die Beziehungen angezeigt werden sollen, bilden Interaktionstechniken der Kategorie *select* die Grundlage. Sollen dagegen für alle Elemente die darzustellenden Beziehungsarten ausgewählt werden, so kommen dafür *filter*-Interaktionstechniken zum Einsatz. Auf diese Weise kann zum Beispiel bestimmt werden, dass nur die Vererbungsbeziehungen und keine Assoziationen angezeigt werden sollen. Oftmals genügt der aktuell gewählte Ausschnitt nicht, um alle Beziehungen zu erfassen, womit auch *explore*-Interaktionstechniken genutzt werden müssen. So kann beispielsweise ein weit entferntes verbundenes Element über einen *Direct-Walk* angesteuert werden, so dass der Ausschnitt der Betrachtung automatisch auf dieses gerichtet wird.

Grad der Details ändern

Über das Ein- und Ausblenden von Elementen durch *filter*-Interaktionstechniken kann der Benutzer direkt die Anzahl der Details in einem Ausschnitt beeinflussen. Mit *abstract/elaborate*-Interaktionstechniken können zudem zusätzliche Informationen nach Bedarf angezeigt werden, wie zum Beispiel die LOC einer Methode in einem Tooltip. Dies könnte in einem Ausschnitt dazu genutzt werden, die Verteilung der LOC auf die einzelnen Methoden oder Klassen zu ermitteln. Eine andere Möglichkeit weitere Informationen in einem Ausschnitt zu erhalten, besteht in der Änderung des Erscheinungsbildes durch Interaktionstechniken der Kategorie *encode*. Werden zum Beispiel die LOC-Werte auf die Größe der Elemente abgebildet, kann der Benutzer direkt erkennen, dass große Methoden mehr LOC besitzen als kleine. Auch *connect*-Interaktionstechniken können in gewisser Weise zusätzliche Details liefern, wenn Verbindungen zwischen Elementen visualisiert

werden, die vorher nicht Bestandteil der Repräsentation waren. Für *connect* wie auch *abstract/elaborate* können *select*-Interaktionstechniken wieder eine Grundlage darstellen. Die Interaktionstechniken der Kategorien *explore* und *reconfigure* beeinflussen dagegen nicht direkt den Grad der Details, sondern ändern den Ausschnitt und sorgen auf diese Weise für mehr oder weniger Informationen.

Elemente vergleichen

Das Vergleichen von Elementen anhand verschiedener Eigenschaften und Kriterien kann durch Interaktionstechniken der Kategorien *reconfigure*, *encode* und *abstract/elaborate* erreicht werden. Klassen können zum Beispiel durch das bloße Umordnen mittels einer *reconfigure*-Interaktionstechnik bereits nach einfachen Kriterien wie der Methodenanzahl verglichen werden. Eine Voraussetzung dafür können wieder die *select*-Interaktionstechniken bilden. Die Änderungen des Erscheinungsbildes (Kategorie *encode*) durch Kodierung verschiedener Eigenschaften schafft eine weitere Grundlage für das Vergleichen von Elementen. Für gezielte und detaillierte Vergleiche können zusätzlich *reconfigure*-Interaktionstechniken eingesetzt werden, um interessierende Elemente räumlich nah anzuordnen und so beispielsweise auch feine Farb- oder Größenunterschiede zu erkennen. Eine weitere Grundlage für das Vergleichen liefert das Einblenden zusätzlicher Details mittels einer *abstract/elaborate*-Interaktionstechnik. Eigenschaften können zum Beispiel in einem Tooltip aufgelistet sein, so dass ein Vergleich völlig unabhängig von der Position der Elemente möglich ist.

Elemente identifizieren

Das Identifizieren einer Menge von Elementen nach verschiedenen Eigenschaften wird überwiegend durch *filter*-Interaktionstechniken realisiert. Die Gestaltung der Filterbedingung kann dabei nach den unterschiedlichsten Kriterien erfolgen, wie zum Beispiel einer Namenskonvention, diversen Metriken oder anhand der Zugehörigkeit zu einem Architekturaspekt wie der Unterteilung in Schichten.

Schlussfolgerung

In Tabelle 5 sind noch einmal alle identifizierten Kategorien der Interaktionstechniken für die jeweiligen Ziele in einer zweidimensionale Matrix dargestellt.

		Interaktionstechniken						
		select	explore	reconfigure	encode	abstract/Elaborate	filter	connect
Ziele	Überblick & Ausschnitt	X	X	X		X	X	
	Beziehungen erfassen	X	X				X	X
	Grad der Details ändern	X			X	X	X	X
	Elemente vergleichen	X		X	X	X		
	Elemente identifizieren						X	

Tabelle 5: Ziele der Interaktion und realisierende Interaktionstechniken

Wie der Tabelle zu entnehmen ist, werden für das Erreichen der einzelnen Ziele verschiedene Interaktionstechniken aus jeder Kategorie eingesetzt. Die Bedeutung einer Kategorie lässt sich dabei aber nicht direkt aus der Anzahl von unterstützten Zielen ableiten. *Explore*-Interaktionstechniken besitzen beispielsweise eine herausragende Stellung beim Erlangen von Erkenntnissen, da der Benutzer oftmals erst durch das Erforschen der Visualisierung eines der Ziele erreichen kann. Interaktionstechniken der Kategorie *select* werden dagegen zwar häufiger eingesetzt, spielen aber für das Verständnis eher eine untergeordnete Rolle, da sie oftmals nur als Vorbedingung für andere Interaktionstechniken eingesetzt werden. Für die Bestimmung der Bedeutungen der einzelnen Kategorien müsste daher zunächst der Einfluss der einzelnen Ziele auf den gesamten Prozess der Erkenntniserlangung identifiziert werden, was im Rahmen dieser Arbeit nicht möglich ist. Aus diesem Grund wird im Folgenden davon ausgegangen, dass jede Kategorie einen Beitrag für das Erlangen von Erkenntnissen leistet. Demzufolge muss eine Benutzungsschnittstelle für die Interaktion mit einer Softwarevisualisierung der Struktur so gestaltet sein, dass sie Interaktionstechniken aus jeder Kategorie bereitstellt. Nur dann kann sie den Benutzer beim Erreichen aller fünf Ziele unterstützen.

Um dies zu belegen, soll der in (R. Müller 2009) beschriebene Prototyp um eine entsprechende Benutzungsschnittstelle erweitert werden. Im folgenden Kapitel werden dafür die technischen Grundlagen und Werkzeuge vorgestellt, die sowohl bei der Implementierung des Prototyps von Müller verwendet wurden, als auch für dessen Erweiterung eingesetzt werden.

4 Technische Grundlagen des Prototyps

Dieses Kapitel legt die technischen Grundlagen für das Verständnis des Prototyps. Da dieser als Eclipse Plugin konzipiert und implementiert ist, wird Eclipse in einem ersten Unterkapitel kurz vorgestellt. Im zweiten Unterkapitel wird das openArchitectureWare-Framework vorgestellt, welches vom Prototyp genutzt wird und in Eclipse integriert ist. Anschließend wird das Zielformat der Visualisierung – extensible3D – vorgestellt. Dabei liegt der Schwerpunkt auf den Ansatzpunkten zur Interaktion.

4.1 Eclipse

Eclipse ist eine der am weitest verbreiteten integrierten Entwicklungsumgebungen (engl. *Integrated Development Environment*, IDE) für die Programmiersprache Java. Sie wird von der Eclipse Foundation als *Open-Source*-Projekt verwaltet und unterliegt der Eclipse Public License (EPL). Das Besondere an dieser IDE ist, dass sie nicht ausschließlich auf die Programmiersprache Java zugeschnitten ist. Eclipse kann für jede Programmiersprache genutzt werden und darüber hinaus auch jeden anderen beliebigen Artefakttyp bearbeiten. Es handelt sich somit bei Eclipse um ein universell einsetzbares Werkzeug für alles mögliche und nichts im Speziellen[2]. Abbildung 16 zeigt den Aufbau der Oberfläche von Eclipse mit den grundlegenden Elementen.

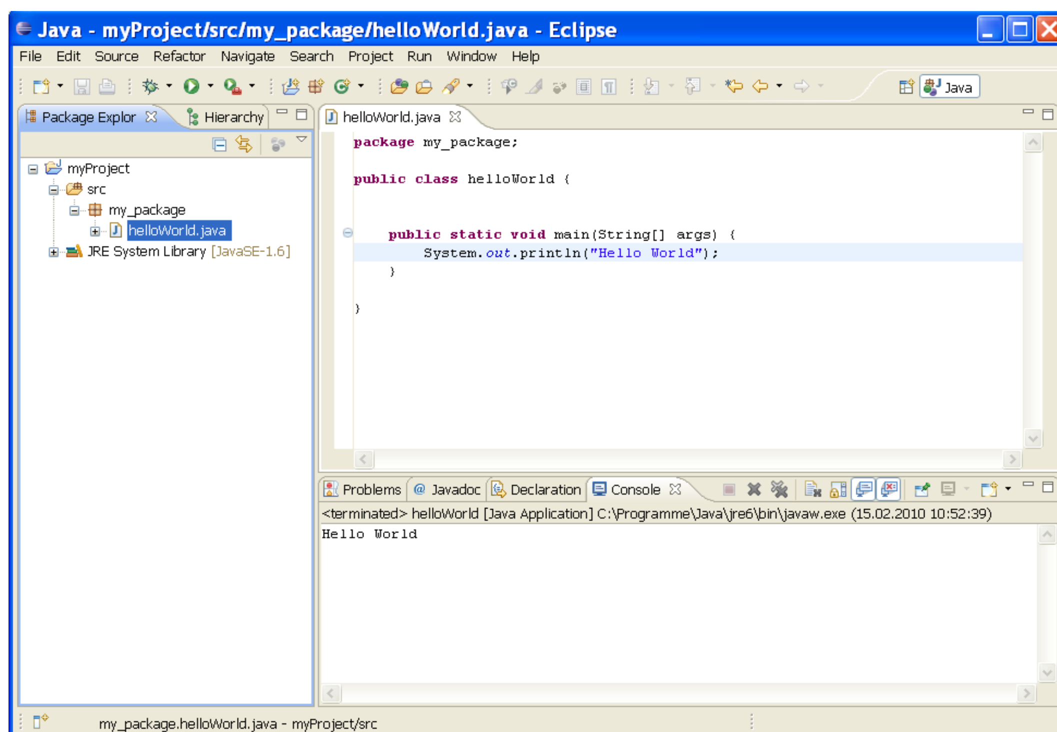

Abbildung 16: Eclipse-Oberfläche

2 „*The Eclipse platform itself is a sort of universal tool platform - it is an IDE for anything and nothing in particular*" http://www.eclipse.org/platform/overview.php

Der *Package Explorer* auf der linken Seite bildet die Projektstruktur ab, während der zentral gelegene *Editor* (im Beispiel für Java-Dateien) die Bearbeitung der einzelnen Projektartefakte realisiert. Unterhalb des Editors befindet sich die *Console* in der Programmausgabe erscheint.

Die Grundlage für den universellen Einsatz liefert die modular aufgebaute Architektur von Eclipse. Mit ihr ist es dem Benutzer möglich, die Umgebung für seine Erfordernisse anzupassen und zu erweitern. Beispielsweise kann jedes Element der Oberfläche aus Abbildung 16 verändert oder ein zusätzliches Element, wie ein Editor für einen speziellen Dateityp, hinzugefügt werden. Der in dieser Arbeit vorgestellte Prototyp ist als eine solche Erweiterung – genannt *Plugin* - implementiert. Im Folgenden soll die Funktionsweise des Plugin-Mechanismus in Eclipse dargestellt werden.

Alle Funktionen in Eclipse, jeder Editor und jedes Menü, sind durch ein Plugin realisiert. Das Laden, Verwalten und Ausführen der Plugins ist die einzige Aufgabe des Eclipse-Kerns. Die Plugins sind über Erweiterungspunkte (engl. *extension points*) lose miteinander gekoppelt. Somit kann ein Plugin auf die Funktionalität anderer Plugins zugreifen und diese erweitern. Umgekehrt kann ein Plugin auch eigene Erweiterungspunkte definieren, auf die wiederum andere zugreifen können.

Neben Plugins gehören noch zwei weitere Bausteine – Fragmente und Features – zur Architektur von Eclipse. Ein Fragment fügt einem bereits fertig gestelltes Plugin weitere Funktionalität hinzu. So können beispielsweise Sprachpakete oder plattformspezifische Inhalte nachträglich ergänzt werden. Ein Plugin und seine zugehörigen Fragmente werden von Eclipse als eine Einheit betrachtet.

Ein Feature hingegen bündelt und verwaltet eine Menge von Plugins als ein Produkt. Der Vorteil liegt darin, dass nicht jedes Plugin einzeln betrachtet werden muss und somit der Überblick über die installierten Plugins erhalten bleibt. Zusätzlich kann ein Feature Produktinformationen, wie Lizenz, Installationshinweise oder Herkunft zur Verfügung stellen. Darüber hinaus liefert Eclipse einen Update-Manager, mit dem sich ein Feature einfach aktualisieren lässt.

Für zusätzliche Informationen zur Erweiterung von Eclipse, insbesondere der Entwicklung von Plugins, wird an dieser Stelle auf (Gamma & Beck 2004) verwiesen. Nachdem nun Eclipse und seine Plugin-Architektur kurz umrissen wurden, kann im folgenden Kapitel das openArchitectureWare-Framework für Eclipse vorgestellt werden.

4.2 Das openArchitectureWare-Framework

OpenArchitectureWare (oAW) ist ein modular aufgebautes Generator-Framework für Eclipse. Es enthält eine Menge von Werkzeugen zur Entwicklung von Generatoren für die modellgetriebene Softwareentwicklung. Dieser Arbeit liegt die oAW Version 4.3.1 zugrunde, welches unter der EPL frei verfügbar ist. Dieses Kapitel soll einen Überblick über den Aufbau und die Funktionsweise des Frameworks liefern und ausgewählte Werkzeuge vorstellen, die im Prototyp verwendet werden. Zunächst werden dazu im folgenden Unterkapitel die theoretischen Grundlagen der modellgetriebenen Softwareentwicklung näher betrachtet.

4.2.1 Modellgetriebene Softwareentwicklung

Die modellgetriebene Softwareentwicklung (engl. *Model Driven Software Development, MDSD*) stellt Modelle als die zentralen Artefakte des Entwicklungsprozesses in den Fokus. Dahinter steht die Idee, dass der Code nicht wie üblich manuell entwickelt, sondern aus formalen Modellen mittels geeigneter Werkzeuge automatisiert erzeugt wird. Modelle werden somit nicht nur für die Dokumentation und Kommunikation genutzt, sondern erhalten den Status des Quellcodes. Stahl et al. definieren MDSD wie folgt:

„Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen.“ (Stahl et al. 2007)

Durch diese Abstraktion von der Implementierung rückt die fachliche Domäne der Software in den Vordergrund. Als Domäne wird ein Bereich bezeichnet, in dessen Kontext die Software entwickelt wird. Ein Beispiel ist die Entwicklung einer Buchhaltungssoftware mit der dazugehörigen Domäne Buchhaltung. Eine formalisierte Domäne des MDSD besteht aus fachspezifischen Begriffen und Sprachen, einem Metamodell und Vorschriften zur Transformation. Der Wechsel der Perspektive – von der technischen Implementierung hin zur fachlich Beschreibung – bringt eine Reihe von Vorteilen bei der Entwicklung von Software mit sich, die sich zum Beispiel in Produktivität, Qualität oder Portabilität niederschlagen. Zusätzlich kann mit diesem Ansatz leichter eine einheitliche Architektur für Software erzeugt werden, welche eine wichtige Voraussetzung für Wiederverwendbarkeit und Flexibilität darstellt.

MDSD gilt als eine pragmatische Umsetzung des allgemeineren Standards Model Driven Architecture (MDA) der Object Management Group (OMG). Auf eine detaillierte Betrachtung des MDA-Standards wird an dieser Stelle verzichtet, da sie den Umfang

dieser Arbeit sprengen würde. Stattdessen wird auf (Gruhn et al. 2006) und die Spezifikation der OMG[3] verwiesen.

Im Folgenden sollen die beiden grundlegenden Konzepte der MDSD – Modelle und Transformationen – vorgestellt werden. Eine vertiefte Betrachtung des Themas liefern (Stahl et al. 2007) und (Pietrek & Trompeter 2007).

Modell und Metamodell

Ein Modell ist allgemein formuliert eine abstrahierte Sicht auf ein System oder ein Teilsystem in geeigneter Form. Im Kontext von MDSD wird ein Modell mit einer wohldefinierten Sprache aus Syntax und Semantik beschrieben. Die Syntax definiert die möglichen Elemente und deren Kombinationsvarianten, während die Semantik diesen Kombinationen eine Bedeutung zuordnet. Modelle können in Textform, graphischer Form oder einer Mischung aus beiden dargestellt werden.

Ein Metamodell ist vereinfacht ausgedrückt ein Modell, welches ein anderes Modell beschreibt. Die Beziehung zwischen System, Modell und Metamodell soll folgendes Beispiel verdeutlichen. Ein Gebäude (System) wird durch seinen Grundriss (Modell) abstrahiert dargestellt. Das Metamodell eines Grundrisses beschreibt, wie dieser aufgebaut sein muss (zweidimensionale Draufsicht). Es definiert, aus welchen Elementen sich ein Grundriss zusammensetzen kann (z.B. Wände, Türen, Treppen und Fenster) und wie diese dargestellt und angeordnet werden können.

Im Sinne von MDSD definiert ein Metamodell die Syntax und Semantik der Sprache von Modellen für eine bestimmten Domäne. Diese Sprachdefinition wird als domänenspezifische Sprache (engl. *Domain Specific Language*, DSL) bezeichnet und ist Bestandteil der formalisierten Domäne.

Da ein Metamodell auch ein Modell ist, kann es selbst wiederum ein Metamodell besitzen, welches als Metametamodell bezeichnet wird. Dies kann theoretisch beliebig fortgesetzt werden oder es wird ein Metamodell definiert, welches sich selbst beschreibt. Es ist dann Modell und Metamodell in einem. Ein formalisiertes Metametamodell, mit dem alle Metamodelle und Modelle einheitlich beschrieben werden können, ist die Grundvoraussetzung für die automatisierte Überführung von Modellen in andere Modelle oder Quellcode. Einer der wichtigsten Vertreter für MDSD ist das Metametamodell der OMG für die MDA, genannt *Meta Object Facility* (MOF). Die OMG unterteilt ihre Metadaten-Architektur in mehrere Ebenen, mit M0 bis M3 bezeichnet, dessen oberste Ebene die MOF bildet. Demnach ist die M0 Ebene die Instanz mit den Objekten der realen

---

3 http://www.omg.org/mda/specs.htm

Welt, während die M1 Ebene das Modell dieser Instanz darstellt. Tabelle 6 gibt einen Überblick über die Modellebenen der OMG Metadaten-Architektur.

Ebene	Modelle	Beschreibung
M3	MOF	Metametamodell, in sich selbst definiert
M2	UML-Metamodell	Metamodell des Aufbaus der einzelnen UML-Modelle
M1	UML-Modelle	Modelle der Objekte der realen Welt (z.B. Klassendiagramm)
M0	Instanz	Objekte der realen Welt

Tabelle 6: Modellebenen MDA

Transformationen

Stahl et al. unterscheiden bei MDSD zwei Arten von Transformationen:

- Modell-zu-Modell-Transformation (engl. *model-to-model transformation*, M2M)
- Modell-zu-Code-Transformation (engl. *model-to-code transformation*, M2C)

In der M2M-Transformation werden ein oder mehrere Ausgangsmodelle in ein Zielmodell übertragen. Dabei werden drei Fälle unterschieden:

- Modellmodifikation
- Modelltransformation
- Modellverwebung

Bei der Modellmodifikation wird das Ausgangsmodell modifiziert, indem zum Beispiel Elemente hinzugefügt beziehungsweise entfernt oder einfach nur Werte der Elemente geändert werden. Das so veränderte Ausgangsmodell ist dann das Zielmodell. Die Modelltransformation überführt ebenfalls ein Quellmodell in ein Zielmodell. Im Gegensatz zur Modellmodifikation besitzen aber beide Modelle ein unterschiedliches Metamodell. In der Modellverwebung werden mindestens zwei Ausgangsmodelle in ein Zielmodell transformiert. Dabei spielt es keine Rolle, ob die Modelle das gleiche Metamodell besitzen oder nicht.

Die M2C-Transformation erzeugt Quellcode aus einem Ausgangsmodell. Dafür existieren Schablonen (engl. *template*) mit entsprechenden Slots, die bei der Transformation mit Werten aus dem Modell belegt werden.

4.2.2 Aufbau des Frameworks

Das oAW-Framework besteht aus einer Workflow-Engine, einer dreiteiligen Sprachfamilie und dem Xtext-Framework. Alle Bestandteile sind ausführlich und mit Beispielen im *openArchitectureWare User Guide*[4] beschrieben.

4 http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/

Die Workflow-Engine realisiert einen Generator mit Hilfe eines *Workflow*. Ein Workflow in oAW beschreibt eine nacheinander ausgeführte Abfolge von Schritten. Er stellt somit die Struktur des Generators dar, während die einzelnen Schritte die operative Ausführung realisieren. Kapitel 4.2.3 stellt den Aufbau eines oAW-Workflow detailliert vor.

Die dreiteilige Sprachfamilie, bestehend aus *Check*, *Xtend* und *Xpand*, realisieren die Transformationen der MDSD. Alle drei basieren dabei auf dem gleichen Typ- und Ausdruckssystem und bauen auf dem Metamodell Ecore auf, welches in Kapitel 4.2.4 vorgestellt wird. *Check* dient zur Überprüfung von Modellen hinsichtlich bestimmter Beschränkungen. Damit sollen mögliche Fehlerursachen in späteren Transformationen bereits im Vorfeld erkannt werden. Erfüllt ein Modell die in *Check* definierten Bedingungen nicht, wird eine entsprechende Meldung ausgegeben und die Ausführung des Workflow abgebrochen. Auf eine genauere Betrachtung der Sprache *Check* wird mit Verweis auf den User Guide verzichtet, da sie nur von geringer Relevanz für den Prototyp ist. In *Xtend* werden M2M-Transformationen beschrieben, wobei alle drei Fälle – Modellmodifikation, Modelltransformation, Modellverwebung – realisiert werden können. In Kapitel 4.2.5 wird diese Sprache ausführlich beschrieben, da die M2M-Transformationen das Herzstück des Prototyps sind. Als dritter Teil der Sprachfamilie realisiert *Xpand* die M2C-Transformationen durch die bereits erwähnten Schablonen. Da allerdings keine M2C-Transformationen im Prototyp vorkommen, wird es an dieser Stelle nur der Vollständigkeit halber genannt.

Mit Hilfe von Xtext kann eine eigene DSL definiert werden, für die anschließend automatisiert ein Editor mit einem Parser generiert wird. Wie bereits für *Xpand* wird auch Xtext an dieser Stelle nur der Vollständigkeit halber erwähnt.

4.2.3 oAW-Workflow

Ein Workflow wird in oAW durch eine Datei im XML-Format (Extensible Markup Language[5]) mit der Endung *oaw* definiert. Die einzelnen Schritte werden durch sogenannte Ablaufkomponenten realisiert. Das oAW-Framework liefert bereits eine Auswahl dieser Komponenten für die häufigsten Anwendungsfälle. Dazu gehören beispielsweise das Einlesen und Überprüfen von Modellen, oder aber das Ausführen einer Transformationsvorschrift. Zusätzlich zu den vorhandenen Ablaufkomponenten können auch eigene in Java implementiert werden. Dazu muss lediglich eine Schnittstelle, die das Framework bereitstellt, implementiert werden. Listing 1 zeigt einen oAW Workflow der ein Modell vom Typ X3D einliest und anschließend eine Transformation durchführt.

5 http://www.w3.org/XML/

```
1  <workflow>
2      <property name="metamodel.x3d" value="metamodel/x3d-3.2.xsd"/>
3      <property name="model.x3d" value="src-gen/myx3dModel.x3d"/>
4      <property name="extensions.dir" value="org::x3d::generator::extensions::"/>
5
6      <component class="org.openarchitectureware.xsd.XMLReader">
7          <metaModel id="x3d" class="org.openarchitectureware.xsd.XSDMetaModel">
8              <schemaFile value="${metamodel.x3d}" />
9          </metaModel>
10         <uri value="${model.x3d}" />
11         <modelSlot value="x3dModel" />
12     </component>
13
14     <component class="oaw.xtend.XtendComponent">
15         <metaModel idRef="x3d" />
16         <invoke value="${extensions.dir}myExtension::modifyX3d(x3dModel)" />
17         <outputSlot value="x3dModelModified" />
18     </component>
19 </workflow>
```

Listing 1: oAW-Workflow: Beispiel

Im Beispiel werden zwei grundlegende Elemente verwendet: Eigenschaften (engl. *property*) und Ablaufkomponenten, auch nur Komponenten (engl. *component*) genannt. Eine Eigenschaft kann als Definition einer Variablen verstanden werden, die zum Beispiel einen Dateipfad enthält. In den Zeilen 2 bis 4 sind drei Eigenschaften definiert, welche die Pfade zu den Dateien des Metamodells, des Modells und den *Xtend*-Transformationsvorschriften enthalten. Nachdem eine Eigenschaft definiert wurde, kann sie innerhalb der Workflowbeschreibung mit der Syntax *${Eigenschaft}* verwendet werden.

Eine Komponente deklariert einen Generatorschritt und wird entsprechend ihrer Reihenfolge im Workflow ausgeführt. Das Attribut *class* definiert die Art der Komponente über die ausführende Java-Klasse. Die erste Komponente (Zeile 6 bis 12) liest demzufolge ein XML-Modell ein, während die zweite Komponente (Zeile 14 bis 18) eine Transformation nach einer *Xtend*-Vorschrift ausführt. Die möglichen Unterelemente einer Komponente sind von ihrer Art abhängig. Die Komponente zum Einlesen besitzt zum Beispiel ein Unterelement für das verwendete Metamodell (*metamodel*), für den Pfad zur Datei des Modells (*uri*) und einen Slot für die Ablage des eingelesenen Modells (*modelSlot*). Ein solcher Slot wird für die Weitergabe von Modellen zwischen den einzelnen Komponenten genutzt. In Listing 1 wird das eingelesene Modell in Zeile 11 im

Slot *x3dModel* abgelegt und in Zeile 16 der Transformationsvorschrift *modifyX3d(x3dModel)* als Parameter übergeben. Nach der Transformation wird das modifizierte Modell im Slot *x3dModelModified* abgelegt und könnte beispielsweise von einer weiteren Komponente in eine Datei geschrieben werden.

4.2.4 Ecore

Das oAW-Framework verwendet das Metametamodell des *Eclipse Modeling Frameworks* (EMF), das als *EMF Core* beziehungsweise *Ecore* bezeichnet wird. EMF ist Teil des *Eclipse Modeling Projects*, zu dem auch die Komponenten des oAW-Frameworks ab der nächsten Version gehören werden. Abgesehen von der zentralen Bedeutung für oAW, werden *Ecore*-Modelle auch als Ausgangspunkt für den Generator des Prototyps verwendet, da mit ihnen Strukturinformationen ideal abgebildet werden können. *Ecore* basiert auf einer Untermenge des von der OMG definierten MOF-Standards, der sogenannten *Essential Meta-Object Facility*. Wie MOF ist auch *Ecore* in sich selbst definiert und somit Modell und Metamodell in einem. Abbildung 17 zeigt einen Ausschnitt der Klassenstruktur des *Ecore*-Modells. Für eine komplette und detaillierte Beschreibung von EMF inklusive des *Ecore*-Modells wird auf (Budinsky et al. 2004) verwiesen.

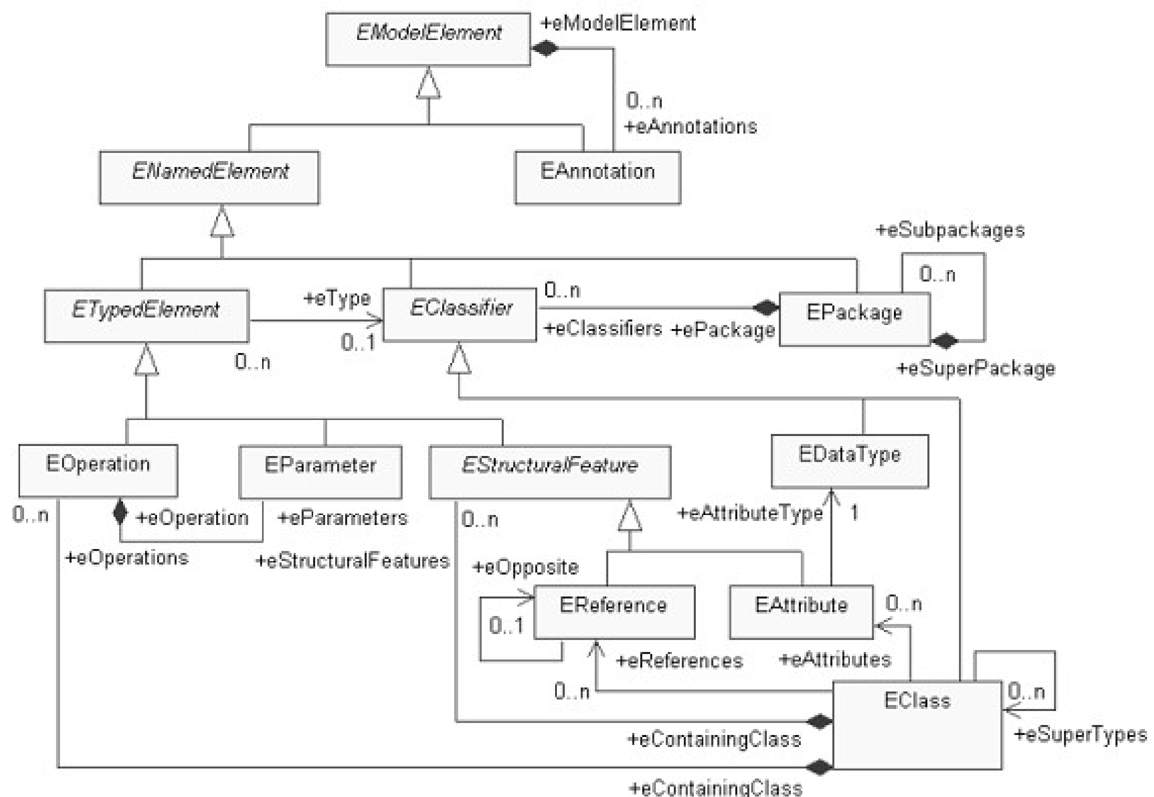

Abbildung 17: Ecore-Klassenmodell: Auszug (Müller 2009)

Das Klassendiagramm zeigt die Zusammenhänge zwischen den wichtigsten Elementen des *Ecore*-Modells. Alle Elemente sind dabei von *EModelElement* abgeleitet und können über

eine *EAnnotation* mit Kommentaren versehen werden. Außer *EAnnotation* erben alle anderen Elemente von *ENamedElement*, womit sie über einen Namen verfügen. Das *EPackage* Element repräsentiert ein Paket, bestehend aus mehreren *EClassifier*-Elementen. Ein *EClassifier* steht entweder für einen *EDataType* oder eine *EClass*. Erstere sind primitive oder komplexe Datentypen. Letztere repräsentieren Klassen, die aus einer Menge von Methoden (*EOperationen*), Attributen (*EAttribute*) und Referenzen (*EReference*) bestehen und von anderen Klassen abgeleitet sein können (*eSuperTypes*).

Pakete können zudem beliebig viele Unterpakete besitzen, die ebenfalls wieder Unterpakete besitzen können, wodurch eine Pakethierarchie abgebildet wird. Als Pakethierarchie wird im Folgenden die Beziehung *Enthalten-In* der Elemente bezeichnet, bei der Attribute und Methoden in Klassen, Klassen in Paketen und Pakete in Oberpaketen enthalten sind.

Die Instanz eines *Ecore*-Modells, genannt *Core*-Modell, besitzt in EMF die Endung *.ecore* und wird in einer XML-basierten Repräsentation, dem *XML Metadata Interachange*-Format, gespeichert.

4.2.5 Xtend Modell-zu-Modell-Transformationen

Xtend ist als funktionale Sprache definiert, bei der alle Operationen durch Funktionen ausgeführt werden. Diese Funktionen werden als Erweiterungen (engl. *extension*) bezeichnet und sind in Dateien mit der Endung *.ext zusammengefasst. Bevor die Erweiterungen im Detail betrachtet werden können, werden im folgenden Abschnitt zunächst das Typsystem und die Ausdruckssprache der Sprachfamilie *Check*, *Xtend* und *Xpand* vorgestellt werden.

Typsystem und Ausdruckssprache

Das Typsystem besteht aus einer Menge von Basistypen und erlaubt es, zusätzliche Typen selbst zu definieren. Zu den *einfachen* Basistypen zählen *String*, *Boolean*, *Integer* und *Real*. Für allgemeine Objekte existiert der Typ *Object* und für Ausdrücke ohne Rückgabewert der Typ *Void*. Die Typen für Mengen lassen sich in *Collection*, *List* und *Set* unterteilen. Zusätzliche Typen können über ein Metamodell importiert werden. So können beispielsweise Typdefinitionen aus einem XML-Schema in *Xtend* genutzt werden, womit zum Beispiel auch komplexe Typen mit Unterelementen möglich sind.

Aufbauend auf dem Typsystem können verschiedene Ausdrücke definiert werden. Die Syntax ähnelt dabei der Programmiersprache Java und der in der UML verwendeten *Object Constraint Language* der OMG. Die Ausdruckssprache verfügt über eine Reihe von Operatoren, Kontrollstrukturen sowie spezielle Mengenoperationen.

Zu den Operatoren zählen zum Beispiel die arithmetischen Operatoren für Addition, Subtraktion, Multiplikation, Division und Modulo (+, -, *, /, %), die logischen Operatoren UND, ODER und Negation (&&, ||, !), sowie die Vergleichsoperatoren (==, !=, >, >=, <, <=).

Zu den Kontrollstrukturen gehören Verzweigung und Mehrfachverzweigung, jedoch keine Schleifen. Diese müssen, falls benötigt, über Rekursion abgebildet werden. Eine Verzweigung kann über die Syntax *if-then-else* oder über den ternären Operator (*Bedingung ? Wahr : Falsch*) realisiert werden. Die Mehrfachverzweigung wird mit dem Schlüsselwort *switch* und einem in Klammern folgenden Ausdruck eingeleitet. Die einzelnen Fälle werden mit dem Schlüsselwort *case* unterschieden, wobei zusätzlich ein *default* Fall definiert werden muss. In Listing 2 sind drei einfache Beispiele für die Kontrollstrukturen unter Verwendung einiger Operatoren abgebildet.

```
1   //Verzweigung if-then-else
2   if (a > b && a != 0)        then a * 2        else a + b
3   //Verzweigung tenärer Operator
4   ( a == 0 && b == 0) ?       0                 : a * b + 1
5
6   //Mehrfachverzweigung switch
7   switch (type) {
8           case "1" : "Box"
9           case "2" : "Zylinder"
10          case "3" : "Kegel"
11          default :   "Box"
12  }
```

Listing 2: Xtend-Ausdruckssprache: Kontrollstrukturen

Eine Menge von Elementen wird in geschweiften Klammern und mit Kommas separiert definiert, zum Beispiel *{1, 2, 3}*. Neben den üblichen Funktionen zum Hinzufügen und Entfernen von Elemente können auf Mengen auch spezielle Operationen ausgeführt werden. Der Funktionsaufruf erfolgt dabei durch den Punktoperator, wie in Listing 3 gezeigt.

```
1  //Teilmengen
2  {1, 2, 3, 4, 5}.select( i | i % 2 == 0)     //{2, 4}
3  {1, 2, 3, 4, 5}.reject( i | i < 4)          //{4, 5}
4
5  //Listen bilden
6  personenMenge.collect( person | person.name)   //Eine Liste mit Namen der Personen
7  personenMenge.name                             //Äquivalent zum oberen Beispiel
8
9  //Sortieren
10 personenMenge.sortBy( person | person.vorname)         //Personen nach Vornamen sortiert
11
12 //Überprüfen
13 {"tomas",  "martin", "anna"}.forAll( name | name.length > 4)        //False
14 {"tomas",  "martin", "anna"}.exists( name | name=="tomas")          //True
```

Listing 3: Xtend-Ausdruckssprache: Mengen und Mengenoperationen

In Zeile 3 wird mit *.reject(i | i < 4)* eine Operation zum Verwerfen (engl. *reject*) mehrerer Elemente aufgerufen. Der in Klammern stehende Ausdruck ist wie folgt zu interpretieren: Wähle alle Elemente i, für die i < 4 gilt. Die so gefilterte Menge von Elementen wird entsprechend der Operation verarbeitet. Es werden also alle Elemente der Menge verworfen, die kleiner sind als vier. Die zweite Verwendung des Punktoperators besteht darin, Unterelemente eines Objekts auszuwählen. Der Ausdruck *person.name* aus Zeile 6 gibt also den Namen der Person und nicht die Person selbst zurück. Wird ein Aufruf eines Unterelements auf eine Menge, wie in Zeile 7, angewendet, gibt der Ausdruck eine Liste mit den Unterelementen zurück.

Damit nicht für jeden Ausdruck eine eigene Erweiterung geschrieben werden muss, können Ausdrücke verkettet werden. Dies geschieht mit mit dem Verkettungsoperator ->, wobei das Ergebnis des letzten Teilausdrucks als Ergebnis des gesamten Ausdrucks zurückgegeben wird. Listing 4 zeigt exemplarisch einen verketteten Ausdruck.

```
1  //Verkettung
2  let titel = "Dr." :
3  person.setName("Mann")                 ->
4  person.setVorname(titel + " Thomas")  ->
5  person
```

Listing 4: Xtend-Ausdruckssprache: Verketteter Ausdruck

Auf dem Objekt *person* werden die zwei Funktionen zum Setzen (engl. *set*) des Namens und des Vornamens ausgeführt. In Zeile 5 wird das Objekt dann als Ergebnis des Gesamtausdrucks zurückgegeben. Das Schlüsselwort *let* aus Zeile 2 definiert eine lokale

Variable, die anschließend innerhalb eines Ausdrucks – wie in Zeile 4 – genutzt werden kann.

Erweiterungen

Der Aufbau einer Erweiterung orientiert sich am üblichen Aufbau einer Funktion in einer Programmiersprache wie Java. Sie besteht aus einem Funktionskopf und einem Funktionskörper.

Der Kopf setzt sich aus einem Rückgabetyp, dem Funktionsnamen und der Parameterliste zusammen. Die Angabe des Rückgabetyps ist optional, da dieser, außer für rekursive Funktionsaufrufe, automatisch ermittelt werden kann.

Im Funktionskörper werden die Parameter entsprechend des enthaltenen Ausdrucks verarbeitet. Eine Erweiterung kann entweder über einen üblichen Aufruf mit Funktionsname und Parameterliste erfolgen oder direkt auf einem Objekt aufgerufen werden. Im zweiten Fall, der sogenannten *member syntax*, wird das Objekt als erster Parameter der Funktion übergeben. Eine weitere Besonderheit stellt der Funktionsaufruf auf einer Menge dar. Ist die Funktion nur für ein einzelnes Objekte, nicht aber für eine Menge der Objekte definiert, so wird die Funktion für jedes Objekt der Menge separat aufgerufen. Listing 5 zeigt die Definition einer Funktion (Zeile 2 & 3) und die drei Möglichkeiten zum Aufruf (Zeile 6, 8 und 10).

```
1  //Definition Erweiterung
2  cached String getMetadataValue(ElementTyp element, String metadataName):
3          element.metadata.select( metadata | metadata.name ==  metadataName).value;
4
5  //Aufruf der Erweiterung
6  getMetadataValue(einElement, "titel")                //gibt den Titel zurück
7  //Aufruf der Erweiterung in member syntax
8  einElement.getMetadataValue("titel")         //gibt ebenfalls den Titel zurück
9  //Aufruf der Erweiterung auf einer Menge mit member syntax
10 ElementMenge.getMetadataValue("titel")       //gibt eine Liste von Titeln zurück
```

Listing 5: Xtend-Erweiterungen: Definition und Aufruf

Das Schlüsselwort *cached* aus Zeile 2 gibt an, dass die Ergebnisse der Funktionsaufrufe gepuffert werden. Wird eine Funktion mehrmals mit den gleichen Parametern aufgerufen, so wird der Ausdruck nicht erneut verarbeitet, sondern das gepufferte Ergebnis zurückgegeben. Dadurch kann eine erhebliche Performanzsteigerung erzielt werden.

Transformationen

Für Transformationen eines Modells müssen neben dem Ändern von Werten und dem Entfernen von Elementen auch neue hinzugefügt werden. Dies kann entweder mit dem

Schlüsselwort *new* geschehen oder indem der Funktion das Schlüsselwort *create* vorangestellt wird. In Listing 6 sind beide Varianten mit einem Beispiel beschrieben.

```
1  import personenMetaModell;
2  extension personen::geschlecht;
3
4  //Erstellen eines Objektes mit new
5  PersonTyp createPerson(String name, String vorname):
6          let person = new Persontyp:
7          person.setName(name)->
8          person.setVorname(vorname)->
9          person.setGeschlecht(vorname.getGeschlecht())->
10         person;
11
12 //Erstellen eines Objektes mit create
13 create PersonTyp createPerson(String name, String vorname):
14         setName(name)->
15         setVorname(vorname)->
16         setGeschlecht(vorname.getGeschlecht());
```

Listing 6: Xtend-Erweiterungen: Erstellen von Objekten

Bei der ersten Variante in Zeile 6 wird mit *new* explizit ein neues Objekt des Typs *Persontyp* angelegt. Mit *create* wird dagegen bereits beim Aufruf der Funktion *createPerson* in Zeile 13 automatisch eine Person erstellt, die im Funktionskörper initialisiert und als Ergebnis zurückgegeben wird. Ein weiterer wichtiger Unterschied zur ersten Variante ist, dass nicht zwingend ein neues Objekt erzeugt wird. Ähnlich wie bei *cached* werden die Ergebnisse der Funktionsaufrufe gepuffert. Wird die Funktion mehrfach mit den gleichen Parametern aufgerufen, so wird also lediglich die Referenz auf das bereits erstellte Objekt zurückgegeben. Dabei müssen allerdings zwei Dinge beachtet werden, die sonst schnell zu Problemen führen können. Zum einen wird der Ausdruck innerhalb der Funktion nicht noch einmal verarbeitet, so dass zum Beispiel der Aufruf einer Funktion auf einem Objekt der Parameterliste nicht ausgeführt wird. Zum anderen wird bei einem Objekt als Parameter nicht dessen innerer Zustand berücksichtigt, sondern lediglich die Referenz. Somit kann es zum unerwünschten Erhalt einer gepufferten Referenz an Stelle eines neuen Objekts kommen.

Die Anweisung *import* aus Zeile 1 importiert ein Metamodell. Somit können die Typen des Metamodells, wie im Beispiel der *PersonTyp*, in den Erweiterungen verwendet werden. Neben Ecore-Metamodellen bietet oAW auch die Möglichkeit, Metamodelle auf Basis von XML-Schema zu importieren.

Sollen Erweiterungen aus anderen Erweiterungsdateien verwendet werden, müssen diese wie in Zeile 2 mit dem Schlüsselwort *extension* und dem vollständigen Java-Klassenpfad importiert werden. Im Beispiel wird die Erweiterung *getGeschlecht(String vorname)* importiert, die aus dem Vornamen das Geschlecht ableitet.

4.3 Extensible 3D

Extensible 3D (X3D) ist ein frei verfügbarer Standard für die Beschreibung und Darstellung dreidimensionaler Szenen und Objekte. Er wird vom Web 3D-Konsortium verwaltet und von mehreren Arbeitsgruppen stetig weiterentwickelt. In Folge einer ausführlichen Diskussion hat Müller dieses Format unter Berücksichtigung der Aspekte Plattformunabhängigkeit, Anwendungsunabhängigkeit und Standardisierung als ein ideales Trägerformat für eine Softwarevisualisierung identifiziert und in Folge dessen als Zielformat für seinen Generator gewählt (R. Müller 2009).

X3D ist der offizielle Nachfolger der *Virtual Reality Modeling Language* (VRML), die ebenfalls vom Web 3D-Konsortium entwickelt wurde. VRML wird im Zuge dieser Arbeit nur am Rande betrachtet, da es lediglich als Importformat für das Virtual Reality-Labor der Universität Leipzig genutzt wird. Die Konvertierung von X3D in VRML erfolgt mittels eines XSLT *Stylesheets*[6].

In Kapitel 4.3.1 werden die Grundlagen des Standards kurz umrissen. Den Kern bildet eine auf XML basierende Sprache zur Modellierung eines Szenegraphen, der die dreidimensionale Szene mit verschiedenen Elementen beschreibt. Der Aufbau des Szenegraphen wird in Kapitel 4.3.2 beschrieben. Neben der geometrischen Darstellung von Objekten können auch Animationen und Interaktionen beschrieben werden. Die entsprechenden Grundlagen vermittelt das Kapitel 4.3.3. Die Elemente eines Szenegraphen können auch durch eigene Elemente – genannt *Prototypen* – ergänzt werden. Das Erstellen von Prototypen wird in Kapitel 4.3.4 ausführlicher betrachtet.

4.3.1 Grundlagen

Die Spezifikation des X3D-Standards besteht aus drei Teilen, die alle von der ISO und der IEC ratifiziert wurden (ISO/IEC 19775, 19776, 19777). Im ersten Teil werden die Architektur, die Basiskomponenten, sowie die Schnittstelle für den Zugriff auf die Szene beschrieben. Alle Ausführungen sind dabei unabhängig von einer konkreten Technologie. Der zweite Teil definiert die drei verwendeten Dateiformate. Das erste Format (Endung *.x3d) nutzt XML als Grundlage, das zweite (Endung *.x3dv) setzt die Syntax des Vorgängers VRML ein. Neben diesen beiden Klartextformaten existiert noch das binäre

6 Extensible Stylesheet Language Transformation http://www.w3.org/TR/xslt20/

Format (Endung *.x3db). Im dritten Teil wird die Sprachanbindung für ECMA-Scripte (*European Computer Manufacturers Association*) und Java beschrieben. Die Sprachen werden für das Erstellen von Skripten für Ereignisse, Animation und Interaktion eingesetzt.

Die gesamte Spezifikation ist in vier Profile unterteilt, die sich aus unterschiedlichen Komponenten zusammen setzen. Diese Profile sind hierarchisch geordnet, so dass ein Profil auf das andere aufsetzt und dieses erweitert. Das kleinste ist das *Interchange* Profil, welches nur grundlegende geometrische Elemente, wie die Primitiven oder Texturen enthält. Danach folgen *Interactive*, *Immersive* und *Full*. Zusätzlich zu diesen vier Profilen gibt es noch das *MPEG-4 Interactive* und das *CDF* Profil.

Die Komponenten eines Profil bestehen aus einer Menge von Knoten (engl. *nodes*) , die die Funktionalitäten der Komponente realisieren. Tabelle 7 gibt einen Überblick über einige Komponenten und deren zugehörige Knoten, die im Prototyp verwendet wurden.

Komponente	**Beschreibung**	**Knotenbeispiele**
Geometry3D	Dreidimensionale geometrische Primitive	Box, Cone, Cylinder, Sphere
Shape	Definiert das Erscheinungsbild der geometrischen Primitiven	Shape, Appearance, Material
Interpolation	Veränderung von Eigenschaften für Animationen	ScalarInterpolator, ColorInterpolator

Tabelle 7: X3D-Komponenten: Beispiele

Die Eigenschaften eines Knotens können entweder über die sogenannten *Felder* oder über Unterknoten bestimmt werden. Felder werden als XML-Attribute gesetzt und müssen einem definierten Typ, wie *Integer* oder *Float*, genügen. Unterknoten werden unter Verwendung der XML-Syntax entsprechend der Spezifikation des Knotentyps in den Knoten geschachtelt.

Die Darstellung des Szenegraphen (engl. *rendering*) übernimmt ein sogenannter X3D-Browser. Dieser liest die X3D-Datei ein, verarbeitet diese und rendert entsprechend der Direktiven die einzelnen geometrischen Elemente. Zudem realisiert er das Ereignismodell, mit dem die Elemente der Szene manipuliert werden. Es existiert eine Vielzahl von kommerziellen und frei erhältlichen X3D-Browsern als eigenständige Programme. Alternativ gibt es auch Plugins für bekannte Web-Browser, wie zum Beispiel den Mozilla Firefox oder den Internet Explorer. Im X3D-Wiki[7] des Web3D-Konsortiums findet sich eine Übersicht über die bekanntesten Browser sowie deren Unterstützung für die einzelnen Komponenten und Dateiformate. Der in dieser Arbeit vorgestellte Prototyp wurde mit dem

7 http://www.web3d.org/x3d/wiki/index.php/Player_support_for_X3D_components

BS Contact Player von BitManagement[8] in der Version 7.0 getestet. Dieser konnte allen Anforderungen an die Umsetzung der X3D-Spezifikation gerecht werden.
Da mit Hinblick auf den Umfang dieser Arbeit nur für den Prototypen relevante Teile von X3D vorgestellt werden können, soll an dieser Stelle für weitere Informationen auf (Brutzman & Daly 2007) verwiesen werden.

4.3.2 Szenegraph

Unter einem Szenegraph wird eine Baumstruktur verstanden, die alle Aspekte einer 3D-Szene in hierarchischer Form organisiert (Brutzman & Daly 2007).
Einer in X3D beschriebenen Szene liegen folgende Eigenschaften zugrunde:

- Dreidimensionales kartesisches Koordinatensystem
- Rotationen erfolgen entgegen dem Uhrzeigersinn
- Distanz wird in Meter, Winkel in Radiant und Zeit in Sekunden gemessen
- Farben werden als Vektor aus drei reellen Zahlen zwischen Null und Eins für rot, grün und blau dargestellt

Bevor der X3D-Szenegraph näher betrachtet wird, soll aber noch kurz auf die Struktur einer X3D-Datei eingegangen werden. Listing 7 zeigt den grundlegenden Aufbau. Wie alle anderen Listings zu X3D bezieht es sich auf das Dateiformat mit XML als Grundlage (Endung *.x3d).

```
1 <?xmlversion="1.0"encoding="UTF-8"?>
2 <X3D profile="Immersive" version="3.2">
3 	<Head>
4 		<Meta content='x3d Strukturbeispiel' name='title'/>
5 	</Head>
6 	<Scene>
7 		<!-- Scenegraph siehe Listing 8 -->
8 	</Scene>
9 </X3D>
```

Listing 7: X3D-Dateistruktur

Nach der für XML üblichen Deklaration in Zeile 1 folgt das X3D-Wurzelelement mit Angaben zum verwendeten Profil und der Version. Im optionalen Kopfbereich, eingeleitet durch das Element *Head*, können Metadaten zum Dokument abgelegt werden. Zusätzlich können extern definierte Prototypen im Kopfbereich deklariert werden. Der eigentliche Szenegraph mit den entsprechenden Knoten befindet sich innerhalb des Elements *Scene*.

8 http://www.bitmanagement.de

Abbildung 18 zeigt die Darstellung eines einfachen *Hello World*!-Beispiels mit einem blauen Würfel und einem Schriftzug in einem X3D-Browser. In Listing 8 ist der zugehörige Szenegraph angegeben und in Abbildung 19 wird dessen Struktur als Baum visualisiert.

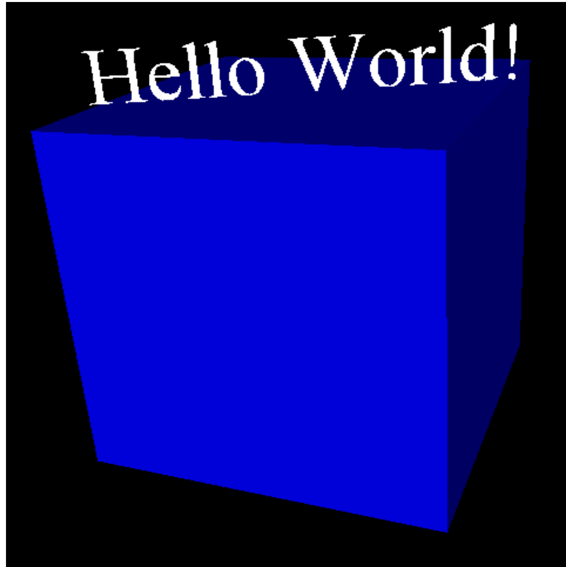

Abbildung 18: X3D-Beispiel: Hello World!

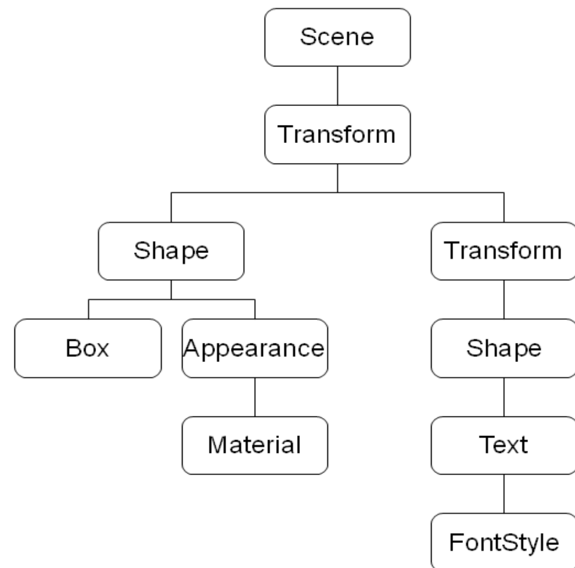

Abbildung 19: X3D-Szenegraph: Baumstruktur

```
1  <Scene>
2      <Transform scale="2 2 2">
3          <Shape>
4              <Box/>
5              <Appearance>
6                  <Material diffuseColor="0 0 1"/>
7              </Appearance>
8          </Shape>
9          <Transform scale="0.5 0.5 0.5" translation="0 1 0">
10             <Billboard>
11                 <Shape>
12                     <Text string='"Hello World!"'>
13                         <FontStyle justify='"MIDDLE"'/>
14                     </Text>
15                 </Shape>
16             </Billboard>
17         </Transform>
18     </Transform>
19 </Scene>
```

Listing 8: X3D-Szenegraph: Hello World!

Ein *Transform*-Knoten, wie in Zeile 2, transformiert alle seine Unterelemente. Mit *scale* werden die Elemente vergrößert beziehungsweise verkleinert. Der Wert des Feldes setzt

sich aus drei reellen Zahlen für x, y und z zusammen. Im Urzustand besitzt ein Knoten immer die Größe von $1m^3$. In Zeile 2 werden somit die enthaltenen Elemente um den Faktor Zwei vergrößert (*scale="2 2 2"*), womit sie nun 2m*2m*2m, also $8 m^3$, groß sind.
Der *Transform*-Knoten aus Zeile 9 verschiebt mit *translation* alle seine Unterelemente um einen Meter in y-Richtung (*translation="0 1 0"*). Da sich dieser aber unterhalb des ersten *Transform*-Knotens befindet, werden die Elemente um zwei Meter verschoben. Der Knotentyp *Shape* definiert einen Körper in der Szene, wobei die Art des Körpers über einen Unterknoten näher bestimmt wird. Die Zeilen 3-8 definieren demzufolge den Würfel mit dem Knotentyp *Box* (Zeile 4). Die Zeilen 11-15 erstellen hingegen den Schriftzug mit dem Knotentyp *Text* (Zeile 12-14). Dieser enthält zusätzlich einen Knoten *FontStyle* zum Setzen der Schriftart. Um die Erscheinung eines Körpers zu bestimmen, wird der Knotentyp *Appearance*, wie in Zeile 5, genutzt. Dieser enthält die entsprechenden Unterknoten, mit denen zum Beispiel Material, Textur oder Schatten bestimmt werden können. Im Beispiel wird die Farbe des Würfels über den Knoten *Material* und das Feld *diffuseColor* auf Blau (0 0 1) gesetzt (Zeile 6). Der Knotentyp *Billboard* aus Zeile 10 bewirkt, dass sich alle Unterelemente mit der Blickrichtung des Betrachters mitdrehen. Andernfalls wäre die Schrift in Abbildung 18 genauso gekippt wie der Würfel.

4.3.3 Ereignismodell

Für die Realisierung von Animationen und Interaktionen in X3D steht ein simples, aber mächtiges Ereignismodell zur Verfügung. Ein Knoten kann über seine Felder Ereignisse auslösen oder empfangen, im Folgenden als ausgehende und eingehende Felder bezeichnet. Ein Ereignis ist demnach die Übertragung des Werts von einem ausgehenden Feld an ein eingehendes Feld. Beide Felder müssen dafür den gleichen Typ besitzen. Die Kommunikationswege zwischen den Knoten werden durch Routen abgebildet. Jede Route verbindet dabei ein ausgehendes und ein eingehendes Feld. Für die weiteren Betrachtungen werden die Knoten in vier Gruppen eingeteilt, abhängig von ihrer Funktion im Ereignismodell.
Die erste Gruppe bilden die *Sensoren*, die von der Laufzeitumgebung des X3D-Browsers aktiviert werden und daraufhin ein Ereignis auslösen. Dazu zählt zum Beispiel der *TouchSensor*, der immer mit einem geometrischen Knoten verbunden ist und vom Benutzer durch einen Mausklick aktiviert wird (Komponente: *Pointing Device Sensor*). Ein weiteres Beispiel ist der *TimeSensor*, der keine geometrische Darstellung besitzt und kontinuierlich im Verlauf der Zeit Ereignisse auslöst (Komponente: *Time*).

Die zweite Gruppe bilden die sogenannten *Interpolatoren*, die bei einem eingehenden Ereignis den Wert verarbeiten und selbst ein Ereignis auslösen (Komponente: *Interpolation*). Ein *Interpolator* wird normalerweise genutzt, um die Darstellung eines Knotens zu verändern. Ein Beispiel ist der *ColorInterpolator*, der die Farbe eines Knotens ändern kann. Dazu muss sein ausgehendes Feld über eine Route mit dem *diffuseColor* Feld eines *Material* Knotens verbunden werden. Ein *Interpolator* empfängt Ereignisse direkt von einem Sensor, einem anderen *Interpolator* oder einem *Ereignishelfer*, welcher zur dritten Gruppe im Ereignismodell gezählt wird.

Die Knoten dieser Gruppe (Komponente: *Event utilities*) verarbeiten Ereignisse und geben das Ergebnis an andere Knoten weiter. Mit einem *BooleanFilter* kann zum Beispiel eine Fallunterscheidung für ein Ereignis vom Typ *Bool* erstellt werden. Dieser löst dann entweder ein neues Ereignis für *true* oder eines für *false* aus, in Abhängigkeit vom Wert des empfangenen Ereignisses.

Alle Knoten, die überwiegend oder ausschließlich nur Ereignisse empfangen ohne eigene auszulösen, bilden die letzte Gruppe. Zu diesen zählen beispielsweise die bereits vorgestellten Knotentypen *Shape*, *Box* und *Transform*. Abbildung 20 zeigt das Ereignismodell im Überblick.

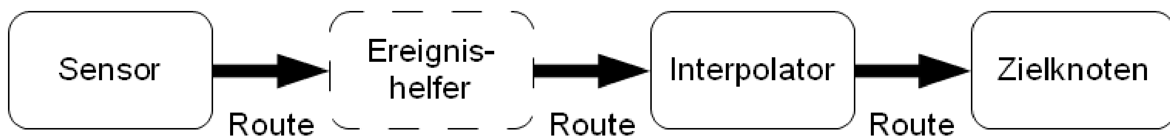

Abbildung 20: X3D-Ereignismodell: Überblick

Im Folgenden soll ein Beispiel für eine Animation unter Verwendung des Ereignismodells gegeben werden. Abbildung 21 zeigt eine Kugel, deren Farbe im Lauf der Zeit von grün über blau zu lila wechselt. In Listing 9 ist der zugehörige Szenegraph beschrieben.

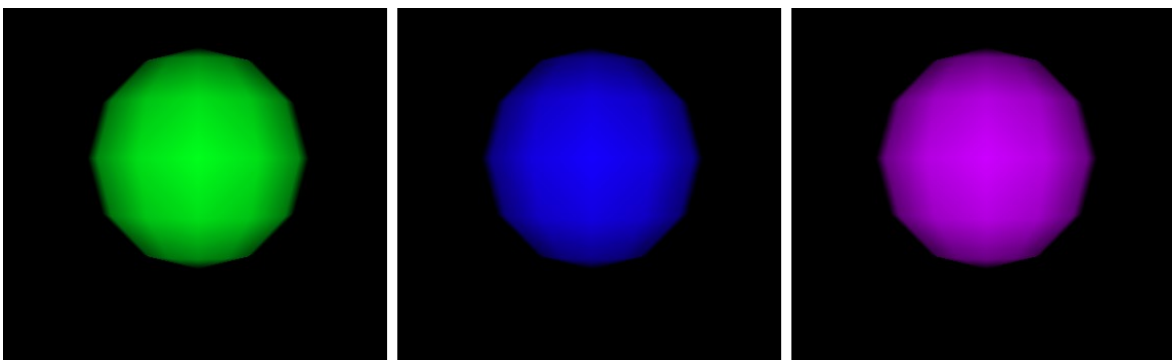

Abbildung 21: X3D-Ereignismodell: Beispiel Farbwechsel nach Zeit

```
1  <Scene>
2      <Transform>
3          <Shape>
4              <Sphere/>
5              <Appearance>
6                  <Material DEF="Material" diffuseColor="0 0 1"/>
7              </Appearance>
8          </Shape>
9      </Transform>
10
11     <TimeSensor DEF="TimeSensor" cycleInterval="10" loop="true"/>
12     <ColorInterpolator DEF="ColorInterpolator" key="0, 1" keyValue="0 1 0, 1 0 1"/>
13     <ROUTE      fromNode='TimeSensor' fromField='fraction_changed'
                   toNode='ColorInterpolator' toField='set_fraction'/>
14     <ROUTE      fromNode='ColorInterpolator' fromField='value_changed'
                   toNode='Material' toField='diffuseColor'/>
15 </Scene>
```

Listing 9: X3D-Ereignismodell: Beispiel Szenegraph

In Zeile 11 wird ein *TimeSensor* erstellt und mit einem Zehn-Sekunden-Intervall für den Zyklus (engl. *cycleInterval*) initialisiert. Innerhalb dieses Intervalls erzeugt er somit kontinuierlich Ereignisse mit aufsteigenden Werten von 0 bis 1. Durch das Feld *loop* beginnt der Sensor nach einem Intervall von neuem bei 0. Der *ColorInterpolator* (Zeile 12) wird mit zwei Feldern – *key* und *keyValue* – erstellt. Jeder in *key* definierte Wert wird auf den entsprechenden Wert in *keyValue* gemappt. Empfängt der *Interpolator* beispielsweise ein Ereignis mit dem Wert *0*, so löst er ein Ereignis mit dem Wert *0 1 0* (grün) aus. Bei einem Wert von *1*, dementsprechend *1 0 1* (lila). Liegt der empfangene Wert zwischen zwei definierten Werten für *key*, so wird der Wert des ausgehenden Ereignis interpoliert. Es entsteht ein kontinuierlicher Farbverlauf, was auch die blaue Farbe in Abbildung 21 nach ca. fünf Sekunden erklärt. Zuletzt werden in den Zeilen 13 und 14 die beiden Routen erstellt. Dafür müssen den Knoten und Feldern eindeutige Bezeichner zugewiesen werden, was durch das Feld DEF, wie zum Beispiel in Zeile 6, geschieht. Diese können dann in den vier Feldern der Route - *fromNode*, *fromField*, *toNode* und *toField* – verwendet werden. Abbildung 22 zeigt Knoten, Felder und Routen im Überblick.

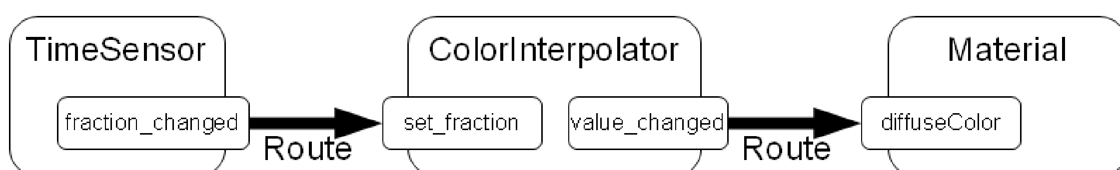

Abbildung 22: X3D-Ereignismodell: Beispiel Überblick

4.3.4 X3D-Prototypen

Wie bereits erwähnt können in X3D auch eigene Knotentypen definiert werden, die als X3D-Prototypen bezeichnet werden. Ein X3D-Prototyp ist oft einfach eine Kombination vorhandener Knotentypen. Wie alle anderen Knoten auch kann er mit Feldern für Eigenschaften versehen werden und Unterknoten enthalten. In Kombination mit dem Ereignismodell können auf diese Weise mächtige Erweiterungen erstellt werden. Aufbau und Funktionsweise eines solchen X3D-Prototyps sollen anhand eines Beispiels erläutert werden. Dazu wird ein Knotentyp *BoolToFloat* definiert, der eingehende Ereignisse vom Typ *Bool* (*true*, *false*) in ausgehende Ereignisse vom Typ *Float* (*1*, *0*) umwandelt. Im Ereignismodell würde ein solcher Knoten die Aufgabe eines Ereignishelfers übernehmen, indem er zum Beispiel zwischen einen *TouchSensor* und einen *ColorInterpolator* geschaltet wird, wie in Abbildung 23 gezeigt. Ein Klick auf die Kugel ändert die Farbe von gelb zu rot.

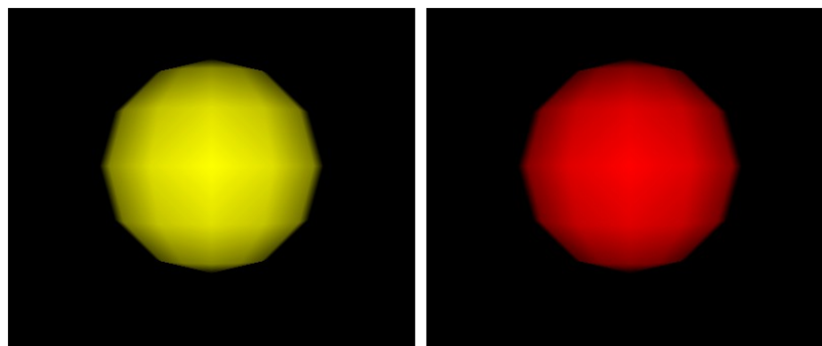

Abbildung 23: X3D-Prototyp: Beispiel Farbwechsel nach Berührung

Abbildung 24 zeigt die verwendeten Knoten und Routen in einer Übersicht. Quell- und Zielknoten sind dabei identisch.

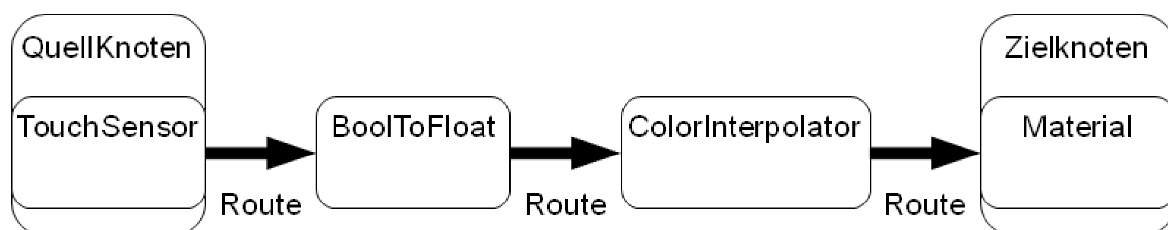

Abbildung 24: X3D-Prototyp: Beispiel Knoteüberblick

Die Definition und die Instantiierung des *BoolToFloat*-X3D-Prototyps werden in Listing 10 gezeigt.

```
1  <!-- Definition X3D-Prototyp -->
2  <ProtoDeclare name="BoolToFloat">
3      <ProtoInterface>
4          <field accessType="inputOnly" name="set_bool" type="SFBool"/>
5          <field accessType="outputOnly" name="get_float" type="SFFloat"/>
6      </ProtoInterface>
7      <ProtoBody>
8          <Script>
9              <!-- ECMA-Skript siehe Listing 11 -->
10         </Script>
11     </ProtoBody>
12 </ProtoDeclare>
13
14 <!-- Erstellen einer Instanz des X3D-Prototyps -->
15 <ProtoInstance name="BoolToFloat" DEF="myBoolToFloat" />
```

Listing 10: X3D-Prototyp: Beispiel Definition

Ein X3D-Prototyp wird innerhalb des *Scene*-Element mit einem *ProtoDeclare*-Knoten definiert (Zeile 2-12) und mit dem Knoten *ProtoInstance* (Zeile 15) instantiiert. Die Identifizierung erfolgt über einen eindeutigen Namen, der über das Feld *name* definiert wird (Zeile 2 beziehungsweise 15).

Die Definition besteht aus einem optionalen *ProtoInterface* (Zeile 3-6) und einem *ProtoBody* (Zeile 7-11). Das *ProtoInterface* beschreibt die Felder, die der X3D-Prototyp besitzt, mit Unterknoten vom Typ *field*. Jeder *field*-Knoten spezifiziert ein Feld mit Namen (engl. *name*), Typ (engl. *type*), optionalem Standardwert (engl. *value*) und einer Zugriffsart (engl. *accessType*). X3D definiert vier Zugriffsarten für Felder, die auch deren Eignung als ausgehendes oder eingehendes Feld für Ereignisse bestimmen. In Tabelle 8 ist dieser Zusammenhang dargestellt.

Zugriffsart	Eingehendes Feld	Ausgehendes Feld
inputOnly	X	
outputOnly		X
inputOutput	X	X
initializeOnly		

Tabelle 8: X3D-Felder: Zugriffsarten

Für die Zugriffsart *initializeOnly* existiert keine Ereignisverarbeitung. Der Wert des Feldes wird nur einmal aus dem Szenegraph gelesen und dann nicht wieder verändert.

Der *ProtoBody* beschreibt den Inhalt des Knotentyps. Er kann zum Beispiel eine Kombination von geometrischen Figuren enthalten, deren Darstellung mit den Feldern des

X3D-Prototyps parametrisiert wird. Da für eine Bool-zu-Float Konvertierung allerdings keine geometrische Darstellung nötig ist, enthält Listing 10 auch nur einen *Script* Knoten. Dessen Inhalt ist in Listing 11 gesondert aufgeführt.

```
1  <Script>
2      <field accessType="inputOnly" name="set_bool" type="SFBool"/>
3      <field accessType="outputOnly" name="get_float" type="SFFloat"/>
4      <IS>
5          <connect nodeField="set_bool" protoField="set_bool"/>
6          <connect nodeField="get_float" protoField="get_float"/>
7      </IS>
8      <![CDATA[ecmascript:
9          function set_bool(value, timeStamp){
10             Browser.print("BoolToFloat set_bool " + value);
11             if(value)
12                 get_float = 1;
13             else
14                 get_float = 0;
15         }
16     ]]>
17 </Script>
```

Listing 11: X3D-Prototyp: Beispiel ECMAScript

Mit *Script*-Knoten wird das Verhalten eines X3D-Prototyps definiert. In den Zeilen 8 und 9 werden mit dem *field*-Knotentyp zwei eigene Felder für das Skript definiert. Im anschließenden *IS*-Block werden diese mit den Feldern des X3D-Prototypen über *connect*-Knoten verbunden (Zeile 5 und 6). Durch diese Bindung kann ein empfangenes Ereignis in einer Funktion des Skripts behandelt oder ein eigenes Ereignis ausgelöst werden. In den Zeilen 8 bis 16 folgt das eigentliche Skript in der Sprache *ECMAScript*. Alle in dieser Arbeit verwendeten X3D-Prototyp-Skripte sind in dieser Sprache verfasst, die auch den Sprachkern von *JavaScript* darstellt. Durch die Bindung aus Zeile 5 wird für jedes empfangene Ereignis *set_bool* die Funktion *set_bool* des Skripts aufgerufen (Zeile 9). Der Wert des Ereignisses wird im ersten Parameter (*value*) übergeben. In der darauf folgenden Zeile wird der Funktionsaufruf für das Debugging protokolliert. In den Zeilen 11 bis 14 wird dann der Wert des Ereignisses verarbeitet. Ist dieser *true*, so wird ein Ereignis *get_float* mit dem Wert 1 ausgelöst, andernfalls eines mit dem Wert 0 (Zeile 12 beziehungsweise 14).

Nachdem nun alle technischen Grundlagen des Prototyps vorgestellt und erläutert wurden, folgt im nächsten Kapitel die Beschreibung der von Müller gelegten Grundlage (R. Müller 2009).

5 Basisprototyp

In diesem Kapitel wird der von Müller unter Anwendung des Generativen Paradigma (Czarnecki & Eisenecker 2000) entwickelte Generator vorgestellt (R. Müller 2009), der im Folgenden als Basisprototyp bezeichnet wird. Er bildet die Grundlage für die Umsetzung des Konzepts zur Interaktion mit einer Softwarevisualisierung der Struktur. Der Basisprototyp generiert aus den Strukturinformationen eines Ecore-Modells eine dreidimensionale Softwarevisualisierung der Struktur in X3D-Format. Als Visualisierungstechnik wird eine abstrakte visuelle Metapher verwendet. Die Schritte der Visualisierungspipeline realisiert der Generator durch Modell-zu-Modell-Transformationen des oAW-Frameworks. Für die Integration in den Entwicklungsprozess ist der Basisprototyp als Eclipse-Plugin konzipiert. Im ersten Unterkapitel soll die Funktionsweise des Generators anhand eines Beispiels näher erläutert werden. Das zweite Unterkapitel ordnet die generierte Softwarevisualisierung anhand der in Kapitel 2.6 vorgestellten Taxonomie ein. Der Aufbau des Generators, inklusive Auszügen der Implementierung, wird anschließend in Kapitel 5.3 beschrieben. Im letzten Unterkapitel werden die Erweiterungspunkte für die Implementierung einer Benutzungsschnittstelle identifiziert und beschrieben.

5.1 Funktionsweise

Zur Demonstration der Funktionsweise wird ein einfaches Beispiel verwendet, welches für die Entwicklung des Basisprototypen von Müller erstellt wurde und als *familytree*-Beispiel bezeichnet wird. Den Ausgangspunkt bildet, wie bereits erwähnt, ein Ecore-Modell, das die benötigten Strukturinformationen der Software, wie Pakete und Klassen, enthält. Die Quelle des Ecore-Modells ist nicht näher bestimmt, so dass es beispielsweise aus annotiertem Java Quellcode abgeleitet, mit einem zusätzlichen Werkzeug modelliert oder aus einem anderen Programm exportiert worden sein kann. Das Ecore-Modell des familytree-Beispiels wird in Abbildung 25 als UML-Klassendiagramm dargestellt. Auf eine Darstellung in XML-Format wird dagegen verzichtet, da diese nicht relevant für das Verständnis des Beispiels ist. Da das Beispiel lediglich zu Testzwecken konzipiert wurde, wird auf eine Semantik im Folgenden kein Bezug genommen.

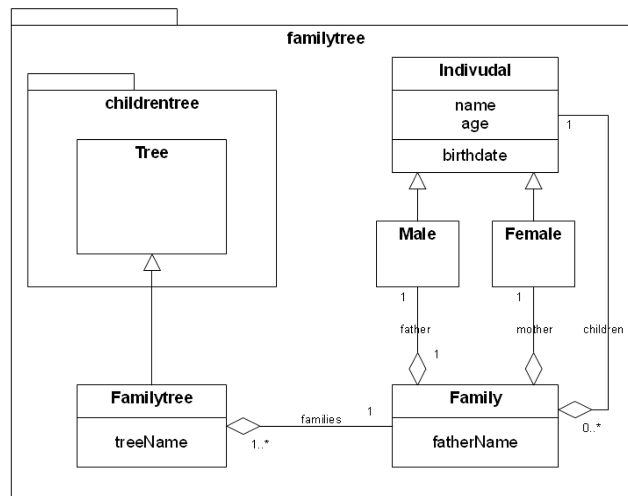

Abbildung 25: Klassendiagramm: Beispiel familytree

Wie dem Klassendiagramm zu entnehmen ist, besteht das Beispiel aus zwei Paketen *familytree* und *childrentree*. Das Paket *familytree* enthält die Klasse *Individual* mit den Attributen *name* und *age* sowie der Methode *birthdate*. Von *Individual* werden *Male* und *Female* als Spezialisierungen abgeleitet. Die Klasse *Family* setzt sich aus einem *father* vom Typ *Male* und einer *mother* vom Typ *Female* sowie keinen bis vielen *children* vom Typ *Individual* zusammen. Zusätzlich besitzt sie ein Attribut *fatherName*. Eine *Family* ist wiederum Bestandteil der Klasse *Familytree*, die sich aus einer bis vielen *families* vom Typ *Family* zusammensetzen kann und über ein Attribut *treeName* verfügt. Ein *Familytree* ist von der Klasse *Tree* des Pakets *childrentree* abgleitet.

Der Visualisierungsprozess des Generators wird über das Kontextmenü von Eclipse gestartet. Auf der linken Seite in Abbildung 26 ist dieser Aufruf des Generators für das *familytree*-Beispiel abgebildet.

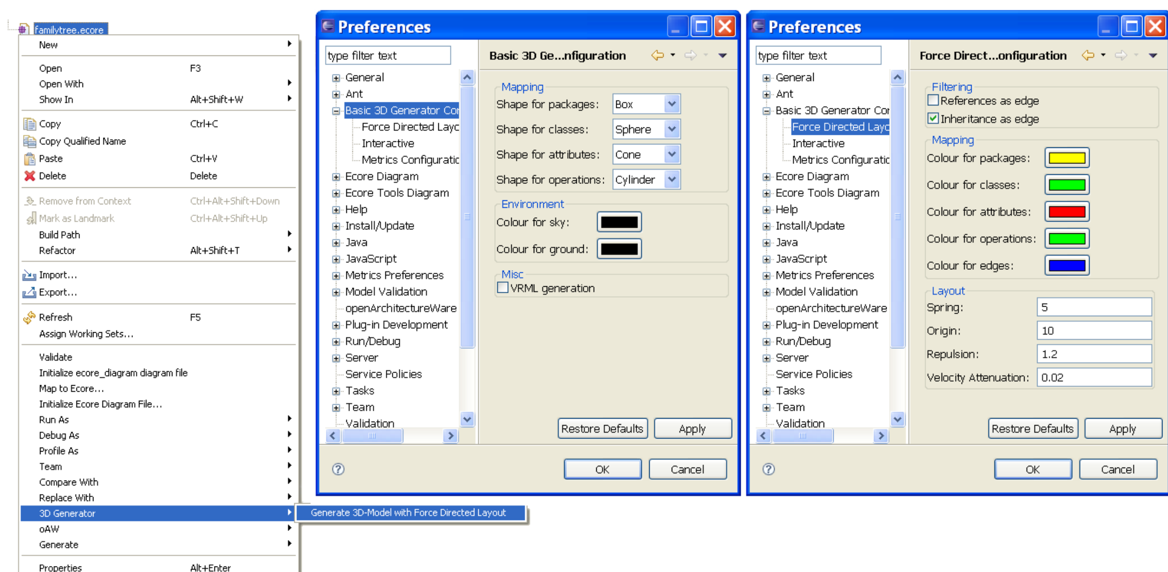

Abbildung 26: Eclipse-Integration: Basisprototyp

Die Konfiguration des Generators erfolgt über die Einstellungsseiten von Eclipse, die auf der rechten Seite der Abbildung 26 dargestellt sind. Im Zuge der Weiterentwicklung des Prototypen wurde die von Müller konzipierte Eigenschaftsseite auf mehrere Seiten umgebrochen und durch weitere ergänzt. Die in der Abbildung dargestellten Seiten entsprechen somit nicht dem Originalzustand, stellen aber die gleichen Elemente für die Konfiguration des Generators zur Verfügung. Die erste Seite enthält die generellen Einstellungen, wie die Form der Elemente, die Farbe des Hintergrunds sowie eine Check-Box für das zusätzliche Erstellen einer VRML-Version, die unter anderem für das Virtual Reality-Labor der Universität Leipzig benötigt wird. Auf der zweiten Seite werden die spezifischen Eigenschaften für die von Müller gewählte abstrakte Metapher eingestellt. Im ersten Bereich werden dazu die zu visualisierenden Beziehungsarten – Vererbung, Assoziation oder beide – ausgewählt. Im zweiten können die Farben für die einzelnen Elementtypen bestimmt werden, während der letzte Bereich die Parameter für den kräftebasierten Algorithmus enthält, der die Positionen der einzelnen Elemente ermittelt.

Das Ergebnis des Visualisierungsprozesses ist eine dreidimensionale Visualisierung der Strukturinformationen aus dem Ecore-Modell in X3D, die anschließend durch einen X3D-Browser dargestellt werden kann. Abbildung 27 zeigt die Visualisierung des familytree-Beispiels.

familytree

childrentree

Individual

Familytree

Abbildung 27: X3D-Modell: Beispiel familytree

Wie der Abbildung zu entnehmen ist, kommt als Visualisierungstechnik eine abstrakte visuelle Metapher aus geschachtelten geometrischen Primitiven zum Einsatz. In der Visualisierung des *familytree*-Beispiels werden dabei die Pakete durch gelbe Würfel, die Klassen durch grüne Kugeln, die Attribute durch rote Kegel, die Methoden durch grüne Zylinder und die Beziehungen durch blaue Linien dargestellt. Für das Beispiel wurden per

Konfiguration nur Vererbungsbeziehungen visualisiert. Jedes Element wird zusätzlich mit seinem Bezeichner versehen, der oberhalb der geometrischen Form angezeigt wird. Die Schachtelung der Elemente erfolgt entsprechend der Pakethierarchie, so dass beispielsweise der Würfel für das Paket *childrentree* innerhalb des Würfels für das Paket *familytree* liegt. Elemente, die andere Elemente enthalten können, werden dazu transparent dargestellt. Ebenfalls abhängig von der Pakethierarchie sind die Größen der einzelnen Elemente, so dass große Elemente mehr Elemente enthalten als kleinere. Im nächsten Kapitel wird die so eben beschrieben Softwarevisualisierung entsprechend der Taxonomie eingeordnet.

5.2 Einordnung

Zur Einordnung der Softwarevisualisierung des Basisprototyps werden die sechs Klassifizierungsmerkmale Aufgabe, Zielgruppe, Ausrichtung, Darstellung, Medium und Automatisierungsgrad der Taxonomie aus Kapitel 2.6 verwendet.

Die Aufgabe, die mit der generierten Softwarevisualisierung des Basisprototyps erfüllt werden soll, ist die Unterstützung der Erkenntniserlangung über den Aufbau und die Struktur von Software. Als alternative Sicht auf ein Softwaresystem kann die Visualisierung sowohl für die Entwicklung neuer Systeme als auch für die Wartung und Erweiterung bestehender Systeme verwendet werden. Einen ersten Teil der Zielgruppe stellen somit Personen aus Softwareprojekten dar. Da der Basisprototyp aber auch als Forschungsgegenstand entworfen wurde, sind auch Personen aus Lehre und Forschung Bestandteil der Zielgruppe.

Gegenstand der Softwarevisualisierung des Basisprototyps ist die Struktur von Software, die aus Paketen, Klassen, Methoden, Attributen und Beziehungen zwischen den Klassen besteht. Die Visualisierung dieser Strukturinformationen erfolgt durch eine abstrakte visuelle Metapher aus geschachtelten geometrischen Primitiven, deren Positions- und Größenangaben über einen kraftgerichteten Layoutalgorithmus ermittelt werden. Über Konfigurationsmöglichkeiten auf den Einstellungsseiten von Eclipse kann der Benutzer die Darstellung seinen individuellen Bedürfnissen und Wünschen anpassen. Die Möglichkeiten zur Interaktion mit dem X3D-Modell werden durch den X3D-Browser vorgegeben. In den meisten Browsern ist nur eine Änderung der Ansicht möglich, indem beispielsweise die Kamera gedreht oder bewegt wird. Ein interaktives Erforschen der Visualisierung ist somit nur stark eingeschränkt möglich.

Durch das standardisierte X3D- beziehungsweise VRML-Format können die generierten Softwarevisualisierungen auf unterschiedlichen Medien, wie einem PC-Monitor oder dem

Virtual Reality-Labor des Instituts für Wirtschaftsinformatik der Universität Leipzig, wiedergegeben werden. Das Generieren der Visualisierungen erfolgt voll automatisch ohne manuelle Eingriffe durch den Benutzer, womit ein sehr gutes Verhältnis von Aufwand und Nutzen erreicht wird. Das nächste Kapitel beschreibt den Ablauf und den Aufbau des dafür verwendeten Generators.

5.3 Visualisierungsprozess des Generators

Im Kern besteht der Generator aus drei M2M-Transformationen, die mit dem oAW-Framework realisiert werden. Abbildung 28 zeigt den Visualisierungsprozess des Generators im Kontext der Visualisierungspipeline (Kapitel 2.4). Der oAW-Workflow, der den Visualisierungsprozess im Basisprototypen realisiert, wird ausführlich in (R. Müller 2009) beschrieben und daher an dieser Stelle nicht noch einmal aufgeführt.

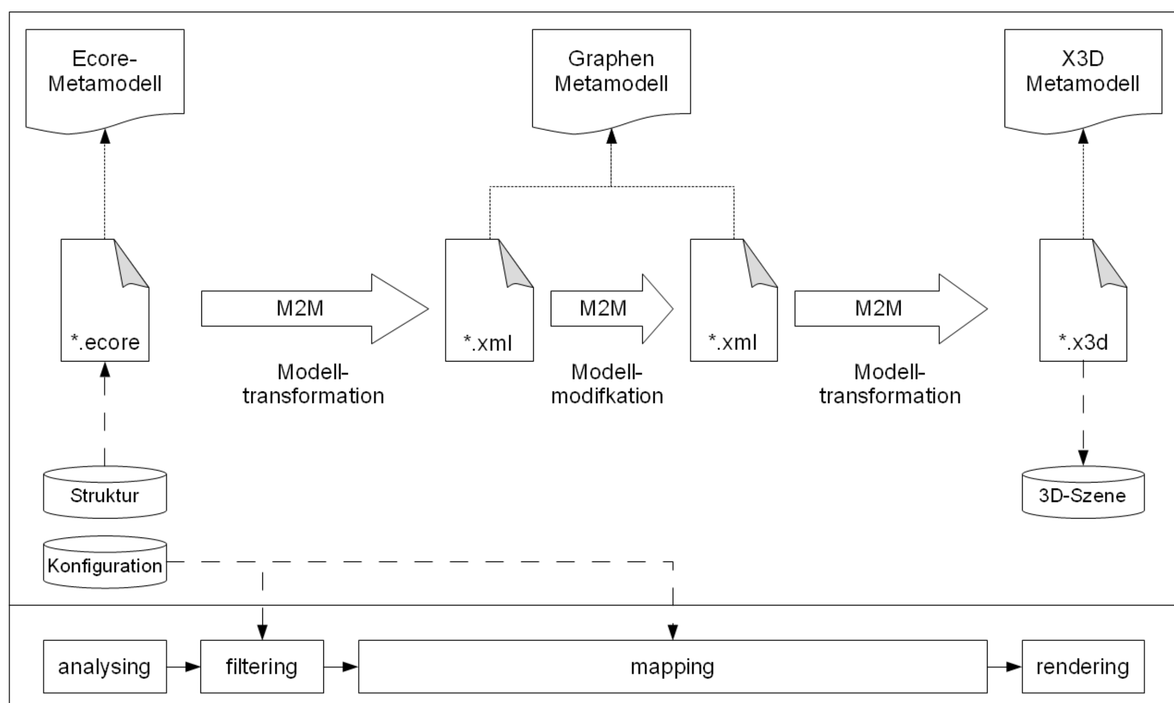

Abbildung 28: Visualisierungsprozess Basisprototyp
(nach R. Müller 2009)

Der *analysing*-Schritt der Visualisierungspipeline wird nicht direkt im Generator umgesetzt. Somit bilden die Strukturinformationen der Software aus dem *Ecore*-Modell und die Konfigurationsinformationen der Eclipse Einstellungsseiten den Ausgangspunkt des Prozesses.

Die erste M2M-Transformation bildet die Bestandteile des *Ecore*-Modells auf ein XML-basiertes Graphenmodell ab. Da nicht alle Bestandteile des *Ecore*-Modells abgebildet werden, wird in dieser Transformation das *filtering* vollzogen. Des Weiteren wird auch ein erster Teilschritt des *mapping* realisiert, da zum Beispiel die in Ecore enthaltene Pakethierarchie ebenfalls in das Graphenmodell übertragen wird.

Der zweite Teil des *mapping* wird in der zweiten M2M-Transformation vollzogen. In dieser Modellmodifikation werden die geometrischen Positionen und Größen der Elemente mit einem kräftebasierten Layout-Algorithmus ermittelt und im Graphenmodell ergänzt.
Die letzte Transformation als dritter Teil des *mapping* überführt das um Positions- und Größenangaben angereicherte Graphenmodell in das X3D-Modell. Dieses kann anschließend in einem X3D-Browser dargestellt werden, womit der Schritt *rendering* vollzogen wird. Im Folgenden sollen die drei Transformationen inklusive Ausschnitten aus ihrer Implementierung beschrieben werden. Für eine ausführliche Beschreibung der Implementierung wird dagegen auf (R. Müller 2009) verwiesen.

5.3.1 Modelltransformation von Ecore zu Graph

Wie bereits erwähnt, werden in dieser ersten Transformation die Bestandteile des Ecore-Modells auf ein XML-basiertes Graphenmodell abgebildet. Das Graphenmodell besteht im Wesentlichen aus den drei Elementtypen *Node* (Knoten), *Edge* (Kanten) und *Cluster* (Gruppen). Abbildung 29 zeigt eine Übersicht des grundlegenden Aufbaus.

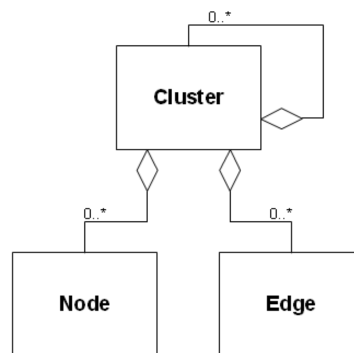

Abbildung 29: Graphenmodell: Aufbau

Wie der Abbildung zu entnehmen ist, sind Elemente vom Typ *Node* und *Edge* immer einem *Cluster*-Element untergeordnet. Zudem kann ein *Cluster* auch anderen *Cluster*-Elemente enthalten. Jedem Element des Graphenmodells können zudem Eigenschaften über *Property*-Elemente hinzugefügt werden.
Bei der Transformation werden die Pakete und Klassen auf *Cluster*-Elemente, Attribute sowie Methoden auf *Node*-Elemente und die Beziehungen zwischen den Klassen auf *Edge*-Elemente abgebildet. Die Elemente des Graphenmodells werden dabei entsprechend der Pakethierarchie aus dem Ecore-Modell geschachtelt. Für die Endpunkte einer Kante wird den *Cluster*-Elementen der entsprechenden Klassen jeweils ein zusätzliches *Node*-Element hinzugefügt. Listing 12 zeigt einen Auszug der Transformationsvorschrift in *Xtend* für das Abbilden von Klassen auf *Cluster*-Elemente.

```
1  create Cluster toCluster(EClass cls):
2      setProperty(cls, "Cluster", this)->
3      setViewType(toClusterViewType("class", cls.name))->
4      setLayoutEngineType(toForceDirectedLayout())->
5      property.add(toTypeProperty("class"))->
6      node.addAll(cls.eContents.typeSelect(EAttribute).toNode())->
7      node.addAll(cls.eContents.typeSelect(EOperation).toNode());
```

Listing 12: Ecore-zu-Graphenmodell: Auszug Xtend-Transformation

In Zeile 2 wird mit *setProperty* eine globale Eigenschaft (engl. *property*) definiert, die sich aus der Klasse und dem Cluster-Element als Schlüssel-Wert-Paar zusammen setzt. Diese Eigenschaft wird für das Abbilden der Beziehungen auf *Edge*-Elemente benötigt. In den Zeilen 3 und 4 folgen die Unterelemente *ViewType* und *LayoutEngineType*, die Informationen für die Ausführung des kräftebasierten Algorithmus enthalten. Im *ViewType*-Element ist zudem den Bezeichner der Klasse hinterlegt. In Zeile 5 wird der Typ des *Cluster*-Elements (in dem Fall *class*) als Eigenschaft hinzugefügt. Anschließend werden die Attribute (*EAttribute*) und Methoden (*EOperation*) der Klasse dem Cluster als *Node*-Elemente mit *node.addAll* hinzugefügt. Dabei wird die überladene Erweiterung *toNode* verwendet. In Listing 13 ist das Cluster-Element einer Klasse, als Auszug des Ergebnisses der ersten Transformation, dargestellt.

```
1  <Cluster>
2          <ViewType Name="Spherical Cluster">
3                  <Property Key="Label" Value="Family"/>
4          </ViewType>
5          <Property Key="Type" Value="class"/>
6          <LayoutEngineType Name="Force Directed">
7                  <Property Key="Spring" Value="5.0"/>
8                  <Property Key="Origin" Value="10.0"/>
9                  <Property Key="Repulsion" Value="1.2"/>
10                 <Property Key="VelocityAttenuation" Value="0.02"/>
11         </LayoutEngineType>
12         <!-- Unterelemente -->
13 </Cluster>
```

Listing 13: Graphenmodell: Beispiel Cluster

5.3.2 Modellmodifikation des Graphen

Bei der Modifikation des Graphenmodells werden die Positions- und Größenangaben ermittelt und als Elemente vom Typ *Property* in das Graphenmodell eingefügt. Für diese Transformation wird die externe Anwendung WilmaScope[9] genutzt, die mittels einer von

9 http://wilma.sourceforge.net/

Müller implementierten Ablaufkomponente aufgerufen wird. Wilmascope ermittelt die Positionen und Größen anhand der Schachtelung der Elemente, den in der Konfiguration definierten Formen der Elemente, wie zum Beispiel Würfel oder Kugeln für Pakete, sowie weiteren Parametern für den kräftebasierten Algorithmus, die im Graphenmodell hinterlegt sind. Nähere Angaben zu den Konfigurationsmöglichkeiten, dem Ablauf des Algorithmus sowie der Implementierung der Ablaufkomponente sind (R. Müller 2009) zu entnehmen. In Listing 14 ist beispielhaft das Cluster-Element für ein Paket angegeben, welches um Angaben zur Größe (Zeile 4) und Position (Zeile 6) ergänzt wurde.

```
1  <Cluster>
2          <ViewType Name="Box Cluster">
3                  <Property Key="Label" Value="familytree"/>
4                  <Property Key="Radius" Value="3.287216"/>
5          </ViewType>
6          <Property Key="Position" Value="-0.009675009 -6.3787383E-4 -0.0024762421"/>
7          <Property Key="Type" Value="package"/>
8          <LayoutEngineType Name="Force Directed">
9                  <Property Key="Spring" Value="5.0"/>
10                 <Property Key="Origin" Value="10.0"/>
11                 <Property Key="Repulsion" Value="1.2"/>
12                 <Property Key="VelocityAttenuation" Value="0.02"/>
13         </LayoutEngineType>
14         <!-- Unterelemente -->
15 </Cluster>
```

Listing 14: Graphenmodel-Modifikation: Beispiel Cluster

5.3.3 Modelltransformation von Graph zu X3D

In der letzten Transformation wird das Graphenmodell in das Zielformat X3D überführt, welches anschließend von einem X3D-Browser dargestellt werden kann. Abhängig von der Konfiguration für Pakete, Klassen, Attribute und Methoden werden die *Cluster*- und *Node*-Elemente in Würfel, Kugeln, Kegel oder Zylinder überführt und zusätzlich der definierte Farbwert ergänzt. *Edge*-Elemente werden durch Linien abgebildet, deren Anfangs- und Endpunkte ein zusätzliches Element erhalten, dessen Form sich nach der Klasse richtet. Werden Klassen beispielsweise als Kugeln abgebildet, so ist die Form der Anfangs- und Endpunkte ebenfalls eine Kugel. Im X3D-Modell entfällt die Schachtelung der Elemente nach der Pakethierarchie, da diese bereits in den Positionsangaben berücksichtigt ist. Das folgende Listing zeigt einen Ausschnitt aus der Transformationsvorschrift in *Xtend*, mit dem die Cluster-Elemente des Graphenmodells in X3D-Elemente des Typs *TransformType* transformiert werden.

```
1  create TransformType toTransform(Cluster cluster):
2      let mat = new MaterialType:
3      let app = new AppearanceType:
4      let shapes = new ShapeType:
5      setScale(cluster.viewType.property.select(prop|prop.key=="Radius").value.first().toString().toScale())->
6      setTranslation(cluster.property.select(prop|prop.key=="Position").value.first().toString())->
7      switch (getProperty(cluster.property.select(prop|prop.key=="Type").value.first().toString() + ".shape")){
8          case 'Sphere': shapes.setSphere(new SphereType)
9          case 'Box': shapes.setBox(new BoxType)
10         default : shapes
11     }->
12     mat.setDiffuseColor(cluster.property.select(prop|prop.key=="Colour").value.first().toString())->
13     app.material.add(mat)->
14     shapes.setAppearance((AppearanceType)app.clone())->
15     shapes.appearance.material.setTransparency(0.8)->
16     shape.add(shapes);
```

Listing 15: Graphenmodell-zu-X3D: Auszug Xtend-Transformation

In den Zeilen 2 bis 4 werden Instanzen der verschiedenen Unterelemente angelegt, damit diese direkt konfiguriert werden können. Die Größe des Elements (Attribut *scale*) wird in Zeile 5 aus dem Graphenmodell gelesen und mit der Methode *toScale* umgewandelt. Zeile 6 definiert analog dazu die Position des Elements (Attribut *translation*). Die Mehrfachverzweigung (Zeile 7 bis 11) bestimmt anschließend die Form des Elements, ausgehend von der Typ-Eigenschaft und der in der Konfiguration definierten Form für diesen Typ. Es folgt das Setzen der Farbe (Zeile 12) und der Transparenz (Zeile 15). In den Zeilen 13, 14 und 16 werden die Unterelemente entsprechend der Spezifikation von X3D ineinander geschachtelt. Die Übertragung des Elementbezeichners wurde aus Platzgründen und wegen der geringen Relevanz für das Gesamtverständnis aus Listing 15 entfernt. Das Listing 16 zeigt das Ergebnis eines transformierten Cluster-Elements, wobei ebenfalls auf den Bezeichner verzichtet wird.

```
1  <Transform scale="3.2896276 3.2896276 3.2896276"
           translation="-0.00290084 -0.003817271 -0.0021060207">
2      <Shape>
3          <Appearance>
4              <Material diffuseColor="1 1 0" transparency="0.8"/>
5          </Appearance>
6          <Box/>
7      </Shape>
8      <!-- Schriftzug für den Bezeichner -->
9  </Transform>
```

Listing 16: Graphenmodell-zu-X3D: Auszug Ergebnis

5.4 Ansatzpunkte der Erweiterung

Bevor die von Müller gelegte Grundlage um eine Benutzungsschnittstelle erweitert werden kann, müssen zunächst die Stellen beschrieben werden, an denen die Erweiterung ansetzen kann. Für die Erweiterung des Basisprototyps wurden drei Ansatzpunkte identifiziert:

- Das Zielformat X3D
- Der Visualisierungsprozess des Generators
- Die Integration als Eclipse-Plugin

Den ersten und gleichzeitig wichtigsten Ansatzpunkt stellt das Zielformat X3D dar. Für die Realisierung der Interaktion in X3D eignen sich das bereits vorgestellte Ereignismodell und die Möglichkeit, eigene Knotentypen (X3D-Prototypen) zu definieren, die spezifische Funktionalität zur Verfügung stellen können. Der Ansatz besteht also darin, X3D-Prototypen aufbauend auf dem Ereignismodell zu definieren, die Funktionalität für die Interaktion bereitstellen. Neben der Realisierung der einzelnen Interaktionstechniken muss eine einheitliche und flexible Architektur erarbeitet werden, welche die einzelnen Abhängigkeiten zwischen den Interaktionstechniken und dem Ereignismodell organisiert und bei Bedarf mit geringem Aufwand erweitert werden kann.

Weil diese Architektur inklusive der X3D-Prototypen in den Visualisierungsprozess integriert werden muss, stellt der Visualisierungsprozess des Generators den zweiten Ansatzpunkt dar. Dessen Ergebnis, im Folgenden als *statisches* X3D-Modell bezeichnet, muss in einer weiteren M2M-Transformation in ein interaktives X3D-Modell überführt werden. Das Anfügen dieser Transformation am Ende des Visualisierungsprozesses ermöglicht es, die anderen Teilschritte des Prozesses zu variieren. So kann zum Beispiel eine andere Metapher für die Visualisierung genutzt werden oder es können weitere Informationen wie beispielsweise Metriken in der Visualisierung verarbeitet werden, ohne dass dies direkten Einfluss auf die Interaktion besitzt. Die einzige Bedingung dafür ist, dass das statische X3D-Modell alle relevanten Daten für die Transformation in ein interaktives X3D-Modell enthält. Da dies im Basisprototyp noch nicht der Fall ist, müssen geringfügige Änderungen an den anderen Transformationen vorgenommen werden.

Die Integration in das Plugin stellt den letzten Ansatzpunkt für die Erweiterung des Prototypen dar. Da der Großteil der Änderungen aber in X3D beziehungsweise am Generator stattfindet, beschränkt sich dieser Ansatzpunkt auf die Gestaltung der Einstellungsseiten und der Verarbeitung der Werte in der Transformation. Das nächste Kapitel beschreibt die realisierte Benutzungsschnittstelle, die durch das Umsetzen der drei in diesem Kapitel identifizierten Ansatzpunkte implementiert wurde.

6 Erweiterung des Prototyps für die Interaktion

Für die Interaktion mit einer Softwarevisualisierung der Struktur wurde der Basisprototyp um eine graphische Benutzungsschnittstelle erweitert, die auf den drei identifizierten Ansatzpunkten des Basisprototyps aufbaut. Im ersten Unterkapitel wird zunächst die Benutzungsschnittstelle des interaktiven X3D-Modells mit allen zur Verfügung gestellten Funktionen beschrieben. Dem schließen sich drei Unterkapitel an, welche die Erweiterungen des Basisprototyps anhand der drei Ansatzpunkte X3D, Visualisierungsprozess des Generators und Integration in Eclipse beschreiben.

6.1 Benutzungsschnittstelle

Die Benutzungsschnittstelle des Prototyps setzt an der Benutzungsschnittstelle eines X3D-Browsers an, deren Aufbau und Funktionsweise zunächst kurz beschrieben werden sollen. Die Anwendungsschnittstelle der meisten X3D-Browser stellt lediglich Funktionen zur Änderung der Perspektive auf die dreidimensionale Szene zur Verfügung, wie beispielsweise das Bewegen oder Drehen der Kamera. Die Dialogschnittstelle besteht in der Regel aus dem Auswählen einer Funktion und dem anschließenden Ausführen dieser durch direkte Manipulation der Szene, was der funktionsorientierten Interaktion entspricht. Die Steuerelemente der Ein-/Ausgabeschnittstelle sind Schaltflächen und Menüs. Als Eingabegeräte kommen Maus und Tastatur zum Einsatz, während die Ausgabe über einen Farbbildschirm erfolgt. Die Benutzungsschnittstellen aller für diese Arbeit untersuchten X3D-Browser ermöglichen überwiegend Interaktionstechniken der Kategorie *explore*. Einige realisieren zusätzlich Funktionen für *encode*-Interaktionstechniken. Der Einfluss dieser auf den Prozess der Erkenntniserlangung von Software ist allerdings gering, da es sich nur um sehr allgemeine Funktionen handelt, wie zum Beispiel das Darstellen der Kanten ohne die Flächen.

Die Erweiterung der Benutzungsschnittstelle des X3D-Browsers erfolgt, indem zusätzliche X3D-Elemente als Steuerelemente in die Szene eingefügt werden. Über das vom Browser realisierte Ereignismodell von X3D nehmen diese die Eingaben des Benutzers, wie zum Beispiel das Klicken der Maus, entgegen und verarbeiten sie entsprechend ihrer Aufgabe. Auf diese Weise wird eine graphische Benutzungsoberfläche geschaffen, die unabhängig von einem bestimmten X3D-Browser die Interaktion mit einer Softwarevisualisierung der Struktur ermöglicht. In Abbildung 30 ist die graphische Benutzungsschnittstelle, kurz GUI, mit dem *familtytree*-Beispiel dargestellt. In den folgenden Ausführungen werden alle Elemente der Repräsentation, die keine Steuerelemente sind, als Darstellungselemente bezeichnet.

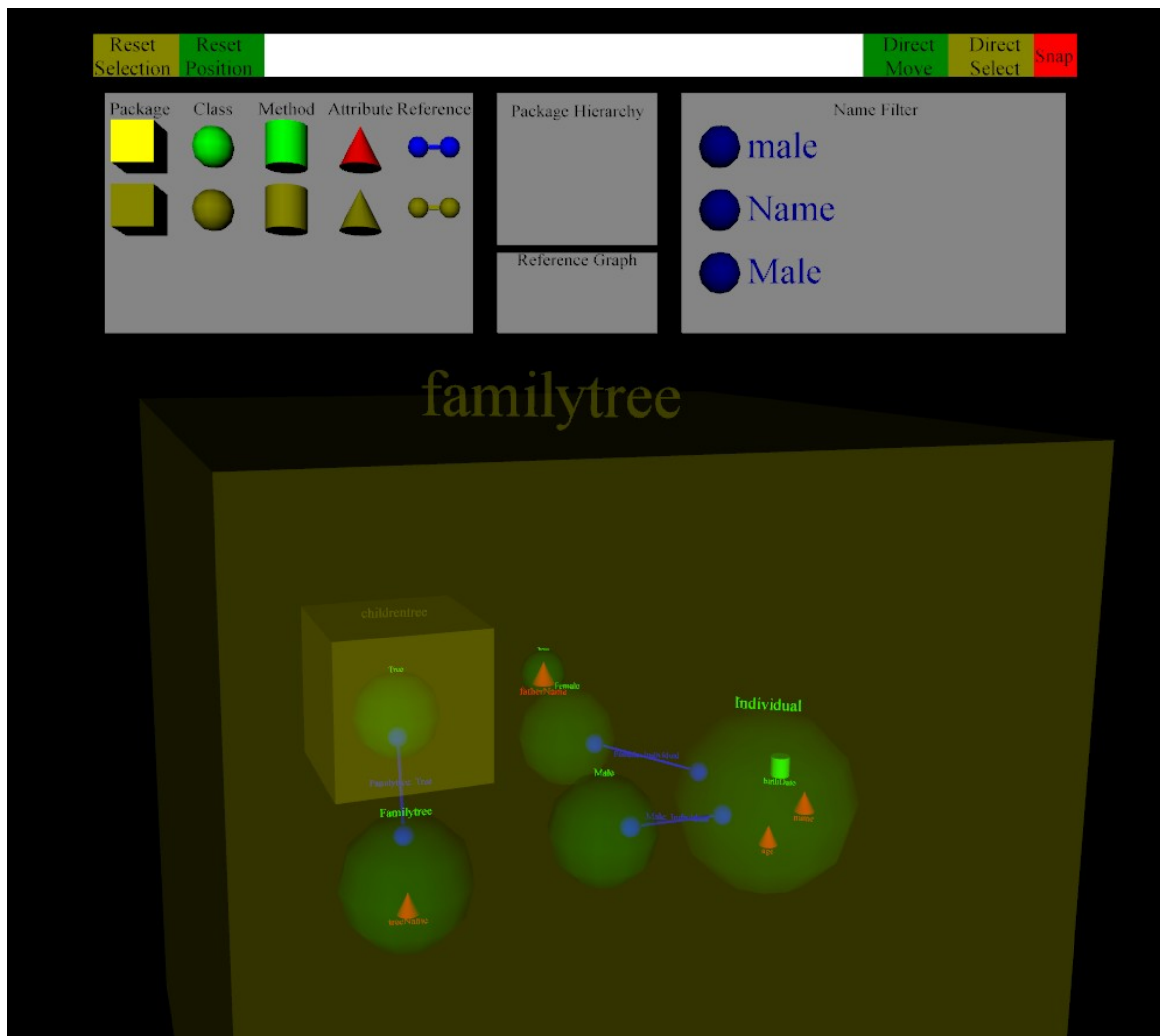

Abbildung 30: GUI: Überblick

Die GUI unterteilt sich in einen Kopf und einen Körper. Der Kopf enthält die allgemeinen Steuerelemente zum Verschieben der GUI (weißer Balken), zum Ein- und Ausblenden des GUI-Körpers (*snap*) sowie zum Ein- und Ausschalten der direkten Manipulation. Im Körper sind dagegen Steuerelemente in verschiedenen Gruppen nach ihren Funktionen sortiert. Durch die Gruppierung wird das Gestaltgesetz der räumlichen Nähe ausgenutzt. Der Körper realisiert die Manipulation nach Elementtyp (linke Gruppe), das Navigieren in der Pakethierarchie und dem Klassengraph (mittlere Gruppen) und das Identifizieren der Elemente anhand ihres Bezeichners (rechte Gruppe). Eine ausführliche Beschreibung der Funktionalität erfolgt in den Kapiteln 6.1.1 bis 6.1.4.

Die Ein/Ausgabenschnittstelle der GUI verfügt über lediglich zwei Arten von Steuerelementen, nämlich *check*-Buttons und *push*-Buttons. Mittels eines Check-Buttons können zwei Zustände für ein- und ausgeschaltet unterschieden werden. Beim Wechsel der Zustände können somit auch zwei unterschiedliche Funktionen ausgeführt werden. Im eingeschalteten Zustand ist er farblich hervorgehoben, wie beispielsweise der *snap*-Button auf rechten Seite des GUI-Kopfes. Ein *push*-Button besitzt dagegen keine Zustände. Bei

seiner Aktivierung kann somit auch nur eine Funktion ausgeführt werden, wie zum Beispiel das Löschen aller Selektionen mit dem *reset-selection*-Button auf der linken Seite des GUI-Kopfes. Alle Ausgaben der GUI wurden unter Berücksichtigung der sieben Eigenschaften Klarheit, Unterscheidbarkeit, Kompaktheit, Konsistenz, Erkennbarkeit, Lesbarkeit, Verständlichkeit der DIN EN ISO 9241-12 gestaltet.

Die Dialogschnittstelle ist ähnlich der eines X3D-Browsers gestaltet. Die Aktivierung eines Steuerelements realisiert entweder direkt die Manipulation der Repräsentation oder wählt eine Funktion aus, die der Benutzer anschließend auf die Darstellungselemente der Repräsentation anwendet. Während es sich bei letzterem immer um eine funktionsorientierte Interaktion handelt, kann im ersten Fall auch eine objektorientierte Interaktion statt finden, da zwischen der Interaktion mit einer festen (funktionsorientiert) und einer selbst bestimmten variablen Menge (objektorientiert) unterschieden wird. Im zuletzt genannten Fall wählt der Benutzer die Darstellungselemente durch Selektion aus.

Die folgenden Kapitel beschreiben die Funktionen der Anwendungsschnittstelle anhand der einzelnen Gruppen der GUI, wobei auch der GUI-Kopf als Gruppe aufgefasst wird. Dabei erfolgt ebenfalls eine Betrachtung der verschiedenen Interaktionstechniken, die mit den Funktionen realisiert werden können.

6.1.1 Direkte Manipulation

Im GUI-Kopf befinden sich auf der rechten Seite die Check-Buttons für die Aktivierung der direkten Selektion (*direct select*) und direkten Positionierung (*direct move*). Ist der entsprechende Check-Button aktiviert, kann der Benutzer die Darstellungselemente direkt mit der Maus auswählen. Für die Selektion genügt ein Klick mit der Maus auf das Darstellungselement. Ist dieses bereits selektiert, wird die Selektion aufgehoben. Die Visualisierung eines selektierten Darstellungselements erfolgt durch eine intransparente hellgelbe Darstellung. Bei der Positionierung wird ein Darstellungselement dagegen zuerst mit einem Klick gegriffen, um es anschließend bei gedrückter Maustaste zu positionieren. In Abbildung 31 sind beispielhaft einige Elemente selektiert, während in Abbildung 32 die Klassen (Kugeln) der Größe nach angeordnet sind. Der entsprechende Check-Button ist dabei farblich hervorgehoben und der GUI-Körper ist mit dem *snap*-Steuerelement ausgeblendet, womit die Elemente der Darstellung besser ausgewählt werden können.

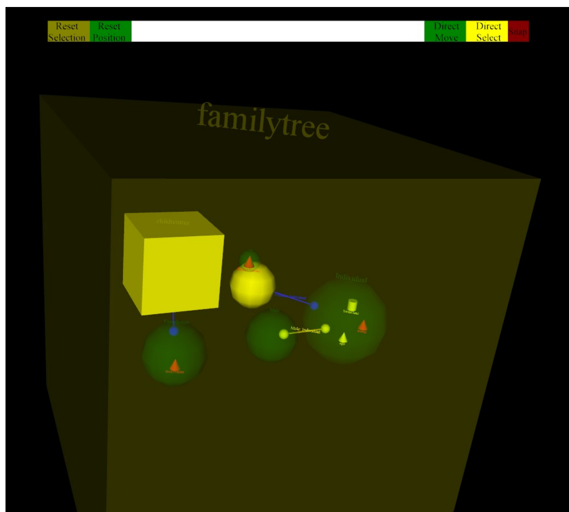

Abbildung 31: GUI: Selektion direkt

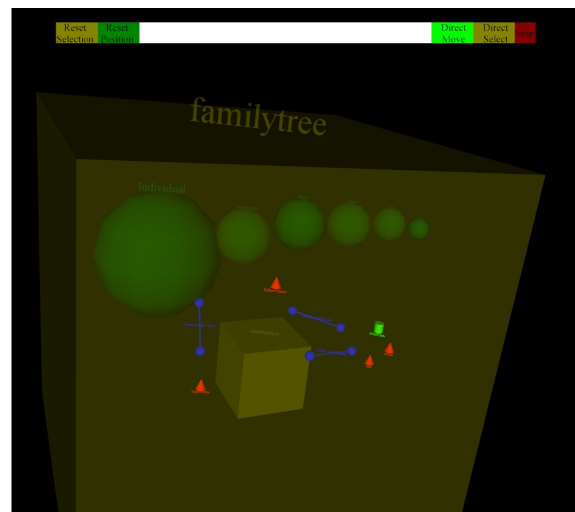

Abbildung 32: GUI: Positionierung direkt

Die beiden Push-Buttons auf der linken Seite des GUI-Kopfes löschen alle Selektionen (*reset selection*) beziehungsweise Positionsänderungen (*reset position*). Das direkte Selektieren der Elemente ist ein typischer Vertreter der *select*-Interaktionstechniken. Somit können Orientierungshilfen erstellt werden oder Elemente für die Anwendung anderer Interaktionstechniken ausgewählt werden. Dagegen ist das direkte Positionieren der einzelnen Elemente eine klassische Variante einer *reconfigure*-Interaktionstechnik. Sie kann beispielsweise für das Vergleichen von Darstellungselementen genutzt werden.

6.1.2 Manipulation nach Elementtyp

Die erste Gruppe im linken Bereich des GUI-Körpers enthält drei Reihen von Steuerelementen, welche die Darstellungselemente nach ihrem Typ – Paket, Klasse, Attribut, Methode, Referenz – manipulieren. Der Benutzer kann damit Darstellungselemente eines bestimmten Typs ein-/ausblenden, selektieren/deselektieren und alle Selektionen löschen. Die Formen der Steuerelemente entsprechen dabei denen der Darstellungselemente, womit das Gestaltgesetz der Ähnlichkeit/Gleichheit erfüllt wird. Für die Unterscheidung der drei Reihen wird eine weitere Variante dieses Gestaltgesetzes angewendet, indem die Steuerelemente entsprechend ihrer Funktion gefärbt sind. Auf den beiden folgenden Abbildungen sind die beiden Funktionen dargestellt, die mit der ersten und der zweiten Zeile der Steuerelemente realisiert werden können.

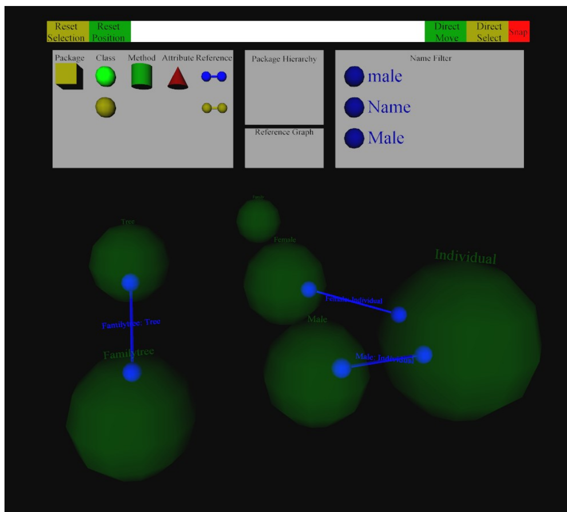

Abbildung 33: GUI: Filter Typ

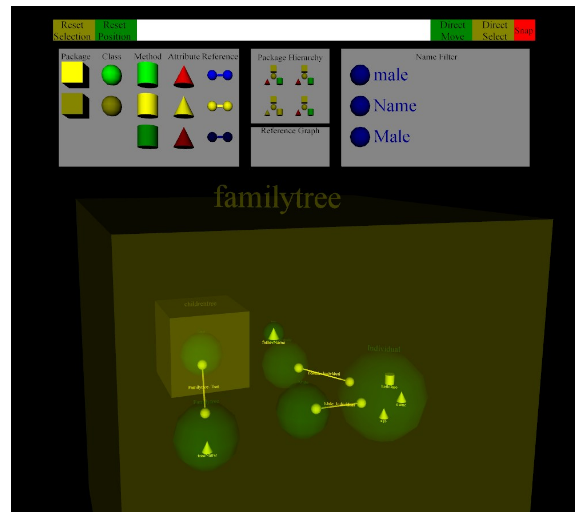

Abbildung 34: GUI: Selektion Typ

Mit der erste Reihe kann der Benutzer die Typen von Darstellungselementen bestimmen, die in der Repräsentation dargestellt werden sollen. Mit dem jeweiligen *check*-Button blendet er dazu alle Darstellungselemente eines Typs ein beziehungsweise aus. In Abbildung 33 wurden auf diese Weise alle Pakete, Methoden und Attribute ausgeblendet, womit nur noch Klassen und ihre Beziehungen dargestellt sind. Der Benutzer kann somit eine *filter*-Interaktionstechnik anwenden, um zum Beispiel den Grad der dargestellten Details zu beeinflussen.

Die *check*-Buttons der zweiten Reihe selektieren alle Darstellungselemente eines bestimmten Typs. In Abbildung 34 wurden so alle Darstellungselemente vom Typ Methode, Attribut und Beziehung selektiert (hellgelbe intransparente Darstellung). Die Steuerelemente der zweiten Reihe sind abhängig von denen der ersten, so dass ein Steuerelement zum Selektieren nur eingeblendet ist, wenn auch die Elemente des entsprechenden Typs eingeblendet sind. Mit dieser Funktionalität wird sowohl eine *filter*- als auch eine *select*-Interaktionstechnik ermöglicht, da die Darstellungselemente zum einen optisch hervorgehoben werden und gleichzeitig für die Anwendung anderer Interaktionstechniken genutzt werden können.

Die dritte Reihe der Steuerelemente löscht die Selektion aller Darstellungselemente des entsprechenden Typs. Dabei spielt es keine Rolle, ob diese durch ein Steuerelement der zweiten Reihe erfolgt ist oder der Benutzer sie mittels direkter Selektion ausgewählt hat. Die Funktion dieser Reihe kann somit als Teil der jeweiligen *select*-Interaktionstechnik aufgefasst werden. Ein Steuerelement der dritten Reihe wird erst eingeblendet, wenn mindestens ein Element des entsprechenden Typs selektiert ist. Da sie somit nach einmaliger Aktivierung ausgeblendet werden, handelt es sich bei den Steuerelementen der dritten Reihe um *push*-Buttons.

6.1.3 Navigation durch Pakethierarchie und Klassengraph

Im mittleren Bereich des GUI-Körpers befinden sich die beiden Gruppen für die Navigation in der Pakethierarchie (*package hierarchy*) und dem Klassengraphen (*reference graph*). Als Klassengraph wird der Graph bezeichnet, welcher aus Klassen (Knoten) und deren Beziehungen (Kanten) besteht, wobei keine Unterscheidung der Beziehungsarten erfolgt. Die über Navigation ausgewählten Darstellungselemente sind gleichzeitig auch selektiert, so dass sie bei einem weiteren Navigationsschritt ebenfalls berücksichtigt werden. Auf diese Weise ist eine mehrstufige iterative Navigation möglich.

Pakethierarchie-Gruppe

Die Pakethierarchie-Gruppe stellt insgesamt vier *push*-Buttons zur Verfügung, die eingeblendet werden, sobald mindestens ein Darstellungselement selektiert ist. Die folgende Abbildung zeigt ein Beispiel, in dem ausgehend von bereits selektierten Darstellungselementen (linkes Bild) die übergeordneten Elemente in der Pakethierarchie ausgewählt werden (rechtes Bild).

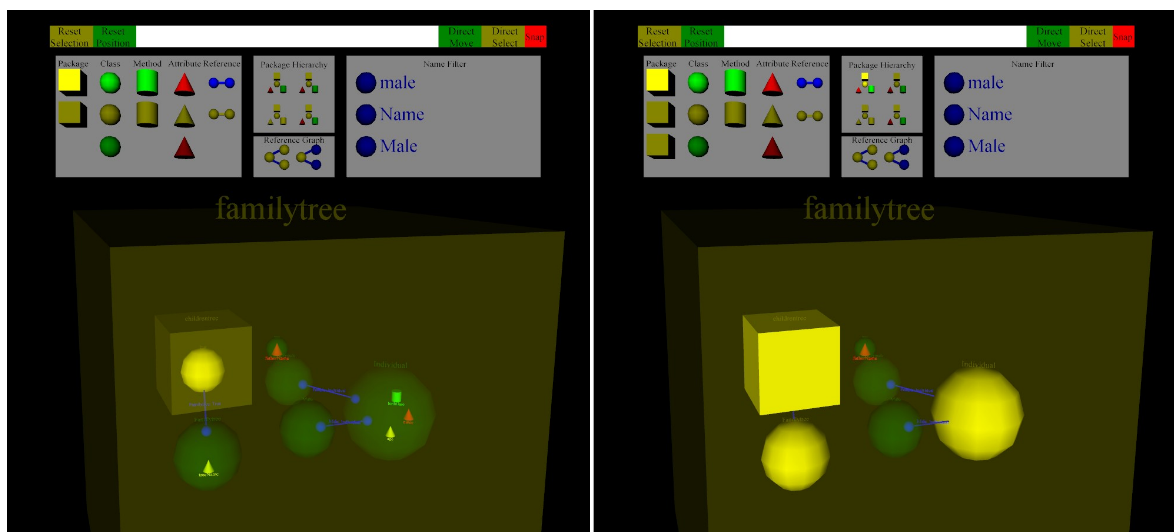

Abbildung 35: GUI: Navigation Pakethierarchie

Mit den beiden *push*-Buttons auf der linken Seite der Pakethierarchie-Gruppe kann der Benutzer die über- beziehungsweise untergeordneten Darstellungselemente auswählen. Im Beispiel der Abbildung 35 wurde demnach der farblich hervorgehobene *push*-Button oben links verwendet. Da es sich um eine iterative Navigation handelt, würde eine weitere Betätigung der gleichen Schaltfläche noch das *familytree*-Paket (großer gelber Würfel) zusätzlich auswählen. Die beiden *push*-Buttons auf der rechten Seite revidieren einen bereits ausgeführten Navigationsschritt. In Abbildung 35 würde mit der Schaltfläche rechts oben der Ausgangszustand der linken Seite wieder hergestellt werden.

Die mit der Pakethierarchie-Gruppe realisierte Funktionalität ermöglicht zum einen eine Interaktionstechnik der Kategorie *connect* und zum anderen eine *select*-

Interaktionstechnik. Zudem kann der Benutzer die Enthalten-In-Beziehung der Elemente unabhängig von einer bestimmten Darstellungsform, wie zum Beispiel der verwendeten Schachtelung, erfassen. Falls die Enthalten-In-Beziehung überhaupt keine visuelle Repräsentation besitzt, kann mit der Pakethierarchie-Gruppe auch eine *abstract/elaborate*-Interaktionstechnik realisiert werden, da sie dann zusätzliche Informationen nach Bedarf liefern kann.

Klassengraph-Gruppe

Die Klassengraph-Gruppe stellt nur zwei *push*-Buttons zur Verfügung, da die Navigation nicht entlang einer Hierarchie erfolgt, sondern lediglich die verbundenen Klassen ausgewählt werden können. Die Art der Beziehung zwischen den Klassen – Vererbung oder Assoziation – wird bei der Navigation nicht unterschieden. Abbildung 36 zeigt die Navigation im Klassengraph anhand einer Beispielklasse.

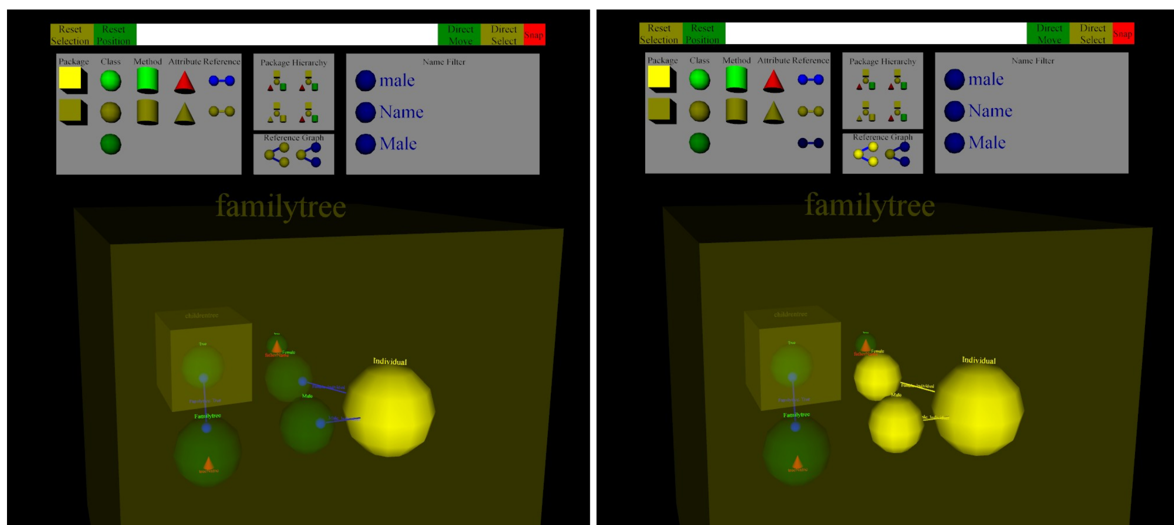

Abbildung 36: GUI: Navigation Klassengraph

Analog zur Pakethierarchie-Gruppe werden mit dem linken Push-Button die verbundenen Klassen ausgewählt, während mit dem rechten der letzte Navigationsschritt wieder revidiert werden kann. Die beiden Schaltflächen werden erst eingeblendet, wenn mindestens eine Klasse ausgewählt ist. Analog zur Pakethierarchie-Gruppe kann der Benutzer mit den Funktionen der Klassengraph-Gruppe eine *connect*- und eine *select*-Interaktionstechnik anwenden. Falls die Beziehungen keine visuelle Repräsentation besitzen, wird mit dieser Funktionalität auch eine *abstract/elaborate*-Interaktionstechnik realisiert, da der Benutzer zusätzliche Informationen erhält.

6.1.4 Identifikation nach Bezeichner und Tooltip

Mit der letzten Gruppe auf der rechten Seite des GUI-Körpers (*Name Filter*) kann der Benutzer Darstellungselemente nach dem Bezeichner auswählen. Jeder der drei *check*-Buttons der Gruppe selektiert/deselektiert dazu eine Menge von Darstellungselementen,

die durch das Anwenden eines regulären Ausdrucks auf die Bezeichner alle Elemente ermittelt wird. Die Beschriftungen der drei *check*-Buttons und die dazugehörigen regulären Ausdrücke werden auf einer Einstellungsseite des Plugins konfiguriert. Abbildung 37 zeigt ein Beispiel, in dem Elemente selektiert sind, welche die Buchstabenkette *Name* beziehungsweise *Male* enthalten. Des Weiteren wird in der Abbildung der Tooltip dargestellt, der angezeigt wird, sobald der Benutzer mit der Maus über das Darstellungselement fährt. Dieser wurde für die bessere Lesbarkeit nachträglich mit einem Bildbearbeitungsprogramm vergrößert.

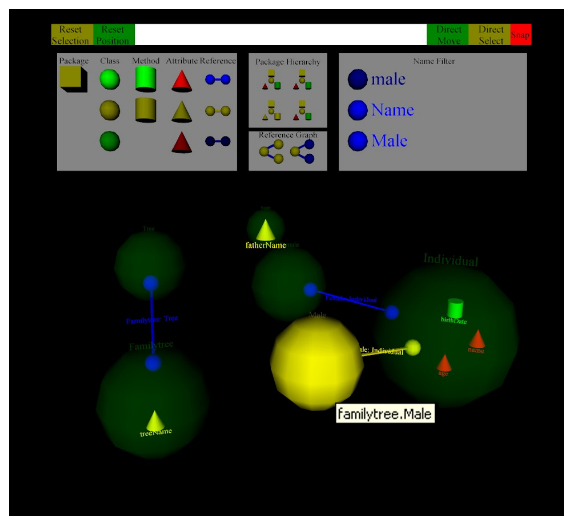

Abbildung 37: GUI: Selektion Name, Tooltip

Das Identifizieren einer Menge von Elementen nach dem Bezeichner stellt eine *filter*-Interaktionstechnik dar. Gleichzeitig kann auch wieder eine *select*-Interaktionstechnik angewendet werden, da die identifizierten Darstellungselemente zugleich auch selektiert sind. Die Funktionalität kann auch für eine *abstract/elaborate*-Interaktionstechnik genutzt werden, da beispielsweise Aspekte der Architektur auch über den Bezeichner ermittelt werden können. Ein entsprechender regulärer Ausdruck kann zum Beispiel alle *exception*-Klassen identifizieren, die bis auf den Bezeichner sonst keine Unterscheidungsmöglichkeiten zu anderen Klassen besitzen.

Mit dem Tooltip wird eine weitere Interaktionstechnik der Kategorie *abstract/elaborate* ermöglicht, da sich der Benutzer zusätzliche Details nach Bedarf anzeigen lassen kann. In der aktuellen Implementierung enthält er lediglich den voll qualifizierten Namen des Darstellungselements. In weiteren Arbeiten könnte der Tooltip aber beispielsweise um Metriken ergänzt werden.

6.2 Architektur des interaktiven X3D-Modells

Nachdem im letzten Kapitel die Funktionen der Benutzungsschnittstelle eingehend erläutert wurden, soll nun der Aufbau des interaktiven X3D-Modells vorgestellt werden,

der diese Funktionalität ermöglicht. Im Gegensatz zum statischen X3D-Modell, das nur aus einer Menge von geometrischen Primitiven besteht, wird für die Realisierung der Interaktion eine Architektur benötigt, mit der die komplexen Wechselwirkungen des Ereignismodells von X3D gehandhabt werden können. Eine solche Architektur ermöglicht es zudem, die Benutzungsschnittstelle mit einfachen Mitteln in verhältnismäßig kurzer Zeit um weitere Funktionen zu erweitern. Sie besteht, wie in Kapitel 5.4 erwähnt, aus X3D-Prototypen, die auf Basis des Ereignismodells von X3D die Interaktion abbilden. Für das bessere Verständnis werden im Folgenden nicht die einzelnen X3D-Prototypen vorgestellt, sondern die Architektur mit einem Schema beschrieben.

Schichtenmodell

Die Architektur unterteilt sich in vier aufeinander aufbauende Schichten (engl. *layer*), von denen jede eine fest definierte Teilaufgabe der Interaktion erfüllt. Diese Unterteilung erlaubt es, die komplexen Beziehungen und Wechselwirkungen zu handhaben und ermöglicht es zudem, weitere Funktionen einfacher zu integrieren. Die nächste Abbildung zeigt das Schichtenmodell bestehend aus den drei Schichten Manipulation (*ManipulationLayer*), Selektion (*SelectionLayer*) und Navigation (*NavigationLayer*).

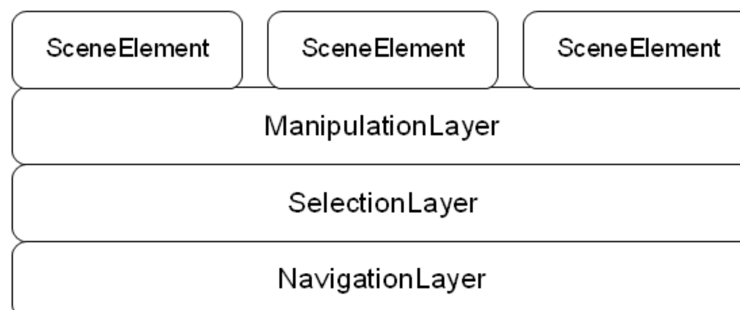

Abbildung 38: X3D-Architektur: Schichtenmodell

Die Darstellungselemente (*SceneElement*) des statischen X3D-Modells bilden die erste Schicht und somit die Grundlage der Schichtenarchitektur. Die Kommunikation zwischen den Schichten erfolgt in beide Richtungen und immer nur mit der jeweils angrenzenden Schicht, so dass keine übersprungen werden kann.

Die Aufgabe der Manipulationsschicht ist die Kapselung der Manipulation der Darstellungselemente. Soll die Darstellung eines Elements in Folge der Interaktion geändert werden, so erfolgt dies nicht direkt am Darstellungselement, sondern wird von der Manipulationsschicht organisiert. Auf der anderen Seite werden Ereignisse der Darstellungselemente, wie der Klick mit der Maus auf das Element, zuerst an die Manipulationsschicht weitergeleitet, um sie anschließend entsprechend zu verarbeiten.

Die Selektionsschicht koordiniert die Selektion der Darstellungselemente, da diese auf mehrere Arten, wie zum Beispiel durch die direkte Selektion oder der Selektion nach

Element-Typ, erfolgen kann. Die Änderung des Erscheinungsbildes wird an die Manipulationsschicht delegiert. Auf der anderen Seite übermittelt die Selektionsschicht den Zustand der Selektion – selektiert, nicht selektiert – an die Navigationsschicht.
Deren Aufgabe ist die Realisierung der Navigation in Hierarchien und Graphen. Wird ein Darstellungselement in einem Navigationsschritt ausgewählt, so leitet die Navigationsschicht dieses Ereignis an die Selektionsschicht weiter.

Ablauf

Nachdem nun der grundlegende Aufbau der Schichtenarchitektur vorgestellt wurde, soll im Folgenden eine Beschreibung des Ablaufs der Interaktion unter Einbezug der Steuerelemente erfolgen. Jedes Darstellungselement erhält pro Schicht einen X3D-Prototyp, der die Funktionen der Schicht zur Verfügung stellt und als *Handler* bezeichnet wird. So besitzt der *ManipulationHandler* zum Beispiel eine Funktion für das Ein- und Ausblenden des Darstellungselements. Der Aufruf dieser Funktionen erfolgt entweder durch einen anderen *Handler* oder durch ein Steuerelement (*ControlElement*). Abbildung 39 zeigt einen Überblick der Kommunikation zwischen Darstellungselement, *Handlern* und Steuerelementen. Ausführliche Übersichten zum Ablauf der Kommunikation auf der Ebene der Ereignisse befinden sich dagegen im Anhang.

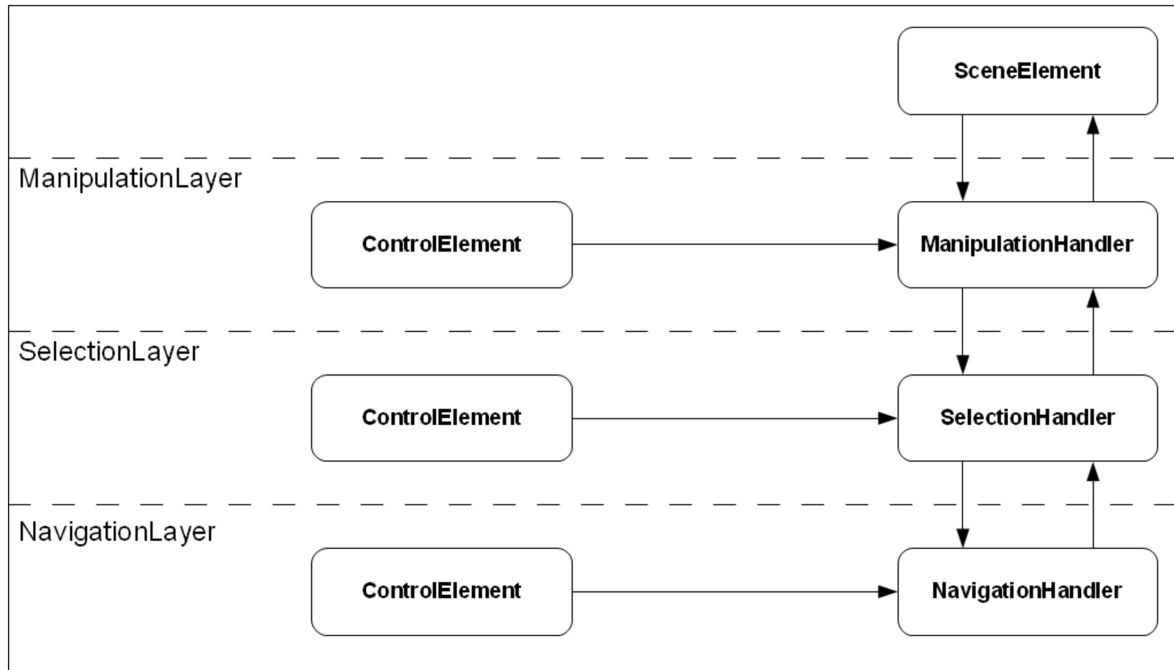

Abbildung 39: X3D-Architektur: Ablauf

Steuerelement

Weil ein Steuerelement für das Ausführen einer Funktion selektiert werden muss und dabei auch eine Manipulation der Darstellung erfolgt, ist eine Steuerelement analog zum Darstellungselement aufgebaut. Somit besitzt jedes Steuerelement auch einen *ManipulationHandler* und einen *SelectionHandler*, wie in Abbildung 40 dargestellt.

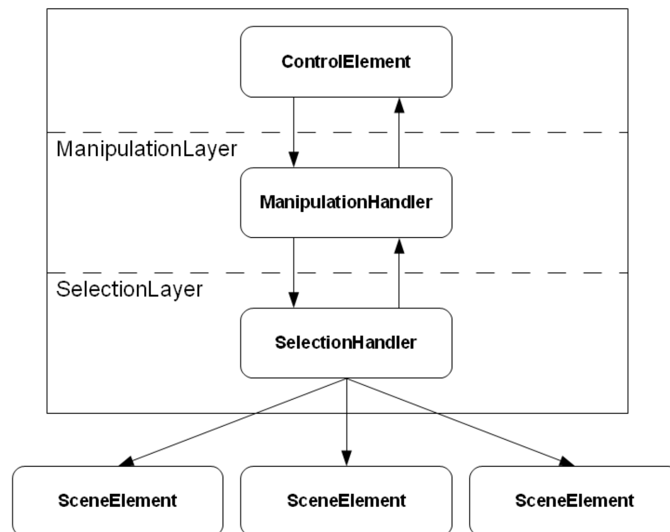

Abbildung 40: X3D-Architektur: Steuerelement

Wird ein Steuerelement mit einem Mausklick aktiviert, so wird dieses Ereignis zuerst an den *ManipulationHandler* weitergeleitet, der daraufhin die Funktion zum Selektieren im *SelectionHandler* aufruft. Je nach Aufgabe des Steuerelements wird anschließend die entsprechende Funktion auf den *Handlern* der Darstellungselemente aufgerufen, was im unteren Teil der Abbildung angedeutet ist.

Damit endet die Beschreibung der Architektur des interaktiven X3D-Modells. Im nächsten Kapitel werden die Erweiterungen des Generators zum Erstellen dieses Modells vorgestellt.

6.3 Erweiterung des Generators

Der Ansatz zur Erweiterung des Generators besteht in einer zusätzlichen Transformation, die das statische X3D-Modell in ein interaktives X3D-Modell überführt. Für die Verarbeitung der Darstellungselemente werden dabei der vollqualifizierte Name als eindeutiger Bezeichner und der Element-Typ benötigt. Da sich diese Informationen aber nicht aus dem statischen X3D-Modell extrahieren lassen, müssen geringfügige Änderungen an den anderen Transformationen des Generators vorgenommen werden. Das folgende Kapitel beschreibt daher, wie das statische X3D-Modell mit dem vollqualifizierten Namen und dem Typ des Elements ergänzt wird. Im darauf folgenden Kapitel 6.3.2 erfolgt dann die Betrachtung der Transformation in ein interaktives X3D-Modell.

6.3.1 Anpassung der Transformationen des Basisprototyps

Die erste Anpassung erfolgt bereits in der Ecore-zu-Graphenmodell-Transformation. Da der Typ eines Elements bereits als *Property*-Element in das Graphenmodell übertragen wird, muss lediglich noch der vollqualifizierte Name aus dem Ecore-Modell ergänzt

werden. Die dafür verwendete überladene *Xtend*-Erweiterung ist in Listing 17 aufgezeigt. Sie ermittelt den vollqualifizierten Namen durch rekursive Aufrufe entlang der Pakethierarchie.

```
1  String fullyQualifiedName(EPackage pkg) :
2      pkg.eSuperPackage == null ? pkg.name : fullyQualifiedName(pkg.eSuperPackage)+'.'+pkg.name;
3
4  String fullyQualifiedName(EClass cls) :
5      fullyQualifiedName(cls.ePackage)+'.'+cls.name;
6
7  String fullyQualifiedName(EAttribute att) :
8      fullyQualifiedName(att.eContainingClass)+'.'+att.name;
9
10 String fullyQualifiedName(EOperation op) :
11     fullyQualifiedName(op.eContainingClass)+'.'+op.name;
```

Listing 17: Generator-Anpassung: Vollqualifizierter Name

Analog zum Typ wird der vollqualifizierte Name als *Property*-Element in das Graphenmodell geschrieben. Eine Ausnahme bilden dabei die *Edge*-Elemente und die zusätzlich eingefügten *Node*-Elemente für die Start- und Endpunkte. Aus nicht geklärter Ursache werden deren *Property*-Elemente bei der Ermittlung von Position und Größe in der zweiten Transformation (Modellmodifikation des Graphen) gelöscht.

Dieses Problem wird durch die zweite Anpassung in der Graphenmodell-zu-X3D-Transformation aufgelöst. Dazu werden in einem vorgelagerten Schritt die noch fehlenden vollqualifizierten Namen ermittelt und dem Graphenmodell nachträglich noch hinzugefügt. Ein *Edge*-Element erhält eine Kombination aus den beiden vollqualifizierten Namen der beiden verbundenen Klassen, getrennt durch die Zahl 2, die für das englische Wort *to* steht (z.B. *familytree.Individual.2.familytree.Male*). Die Reihenfolge der Klassennamen besitzt dabei keine Semantik. Die zusätzlichen *Node*-Elemente für Anfangs- und Endpunkt erhalten den vollqualifizierten Namen des *Edge*-Elements. Für die eindeutige Unterscheidung wird noch zusätzlich die Zeichenkette *StartNode* beziehungsweise *Endnode* angehängt. Wie beim *Edge*-Element besitzt auch die Zuweisung einer der beiden Zeichenketten keine Semantik, sondern wird lediglich der Eindeutigkeit halber vorgenommen.

Die letzte Anpassung erfolgt durch die Übertragung der *Property*-Elemente des Graphenmodells in das statische X3D-Modell, wofür das X3D-Element *MetadataString* verwendet wird. Über die Attribute *name* und *reference* werden dabei die Schlüssel-Wert-Paare der *Property*-Elemente abgebildet. Das Attribut *value* des *MetadataString*-Elements kann dafür nicht verwendet werden, da dessen Typ (*MFString*) nicht mit oAW kompatibel

ist und somit nicht in einer *Xtend*-Erweiterung genutzt werden kann[10]. Die beiden *MetadataString*-Elemente für den vollqualifizierten Namen und den Typ werden nach der X3D-Spezifikation in ein *MetadataSet*-Element geschachtelt. Listing 18 zeigt das erweiterte X3D-Element für das *familytree*-Paket. Ein so erweitertes statisches X3D-Modell bildet den Ausgangspunkt für die Transformation in ein interaktives X3D-Modell, die im nächsten Kapitel beschrieben wird.

```
1  <Transform scale="3.2896276 3.2896276 3.2896276"
          translation="-0.00290084 -0.003817271 -0.0021060207">
2      <MetadataSet>
3          <MetadataString name="QualifiedName" reference="familytree"/>
4          <MetadataString name="Type" reference="package"/>
5      </MetadataSet>
6      <Shape>
7          <Appearance>
8              <Material diffuseColor="1 1 0" transparency="0.8"/>
9           </Appearance>
10         <Box/>
11     </Shape>
12     <!-- Schriftzug: familytree -->
13 </Transform>
```

Listing 18: Generator-Anpassung: Auszug statisches X3D-Modell

6.3.2 Modelltransformation in ein interaktives X3D-Modell

Listing 19 ist ein Auszug aus dem Workflow des Generators und zeigt die Transformation in ein interaktives X3D-Modell.

10 http://www.openarchitectureware.org/forum/viewtopic.php?forum=2&showtopic=10240&highlight=X3D

```
1  <!-- [...] Generiere statisches X3D-Modell [...] -->
2  <feature isSelected="interactionEnabled">
3      <!-- Load proto xml  -->
4      <component class="org.openarchitectureware.xsd.XMLReader">
5          <metaModel idRef="x3d" />
6          <modelSlot value="protoTypes"/>
7          <uri value="${model.proto}"/>
8      </component>
9
10     <!-- transform model to interactive x3dModel -->
11      <component class="oaw.xtend.XtendComponent">
12         <metaModel idRef="x3d" />
13         <invoke value=
               "${extensions.dir}interaction::x3d2interactive::x3d2interactive(x3dModel, protoTypes)" />
14         <outputSlot value="x3dModel" />
15     </component>
16 </feature>
17 <!-- [...] Schreibe interaktives X3D-Modell [...] -->
```

Listing 19: Modellmodifikation zur Interaktion: Workflow-Aufruf

Nach dem Generieren des statischen X3D-Modells wird die Transformation in ein interaktives X3D-Modell optional ausgeführt. Dazu wird in Zeile 2 geprüft, ob die Eigenschaft *interactionEnabled* in den Einstellungen des Plugins auf *true* gesetzt ist. In den Zeilen 4 bis 8 werden die X3D-Prototypen für die Interaktion aus einer separaten X3D-Datei geladen. Die Ablaufkomponente aus den Zeilen 11 bis 15 ruft anschließend die *Xtend*-Transformationsvorschrift mit *x3d2interactive(x3dModel, protoTypes)* auf und übergibt ihr dabei das statische X3D-Modell (*x3dModel)* und die geladenen X3D-Prototypen (*protoTypes)*. Im folgenden Listing ist ein gekürzter Auszug der *Xtend*-Transformationsvorschrift enthalten, der den generellen Aufbau der Transformation beschreibt.

```
1   create SceneType toBrowserScene(SceneType x3dScene, X3dType protoType):
2       let manipulabeSceneElements = x3dScene.transform.toManipulableSceneElements():
3       let selectableSceneElements = manipulabeSceneElements.toSelectableSceneElements():
4       let userInterfaceControls = new TransformType:
5
6       let elementTypeEnableGroup =
            createElementTypeEnableGroup(selectableSceneElements, userInterfaceControls):
7       let elementTypeSelectionGroup =
            createElementTypeSelectionGroup(selectableSceneElements, userInterfaceControls):
8       let packageNavigationyGroup =
            createPackageNavigationGroup(selectableSceneElements, userInterfaceControls):
9
10      protoDeclare.addAll(protoType.scene.protoDeclare)->
11      transform.addAll(selectableSceneElements)->
12      group1.add(createUserInterface(userInterfaceControls))->
13
14      group1.add(elementTypeEnableGroup)->
15      group1.add(elementTypeIsSelectedGroup)->
16      group1.add(packageNavigationyGroup)->
17
18      background.add(x3dScene.background.first());
```

Listing 20: Modellmodifikation zur Interaktion: Xtend-Auszug

Wie sich dem Kopf der Erweiterung entnehmen lässt, wird ein X3D-Element vom Typ *Scene* erstell, wofür das statische X3D-Modell sowie die X3D-Prototypen als Parameter übergeben werden. In Zeile 2 werden alle Darstellungselemente (*SceneElements*) des statischen X3D-Modells in manipulierbare Darstellungselemente umgewandelt. Dafür werden dem Darstellungselement der *ManipulationHandler*, die Sensoren für Selektion und Positionierung sowie alle benötigten Routen und Ereignishelfer hinzugefügt. Analog dazu werden in Zeile 3 die manipulierbaren Darstellungselemente in selektierbare Darstellungselemente überführt, indem unter anderem der *SelectionHandler* ergänzt wird. Demnach verfügt ein selektierbares Darstellungselement über die Funktionen zur Selektion des Elements sowie zur Manipulation der Darstellung (vgl. Kapitel 6.2 Abbildung 39). Der *NavigationHandler* kann dagegen nicht in gleicher Weise ergänzt werden, da ein Darstellungselement für jede Navigationsfunktion (Pakethierarchie oder Klassengraph) unterschiedliche *NavigationHandler* besitzt.

In Zeile 4 wird ein Behälter vom Typ *Transform* für die Steuerelemente der GUI angelegt. Die einzelnen Funktionen zur Interaktion werden in den Zeilen 6, 7 und 8 erstellt, wobei die dafür benötigten X3D-Elemente in einem *Group*-Element gespeichert werden. Jede Zeile stellt nur ein Beispiel für die Interaktionsmöglichkeiten auf einer der drei Schichten

dar. Das Ein-/Ausblenden (Manipulationsschicht) und das Selektieren der Elemente nach ihrem Typ (Selektionsschicht) wird in den Zeilen 6 und 7 erstellt, während in Zeile 8 die Navigation in der Pakethierarchie (Navigationsschicht) generiert wird. Neben den Darstellungselementen wird auch der Behälter für die Steuerelemente übergeben, in den die benötigten Steuerelemente ablegt werden.

In Zeile 10 werden die Definitionen der X3D-Prototypen als erstes zum *Scene*-Element hinzugefügt. Dem folgen in Zeile 11 alle Darstellungselemente, die bereits um die entsprechenden Funktionen zur Manipulation, Selektion und Navigation erweitert wurden. Die GUI wird anschließend in Zeile 12 mit den Steuerelementen aus dem Behälter generiert und als *Group*-Element dem *Scene*-Element hinzugefügt[11]. Die *Group*-Elemente der einzelnen Funktionen zur Interaktion folgen anschließend in den Zeilen 14 bis 16. Mit der bloßen Übertragung des *Background*-Elements aus dem statischen X3D-Modell endet die Transformation in ein interaktives X3D-Modell.

6.4 Integration in das Eclipse-Plugin

Die letzte Erweiterung des Basisprototyps stellt die Integration in Eclipse dar. Abbildung 41 zeigt die Einstellungsseite des Generators für die Interaktion.

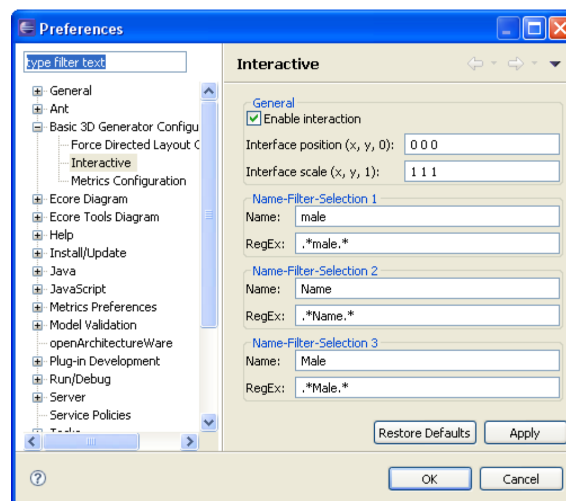

Abbildung 41: Eclipse-Einstellungsseite: Interaktion

Im oberen Bereich der Seite können die generellen Einstellungen (engl. *general*) vorgenommen werden. Über die Check-Box (*Enable interaction*) wird bestimmt, ob das statische X3D-Modell überhaupt in ein interaktives überführt werden soll (vgl. Kapitel 6.3.2, Listing 19 Workflow Auszug). Des Weiteren können die Position und die Skalierung der GUI verändert werden, wobei die endgültige Position auch von der Skalierung abhängig ist. Auch wenn für die Skallierung die x- und y-Werte genügen würden, muss für

11 Die Bezeichnung *group1* entsteht durch die Auflösung eines Bezeichnerkonflikts beim Parsen des X3D-Metamodells

eine erfolgreiche Verarbeitung ein z-Wert eingetragen werden. Dieser sollte für die Position immer 0 und für die Skallierung immer 1 sein.
Die drei Bereiche der unteren Seitenhälfte werden für die Funktion der Identifikation einer Menge von Darstellungselementen nach den Bezeichnern benötigt (vgl. Kapitel 6.1.4). In jedem Bereich wird dazu der Bezeichner für das Steuerelement und der reguläre Ausdruck für die zu identifizierende Menge von Darstellungselementen konfiguriert.
Mit der Umsetzung dieses dritten Ansatzpunkts schließt das Kapitel zur Erweiterung des Basisprototyps. Das nächste Kapitel zieht ein Fazit aus den Erkenntnissen dieser Arbeit und gibt einen Ausblick auf mögliche weitere Arbeiten.

7 Fazit und Ausblick

Mit dieser Arbeit wurde die grundlegende Bedeutung der Interaktion beim Erlangen von Erkenntnissen durch eine Softwarevisualisierung aufgezeigt. So dient sie nicht nur zur Lösung des Platzproblems bei großen Datenmengen, sondern auch zur tieferen Erforschung und Erkundung der Datengrundlage, welche in der Visualisierung dargestellt wird. Obwohl die Interaktion somit einen entscheidenden Beitrag zum Prozess des Verstehens liefert, konzentriert sich der Großteil der Forschung fast ausschließlich auf die Repräsentation. Diese Arbeit liefert somit auch einen Beitrag, diesen Missstand aufzuzeigen und ein Stück weit zu beheben, indem Erkenntnisse präsentiert und gleichzeitig Anregungen gegeben werden das Thema Interaktion mit Softwarevisualisierungen weiter zu Erforschen.

Dazu wurden zunächst fünf Ziele identifiziert und beschrieben, die ein Benutzer bei der Interaktion mit einer Softwarevisualisierung der Struktur verfolgen kann. In einem zweiten Schritt wurden dann geeignete Kategorien von Interaktionstechniken ermittelt, die das Erreichen dieser Ziele unterstützen können. Zudem wurde ein Überblick über die Gestaltungsmöglichkeiten einer Benutzungsschnittstelle gegeben, die den Einfluss von Benutzer, System und Aufgabe auf die Interaktion berücksichtigen.

Insbesondere bei der Eignung der verschiedenen Interaktionstechniken für das Erreichen der einzelnen Ziele besteht jedoch noch immenser Forschungsbedarf. In dieser Arbeit wurden lediglich diejenigen Kategorien identifiziert, die laut ihrer Definition einen Beitrag zum Erreichen der Ziele leisten können. Wie allerdings einzelne Vertreter einer Kategorie oder die Kategorien im Vergleich zueinander den Prozess der Erkenntniserlangung unterstützen, konnte nicht untersucht werden. Dies liegt zum einen am Umfang, den eine solche Untersuchung mit sich bringt, und zum anderen an den komplexen Einflüssen der technischen Umsetzung des Systems, der Physiologie und Psychologie des Benutzers sowie den konkreten Zielen. Des Weiteren wurde die Interaktion nur im Kontext der Visualisierung der Struktur von Software untersucht, womit die anderen Aspekte, nämlich Verhalten und Evolution, eigene Ansatzpunkte zur Erforschung des Einflusses von Interaktion auf das Verständnis von Software ergeben.

Zur Umsetzung der theoretischen Erkenntnisse dieser Arbeit wurde der Prototyp von Müller erweitert, der nun automatisiert eine interaktive dreidimensionale Softwarevisualisierung der Struktur in X3D-Format generiert. Aufbauend auf der Benutzungsschnittstelle des X3D-Browsers wurde eine graphische Benutzungsschnittstelle aus X3D-Elementen entworfen, die über das Ereignismodell von X3D die Ausführung

verschiedener Interaktionstechniken ermöglicht. Die so erweiterte Benutzungsschnittstelle verfügt über mindestens einen Vertreter jeder Kategorie von Interaktionstechniken, so dass das Erreichen der fünf identifizierten Ziele unterstützt werden kann. Bei der Implementierung wurde besonders darauf geachtet, dass sich die graphische Benutzungsschnittstelle mit geringem Aufwand schnell und einfach erweitern lässt. Somit können weitere Funktionen integriert werden, mit denen der Benutzer andere Interaktionstechniken realisieren kann. In folgenden Arbeiten können darauf aufbauend weitere Untersuchungen zur Interaktion erfolgen.

Bereits auf der aktuellen Datengrundlage, dem Ecore-Modell, könnten somit diverse Interaktionstechniken jeder Kategorie umgesetzt werden. So könnten etwa die Elemente durch die Eingabe des Bezeichners über die Tastatur identifiziert werden (Kategorien *filter* und *select*) oder eine selektierte Menge von Elementen könnte verschoben oder ausgeblendet werden (Kategorie *reconfigure* bzw. *filter*). Auch das Vergrößern beziehungsweise Verkleinern von Ausschnitten wie Pakete oder Klassen ist ein Beispiel für die Erweiterung der Benutzungsschnittstelle (Kategorien *reconfigure* und *abstract/elaborate*).

Wird die Datengrundlage dagegen erweitert, ergeben sich noch mehr Möglichkeiten zur Interaktion. Ein Ansatzpunkt ist die Darstellung des Quellcodes, so dass dieser für verschiedene Methoden oder ganze Klassen eingeblendet werden kann und somit nach Bedarf Bestandteil der Softwarevisualisierung wird (Kategorie *abstract/elaborate*). Codemetriken sind eine weitere Ergänzungsmöglichkeit der Datengrundlage. (Schilbach 2010) erweitert den Visualisierungsprozess des Generators von Müller um statische Codemetriken, die als Parameter in die Darstellung, wie beispielsweise Farbe oder Größe, einfließen. Aufbauend auf dieser Erweiterung könnten interaktive Änderungen der Darstellung entsprechend der Metriken realisiert werden (Kategorien *encode* und *abstract/elaborate*) oder Elemente nach Werten oder einem Wertebereich identifiziert werden (Kategorien *filter* und *select*). Das Färben aller Methoden entsprechend ihrer Komplexität im Farbbreich von grün (einfach) zu rot (komplex) ist ein Beispiel, mit dem sich auf einen Blick kritische Stellen erkennen ließen.

Wie bereits erwähnt, wurde in dieser Arbeit nur die Interaktion mit der Softwarevisualisierung der Struktur näher untersucht und entsprechend in der Benutzungsschnittstelle des Prototyps umgesetzt. Wird die Datengrundlage entsprechend erweitert, lassen sich dagegen auch Interaktionstechniken für Ablauf und Evolution gestalten. Dabei könnten auch Animationen in X3D genutzt werden, wie beispielsweise

Kamerafahrten für den Aufrufpfad einer Funktion oder die Veränderung eines Ausschnitts im Verlauf der Entwicklung.

Einen weiter gefassten Ansatzpunkt für die Erforschung der Interaktion liefern virtuelle Realitäten. Werden bei einem überwiegend großen Teil der Softwarevisualisierungen die übliche Konfiguration aus Maus, Tastatur und Farbbildschirm verwendet, so liefert eine virtuelle Realität gleich eine ganze Reihe neuer Ein- und Ausgabegeräte, mit denen folglich auch ganz andere Interaktionstechniken realisiert werden können. Neben der Entwicklung neuer Techniken muss aber auch deren Eignung für das Erreichen der Ziele wieder Gegenstand der Untersuchung sein.

All diese aufgezeigten Ansatzmöglichkeiten belegen, dass noch viel Forschungsbedarf rund um die Interaktion mit einer Softwarevisualisierung besteht. Ausgehend von der grundlegenden Bedeutung der Interaktion für den Prozess des Verstehens, muss der Interaktion mit einer Softwarevisualisierung somit deutlich mehr Beachtung in der Forschung geschenkt werden, als dies bisher der Fall ist.

Literaturverzeichnis

Amar, R., Eagan, J. & Stasko, J., 2005. *Low-Level Components of Analytic Activity in Information Visualization*. In IEEE Computer Society, S. 15.

Bassil, S. & Keller, R.K., 2001. *Software Visualization Tools: Survey and Analysis*. International Conference on Program Comprehension.

Battista, G.D., 1999. *Graph drawing*, Prentice Hall.

Bertin, J. & Scharfe, W., 1982. *Graphische Darstellungen und die graphische Weiterverarbeitung der Information*, Walter de Gruyter.

Bohnet, J. & Döllner, J., 2005. *Konzepte der Softwarevisualisierung für komplexe, objektorientierte Softwaresysteme*, Universitätsverlag Potsdam.

Brutzman, D. & Daly, L., 2007. *X3D*, Academic Press.

Budinsky, F. et al., 2004. *Eclipse Modeling Framework*, Pearson Education.

Card, S.K., Mackinlay, J.D. & Shneiderman, B., 1999. *Readings in information visualization*, Morgan Kaufmann.

Czarnecki, K. & Eisenecker, U., 2000. *Generative programming*, Addison Wesley.

Diehl, S., 2007. *Software visualization*, Springer.

Diestel, R., 2000. *Graphentheorie*, Springer.

DIN EN ISO 9241-110, Norm DIN EN ISO 9241-110 Ergonomie der Mensch-System-Interaktion - Teil 110: Grundsätze der Dialoggestaltung.

DIN EN ISO 9241-12, Norm DIN EN ISO 9241-12 Ergonomische Anforderungen für Bürotätigkeit mit Bildschirmgeräten - Teil 12: Informationsdarstellung.

Dix, A., Finlay, J. & Abowd, G.D., 2004. *Human-computer interaction*, Pearson Education.

Eades, P., 1984. *A Heuristic for Graph Drawing*. Congressus Numerantium, 42, S. 149-160.

Foley, J.D., VanDam, A. & Feiner, S.K., 1995. *Computer graphics: principles and practice*, Addison-Wesley.

Fruchterman, T.M.J. & Reingold, E.M., 1990. *Graph drawing by force-directed placement*, Dept. of Computer Science, University of Illinois at Urbana-Champaign.

Gamma, E. & Beck, K., 2004. *Eclipse erweitern*, Pearson Education.

Gračanin, D., Matković, K. & Eltoweissy, M., 2005. *Software visualization*. Innovations in Systems and Software Engineering, S. 221-230.

Gruhn, V., Pieper, D. & Röttgers, C., 2006. *MDA*, Springer.

Haber, R.B. & McNabb, D.A., 1990. *Visualization idioms: A conceptual model for scientific visualization systems.* In: Visualization in Scientific Computing, IEEE Computer Society Press, IEEE Computer Society Press.

Heinecke, A.M., 2004. *Mensch-Computer-Interaktion*, Hanser Verlag.

ISO/IEC 19775, Information technology -- Computer graphics and image processing -- Extensible 3D (X3D).

ISO/IEC 19776, Information technology -- Computer graphics, image processing and environmental data representation -- Extensible 3D (X3D) encodings.

ISO/IEC 19777, Information technology -- Computer graphics and image processing -- Extensible 3D (X3D) language bindings.

Johnson, C. et al., 1999. *Interactive Simulation and Visualization*. Computer, S. 59-65.

Kamada, T. & Kawai, S., 1989. *An algorithm for drawing general undirected graphs*. Information Processing Letters, S. 7-15.

Klinkmüller, C., 2009. *Visualisierung von Unternehmensfähigkeiten im Kontext der Geschäftsanalyse.*Diplomarbeit Universtität Leipzig

Knight, C. & Munro, M., 1999. *Comprehension with[in] Virtual Environment Visualisations*. In IEEE Computer Society, S. 4.

Lakoff, G. & Johnson, M., 1980. *Metaphors we live by*, University of Chicago Press.

Mackinlay, J.D., 1986. *Automatic design of graphical presentations*, Stanford University.

Maletic, J.I., Marcus, A. & Collard, M.L., 2002. *A Task Oriented View of Software Visualization*. In IEEE Computer Society, S. 32-40.

Mulder, J.D., Wijk, J.J.V. & Liere, R.V., 1999. *A survey of computational steering environments*. Future Generation Computer Systems, S. 119-129.

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

Myers, B.A., 1990. *Taxonomies of Visual Programming and Program Visualization*. Journal of Visual Languages and Computing, S. 97-123.

Panas, T. et al., 2007. *Communicating Software Architecture using a Unified Single-View Visualization*. In Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems, S. 217-228.

Pietrek, G. & Trompeter, J., 2007. *Modellgetriebene Softwareentwicklung. MDA und MDSD in der Praxis* 1. Aufl., Entwickler Press.

Price, B.A., Baecker, R. & Small, I.S., 1993. *A Principled Taxonomy of Software Visualization*. Journal of Visual Languages and Computing, S. 211-266.

Reingold, E.M. & Tilford, J.S., 1981. *Tidier Drawings of Trees*. IEEE Transactions on Software Engineering, S. 223-228.

Rekimoto, J. & Green, M., 1993. *The Information Cube: Using Transparency in 3D Information Visualization*. In Proceedings of the Third Annual Workshop on Information Technologies & Systems (WITS'93), S. 125-132.

Robertson, G.G., Mackinlay, J.D. & Card, S.K., 1991. Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '91), S. 189-194.

Roman, G. & Cox, K.C., 1993. *A Taxonomy of Program Visualization Systems*. Computer, S. 11-24.

Santos, C.R.D. et al., 2000. *Metaphor-Aware 3D Navigation*. In IEEE Computer Society, S. 155.

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

Schilbach, J., 2010. *Statische Codemetriken als Bestandteil dreidimensionaler Softwarevisualisierungen*. Diplomarbeit Universität Leipzig

Schumann, H. & Müller, W., 1999. *Visualisierung*, Springer.

Shneiderman, B., 1996. *Proceedings of the 1996 IEEE Symposium on Visual Languages*. In IEEE Computer Society, S. 336.

Shneiderman, B., Maryland, U.O. & Science, C.P.D.O.C., 1983. *Direct manipulation*, Department of Computer Science, University of Maryland.

Spence, R., 2007. *Information visualization*, Pearson/Prentice Hall.

Stahl, T. et al., 2007. *Modellgetriebene Softwareentwicklung*, dPunktVerlag.

Stasko, J.T. & Patterson, C., 1991. *Understanding and characterizing program visualization systems*, Available at: citeseer.ist.psu.edu/stasko91understanding.html.

Sugiyama, K., Tagawa, S. & Toda, M., 1981. *Methods for Visual Understanding of Hierarchical System Structures*. IEEE Transactions on Systems, Man, and Cybernetics, S. 109-125.

Wetherell, C. & Shannon, A., 1979. *Tidy Drawings of Trees*. IEEE Transactions on Software Engineering, S. 514-520.

Yi, J.S. et al., 2007. *Toward a Deeper Understanding of the Role of Interaction in Information Visualization.* IEEE Transactions on Visualization and Computer Graphics, S. 1224-1231.

Anhang

A.1 X3D-Architektur: ManipulationLayer

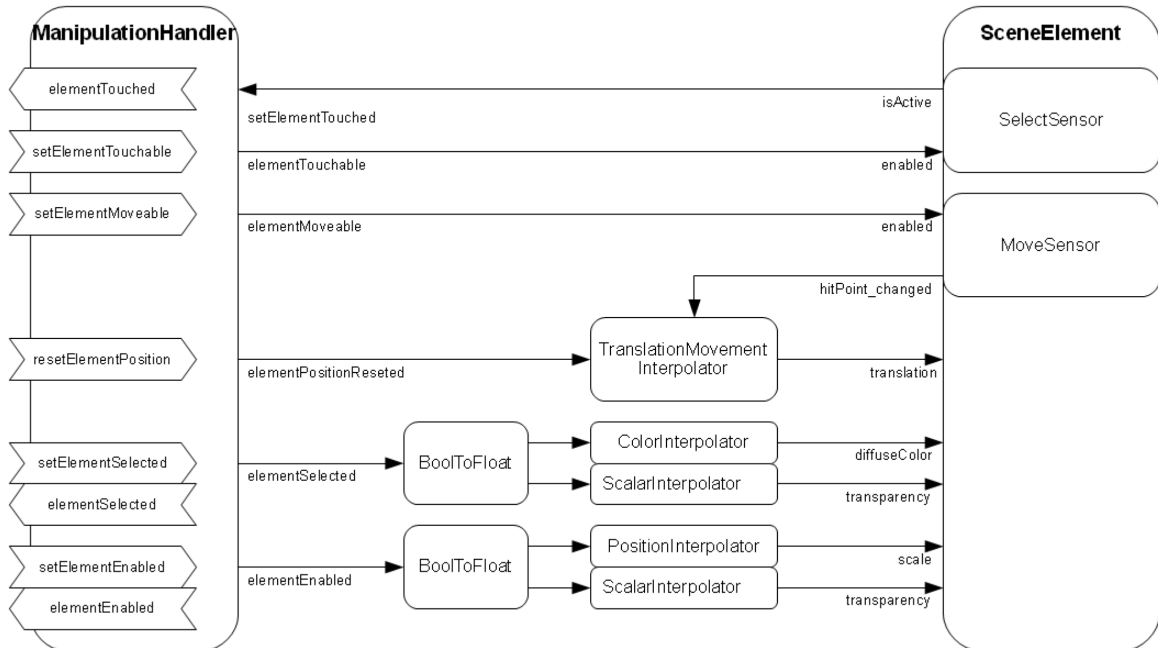

Abbildung 42: X3D-Architektur: ManipulationLayer

A.2 X3D-Architektur: SelectionLayer

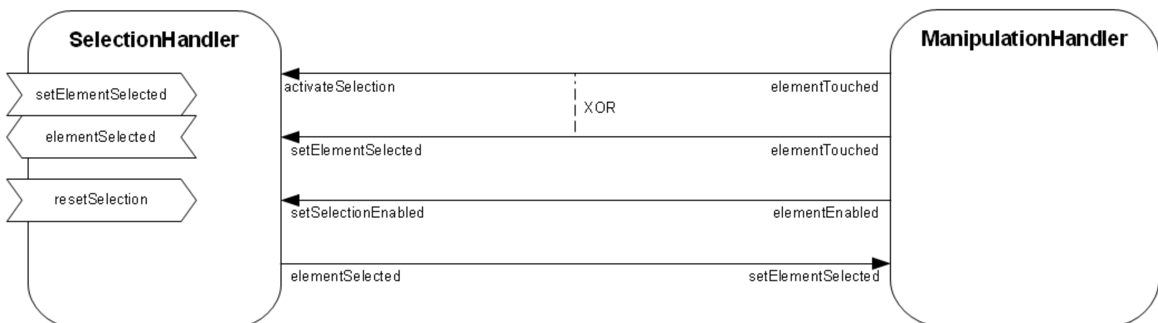

Abbildung 43: X3D-Architektur: SelectionLayer

A.3 X3D-Architektur: NavigationLayer

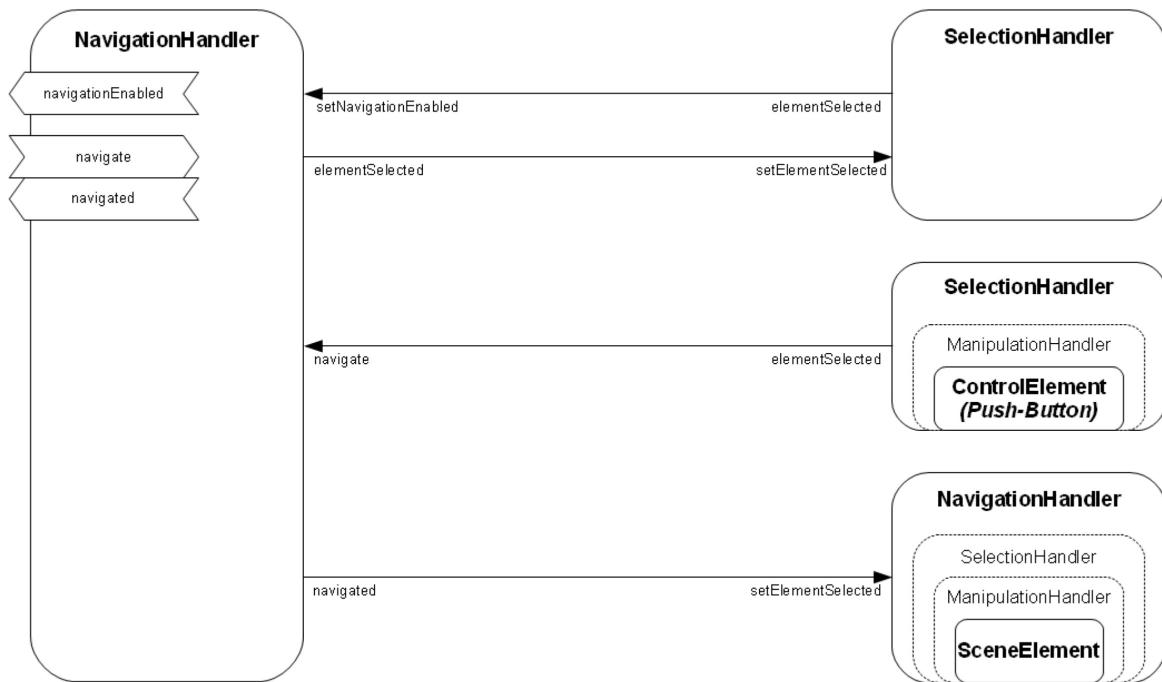

Abbildung 44: X3D-Architektur: NavigationLayer

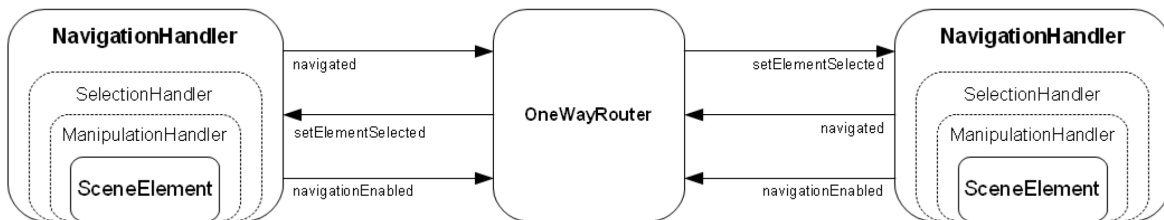

Abbildung 45: X3D-Architektur: NavigationLayer OneWayRouter