UNIVERSITÄT LEIPZIG

KONZEPTION UND PROTOTYPISCHE IMPLEMENTIERUNG EINES GENERATORS ZUR SOFTWAREVISUALISIERUNG IN 3D

RICHARD MÜLLER

Heft 4

FORSCHUNGSBERICHTE DES INSTITUTS FÜR WIRTSCHAFTSINFORMATIK

UNIVERSITÄT LEIPZIG

Forschungsberichte des Instituts für Wirtschaftsinformatik der Universität Leipzig
Heft 4

Konzeption und prototypische Implementierung eines Generators zur Softwarevisualisierung in 3D

Richard Müller

Herausgeber	Prof. Dr. Rainer Alt, Prof. Dr. Ulrich Eisenecker, Prof. Dr. Bogdan Franczyk
ISSN	1865-3189
Redaktion	Institut für Wirtschaftsinformatik der Universität Leipzig
Telefon	(0341) 97 33 720 (0341) 97 33 600
E-Mail	iwi@wifa.uni-leipzig.de
Internet	http://www.iwi.uni-leipzig.de/
Redaktionsschluss	24.11.2009

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

API	Application Programming Interface
CAD	Computer Aided Design
CVS	Concurrent Versions System
DSL	Domain Specific Language
DTD	Document Type Definition
ECMA	European Computer Manufacturers Association
EMF	Eclipse Modeling Framework
EMOF	Essential Meta-Object Facility
GEOMI	Geometry for Maximum Insight
GMT	Generative Modeling Technologies
Gravisto	Graph Visualization Toolkit
HTML	Hypertext Markup Language
IDE	Integrated Development Evironment
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
JAR	Java Archive
JDK	Java Development Tools
M2C	Model-To-Code-Transformation
M2M	Model-To-Model-Transformation
MDA	Model Driven Architecture
MDSD	Model Driven Software Development
MDV	Model Driven Visualization
MOF	Meta-Object Facility
oAW	openArchitectureWare
OCL	Object Constraint Language
OMG	Object Management Group
OSGi	Open Services Gateway Initiative
PDE	Plugin Development Environment
QVT	Query/View/Transformation
SAI	Scene Access Interface
SDK	Software Development Toolkit
SVN	Subversion
URL	Uniform Resource Locator
VGJ	Visualizing Graphs with Java
VRML	Virtual Reality Modeling Language
X3D	Extensible 3D
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language
XSL-FO	XSL Formatting Objects
XSLT	XSL Transformation

1 Einleitung

Brooks [1987] hat bereits vor über 20 Jahren in *No Silver Bullet: Essence and Accidents of Software Engineering* einige grundlegende Probleme der Softwareentwicklung identifiziert. Als Metapher verwendet er Werwölfe, die nur auf magische Weise mittels silbernen Kugeln (engl. *silver bullets*) besiegt werden können. Doch solche magischen Wunderwaffen existieren in der Realität nicht. Eines dieser Probleme bezieht sich auf die Komplexität. Software war, ist und wird auch weiterhin durch einen komplexen Charakter geprägt sein. Die Komplexität bezieht sich dabei auf verschiedene Aspekte, beispielsweise auf die Größe der Implementierung eines Systems. Viele der über die Jahre gewachsenen Altsysteme umfassen mittlerweile mehrere Millionen Quelltextzeilen. Außerdem sind in den Prozess der Entwicklung von Software viele unterschiedliche Interessensbeteiligte involviert. Entwickler, Manager und Kunden besitzen alle unterschiedliche Anforderungen an und Kenntnisse über das Softwareprodukt. Die Herausforderung besteht darin, durch Nutzung menschlicher Stärken in der Wahrnehmung und des Erlebens die Größe von Softwareentwicklungsprojekten überschaubar und ihre Komplexität auf einer gegenständlichen Ebene beherrschbar zu machen. Die Softwarevisualisierung bietet hier viel Potential. Im Mittelpunkt dieses interdisziplinären Gebietes steht die Gewinnung von Verständnis über Strukturen, Abhängigkeiten, Verhalten der Systeme und deren Komponenten. Des Weiteren gibt die Softwarevisualisierung Einsicht in Entwicklungs- und Managementprozesse, denen ein dauerhaft eingesetztes, fortgeführtes und komplexes Softwaresystem unterliegt. Schließlich bietet die Softwarevisualisierung aufgaben- und rollenspezifische Sichten auf Softwaresysteme und -entwicklungsprojekte, die den Fähigkeiten und Informationsbedürfnissen der betreffenden Personengruppen entsprechen.
Zwei Studien aus den Jahren 2001 und 2003 belegen die zunehmende Bedeutung der Softwarevisualisierung in den Bereichen der Softwareentwicklung, der Softwarewartung, des Reengineerings und des Reverse-Engineerings. Bassil und Keller [2001] befragten 107 Entwickler aus der Industrie, was sie sich von der Softwarevisualisierung versprechen. Als Vorteile wurden Geld- und Zeiteinsparung, ein besseres Verständnis von Software, Produktivitäts- und Qualitätssteigerungen, Handhabung der Komplexität und eine einfachere Fehlersuche genannt. Viele Teilnehmer kritisierten die mangelnde Integration bestehender Softwarevisualisierungswerkzeuge mit anderen Anwendungen und wünschten sich einen besseren Im- und Export von Daten und Visualisierungen. An der anderen Umfrage von Koschke [2003] nahmen 111 Forscher aus den Gebieten Softwarewartung, Reengineering und Reverse-Engineering teil. Laut dieser Studie befanden 40% der Teilnehmer die Softwarevisualisierung als absolut notwendig und weitere 42% als wichtig.

1.1 Motivation

Die Arbeit setzt an den Kritikpunkten an, die aus der 2001 von Bassil und Keller durchgeführten Umfrage hervorgingen. Demnach wurden einerseits die vom Softwareentwicklungsprozess entkoppelten Werkzeuge zur Visualisierung und andererseits die schlechten Im- und Exportmöglichkeiten von Visualisierungen durch die Befragten als negativ angesehen. Wenn sich ein Werkzeug zur Visualisierung in die Entwicklungsumgebung integriert, fallen die Kosten für

dessen Bewertung und Einsatz wesentlich geringer aus. Die erzeugten Visualisierungen sollten in einem standardisierten Format vorliegen, so dass deren Im- und Export eine weitreichende Unterstützung findet.
Ein weiteres wichtiges Akzeptanzkriterium für den praktischen Einsatz eines Softwarevisualisierungswerkzeuges bildet dessen Automatisierungsgrad [Stasko und Patterson, 1991]. Viele der vorhandenen Werkzeuge weisen einen geringen Automatisierungsgrad auf. Das bedeutet, dass die Visualisierungen gar nicht oder nur teilweise automatisch erzeugt werden. Die Montage der Visualisierung bedarf also noch eines manuellen Eingreifens. Als Beispiele seien Imsovision [Maletic et al., 2001] oder das Werkzeug von Feijs und Jong [1998] zu nennen. Die bestehenden Probleme lassen sich wie folgt zusammenfassen:

- Teilweise geringer Automatisierungsgrad des Visualisierungsprozesses, insbesondere bei Werkzeugen zur dreidimensionalen Visualisierung

- Viele Werkzeuge zur Visualisierung sind vom Entwicklungsprozess entkoppelt

- Unzureichender Im- und Export von Visualisierungen

1.2 Ziel der Arbeit

Aus den identifizierten Problemen auf dem Gebiet der Softwarevisualisierung im Allgemeinen und der dreidimensionalen Softwarevisualisierung im Besonderen ergeben sich folgende Ziele:

- Entwicklung eines Konzeptes zur vollautomatisierten Erzeugung dreidimensionaler Modelle zur Visualisierung von Softwaresystemen

- Prototypische Implementierung eines Generators zur Erzeugung dreidimensionaler Strukturmodelle von Softwaresystemen

Das zu entwickelnde Konzept umfasst insgesamt drei Aspekte: Es soll die vollautomatische Erzeugung von 3D-Modellen sowohl aus konzeptioneller Sicht als auch aus technischer Sicht beschreiben. Damit stellt es Informationen darüber bereit, was und wie etwas visualisiert werden kann. Des Weiteren soll es ein Leitfaden in Form eines strukturierten Entwicklungsprozesses beinhalten. Unter Anwendung des Konzeptes soll es möglich sein, einen Generator zu entwickeln, der aus einer Anforderungsspezifikation mittels Konfigurationswissen aus elementaren, wiederverwendbaren Komponenten ein 3D-Modell automatisch erzeugt.
Eine sehr verbreitete Umgebung zur Entwicklung von Softwaresystemen stellt Eclipse dar. Der zu entwickelnde Prototyp soll sich sich als Plugin in diese Plattform integrieren lassen. Die mit diesem Werkzeug generierten Visualisierungen müssen auf einem standardisierten Format für dreidimensionale Grafiken basieren. Damit soll eine Anwendungs-, und Plattformunabhängigkeit der Visualisierung im Hinblick auf deren Portabilität sichergestellt werden. Darüber hinaus sollen die generierten 3D-Modelle im Virtual-Reality-Labor des Instituts für Wirtschaftsinformatik an der Universität Leipzig und auf beliebigen PCs dargestellt und interaktiv erkundet werden können.

1.3 Vorgehen der Arbeit

Nach der Klärung der zentralen Ziele wird im Folgendem dargestellt, welches wissenschaftliche Vorgehen zur Zielerreichung beziehungsweise zur Problemlösung eingesetzt wird und welcher konkrete Aufbau sich daraus für diese Arbeit ergibt.
Das Vorgehen orientiert sich am konstruktionsorientierten Paradigma nach Hevner et al. [2004]. Im Mittelpunkt dieses Paradigmas steht die Entwicklung neuer und innovativer Artefakte, die auf die Lösung wichtiger und relevanter Probleme ausgerichtet sind. Die Autoren unterscheiden vier unterschiedliche Arten von Artefakten [Hevner et al., 2004, S. 2]:

- Konstrukte (engl. *constructs*) umfassen Vokabular und Symbole.
- Modelle (engl. *models*) umfassen Abstraktionen und Repräsentationen.
- Methoden (engl. *methods*) umfassen Algorithmen und Praktiken.
- Instanzen (engl. *instatiations*) umfassen implementierte und prototypische Systeme.

Die Entstehung solcher Artefakte ist grundsätzlich durch zwei Prozesse gekennzeichnet. Der Konstruktionsprozess umfasst die Entwicklung von zielgerichteten Artefakten, die die Lösung bisher unzureichend gelöster Probleme adressieren. Im Evaluationsprozess wird die Nützlichkeit des Artefaktes zur Problemlösung bewertet. Sowohl bei der Konstruktion als auch bei der Bewertung müssen forschungsmethodische Grundsätze eingehalten werden. Das bedeutet für die Entwicklung von Artefakten, dass sie *"[...] relies on existing "kernel theories" that are applied, tested, modified, and extended through the experience, creativity, intuition, and problem solving capabilities of the researcher [...]."* [Hevner et al., 2004, S. 2]. Analog muss auch bei der Bewertung auf adäquate Methoden zurückgegriffen werden. Weiterhin sollen die Artefakte einen Forschungsbeitrag leisten, indem sie klar nachvollziehbare Ergebnisse sowohl für ein technikorientiertes als auch ein managementorientiertes Publikum liefern.
In der Arbeit wird davon ausgegangen, dass bereits Lösungs- und Gestaltungsvorschläge aus der Informatik beziehungsweise Wirtschaftsinformatik sowie aus der Softwarevisualisierung vorliegen, die in geeigneter Weise adaptiert und kombiniert werden müssen. In diesem Zusammenhang sind die generative und die modellgetriebene Softwareentwicklung besonders hervor zu heben, denn sie liefern wichtige theoretische Grundlagen für die Zielerreichung. So stellt die generative Softwareentwicklung beispielsweise Vorgehensweisen zur Erstellung von Generatoren zur Verfügung. Die modellgetriebene Softwareentwicklung liefert unter anderem Werkzeuge, die die technische Umsetzung von Generatoren realisieren. Im Sinne des konstruktionsorientierten Paradigmas werden drei Arten von Artefakten entwickelt. Das Konzept einschließlich der konzeptionellen und technischen Beschreibung zur vollautomatisierten Erzeugung dreidimensionaler Modelle zur Visualisierung von Softwaresystemen entspricht einem *Modell*. Der Leitfaden zur Entwicklung eines solchen Generators stellt eine *Methode* dar. Die prototypische Implementierung eines Generators unter Anwendung des Konzeptes dient als dessen Realisierbarkeitsnachweis. Die Evaluation des Prototyps beziehungsweise der *Instanz* erfolgt unter Verwendung etablierter Methoden aus dem Bereich der Softwarevisualisierung. Sowohl bei dem Konzept als auch bei dem Prototyp wird darauf Wert gelegt, die Entwicklungsprozesse als iterative Suchprozesse darzustellen und die getroffenen Entscheidungen klar nachvollziehbar zu begründen.

1.4 Aufbau der Arbeit

Die Arbeit gliedert sich in insgesamt sechs Teile. Sie beginnt mit der Einführung in die theoretischen Grundlagen. Hier werden die Konzepte und Begriffe der Softwarevisualisierung beschrieben und es wird ein Überblick über die Paradigmen der generativen und der modellgetriebenen Softwareentwicklung gegeben.
In Kapitel 3 werden die Werkzeuge zur Implementierung des Prototyps vorgestellt. Dabei nimmt Eclipse eine Schlüsselposition ein, denn die Plugin Development Environment, das Eclipse Modeling Framework und openArchitectureWare sind Bestandteil dieser Plattform.
Es folgt in Kapitel 4 die Beschreibung von Extensible 3D (X3D). Die Wahl der technischen Basis für dreidimensionale Inhalte wird begründet und eine Einführung in X3D gegeben. Abschließend werden bestehende Ansätze auf dem Gebiet der Softwarevisualisierung mit dieser Technik betrachtet.
Im Kapitel 5 werden die vorgestellten Paradigmen mit der Softwarevisualisierung in Verbindung gebracht. Es wird ein allgemeines Konzept, bestehend aus einem Modell und einer Methode, entwickelt. Ersteres beschreibt, wie ausgehend von einem Modell mit Informationen über Struktur, Verhalten oder Evolution von Software mittels einer oder mehrerer Transformationen automatisiert ein 3D-Modell auf der Basis von X3D generiert werden kann. Die Methode erläutert den Entwicklungsprozess eines solchen Generators.
Zur Validierung des entwickelten Konzeptes aus dem vorangegangenen Kapitel wird in Kapitel 6 der im Rahmen der Arbeit entwickelte Prototyp zur Visualisierung der Struktur von Software vorgestellt und bewertet.
Im letzten Kapitel werden die erzielten Ergebnisse der Arbeit zusammengefasst. Des Weiteren wird unter Einbeziehung der gewonnenen Erkenntnisse ein Ausblick auf sinnvolle Erweiterungen beziehungsweise zukünftige Entwicklungen gegeben und damit potentieller Forschungsbedarf aufgezeigt.

2 Theoretische Grundlagen

Dieses Kapitel bildet das theoretische Fundament der Arbeit und gliedert sich in insgesamt drei Abschnitte. Zunächst wird das Gebiet der Softwarevisualisierung vorgestellt. Es schließt sich die Darstellung des generativen und des modellgetriebenen Paradigmas an. Der Begriff Paradigma kommt aus dem griechischen und bedeutet im Allgemeinen "Beispiel" oder "Muster" [Meyers Online Lexikon, 2008]. Kuhn [1962, S. X] konkretisiert diesen Begriff. Demnach bezeichnet ein *Paradigma* eine vorherrschende Denkweise in einer bestimmten Zeit, die Vorgehensweisen zur Problemlösung bereitstellt und dabei auf einem unter Fachleuten anerkannten Konsens fußt.

2.1 Softwarevisualisierung

In den nächsten Abschnitten wird das Gebiet der Softwarevisualisierung näher beleuchtet. Ausgehend von den Grundlagen erfolgt im Anschluss ein Blick auf dessen historisch gewachsenes Begriffsverständnis. Darüber hinaus werden Aufgaben der Softwarevisualisierung identifiziert, Methoden zur Strukturierung und Klassifizierung eingeführt, der Visualisierungsprozess erläutert und schließlich ausgewählte Visualisierungstechniken aufgezeigt.

2.1.1 Grundlagen und Einordnung

Wie der Name schon vermuten lässt, liegen die Wurzeln der Softwarevisualisierung auf dem Gebiet der Visualisierung. Card et al. definieren die Visualisierung folgendermaßen:

> *"The use of computer-supported, interactive, visual representations of data to amplify cognition."* [Card et al., 1999, S. 6]

Das Ziel einer Visualisierung besteht darin, Wissen über Daten zu erlangen oder zu erweitern. Durch die Überführung der Daten in eine visuelle Darstellungsform soll die Wahrnehmung des Menschen unterstützt und damit der Prozess des Verstehens gefördert werden. Das auf diese Weise erlangte Verständnis über die Daten bildet dann die Grundlage für Entscheidungen, weitere Exploration oder Erklärungen. Hierin bestätigt sich das bekannte Sprichwort *"Ein Bild sagt mehr als 1000 Worte."*.
Mackinlay [1986] definiert zwei wesentliche Kriterien, die zur Bewertung der Abbildung von Daten beziehungsweise der darin enthaltenen Informationen auf eine visuelle Repräsentation herangezogen werden können: die Expressivität oder Ausdrucksmächtigkeit (engl. *expressiveness*) und die Effektivität (engl. *effectiveness*). Die *Ausdrucksmächtigkeit* bezieht sich auf das Leistungsvermögen der Visualisierung. Die Informationen, die es zu übermitteln gilt, müssen in der Visualisierung unverfälscht dargestellt werden. Die *Effektivität* hingegen gibt an, wie gut die Visualisierung als Mittel der Informationsübertragung zum Aufbau des mentalen Modells beim Betrachter dient und damit den Prozess des Verstehens fördert.
Mentale Modelle sind individuelle kognitive Repräsentationen von Sachverhalten, mit deren Hilfe ein Mensch denkt [Hasebrook, 1995, S. 123ff]. Beim Aufbau solcher Denkmodelle verschmelzen Informationen aus mehreren Quellen. An erster Stelle steht hierbei das vorhandene Wissen über den Sachverhalt. Im Kontext der Visualisierung kommen die Informationen, die

aus der visuellen Darstellung gewonnen wurden, hinzu. Demzufolge ist es notwendig, das Fähigkeitenprofil des Betrachters zu berücksichtigen.
Neben der Ausdrucksmächtigkeit und der Effektivität einer Visualisierung spielt ein weiteres Kriterium, die *Angemessenheit* oder auch *Effizienz*, in der Praxis eine wesentliche Rolle [Schumann und Müller, 2000, S. 12]. Sie liefert Aussagen darüber, in welchem Verhältnis der Aufwand zur Erzeugung einer Visualisierung zu dem Nutzen, der mit ihr erzielt wird, steht.
Das Gebiet der Visualisierung gliedert sich in zwei Teilgebiete: die wissenschaftliche Visualisierung und die Informationsvisualisierung. Eine Möglichkeit der Unterscheidung beider Gebiete basiert auf der Art der Daten, die visualisiert werden [Card et al., 1999, S. 6-7]. Während in der wissenschaftlichen Visualisierung physische Daten verarbeitet werden, geht es in der Informationsvisualisierung um abstrakte Daten. Physische Daten sind das Ergebnis von Messungen oder Simulationen und können direkt physikalischen Prozessen zugeordnet werden. Beispiele wären Daten über den menschlichen Körper, das Wetter oder die Erde. Im Gegensatz dazu sind Daten über Geldflüsse, Geschäftsprozesse oder Meinungsumfragen eher abstrakt und besitzen keine direkten physischen Entsprechungen in der realen Welt. Diese Unterscheidung ist nicht in jedem Fall eindeutig, aber sie dient als grobe Orientierung. Im Allgemeinen ist Software durch zwei wesentliche Merkmale gekennzeichnet: Sie ist zum einen intangibel, also unfassbar, und zum anderen unsichtbar [Brooks, 1987; Gracanin et al., 2005]. Demzufolge ist Software auch abstrakt. Die Softwarevisualisierung ist somit als Teilgebiet der Informationsvisualisierung einzuordnen [Diehl, 2007, S. 3]. Dadurch ist es möglich, etablierte Konzepte der Informationsvisualisierung zu adaptieren und auf die Softwarevisualisierung zu übertragen. Darüber hinaus tangiert die Softwarevisualisierung außerdem die Gebiete Softwareentwicklung, Computergrafik, Mensch-Maschine-Kommunikation und kognitive Psychologie [Marcus et al., 2005]. Abbildung 2-1 veranschaulicht diesen interdisziplinären Charakter.

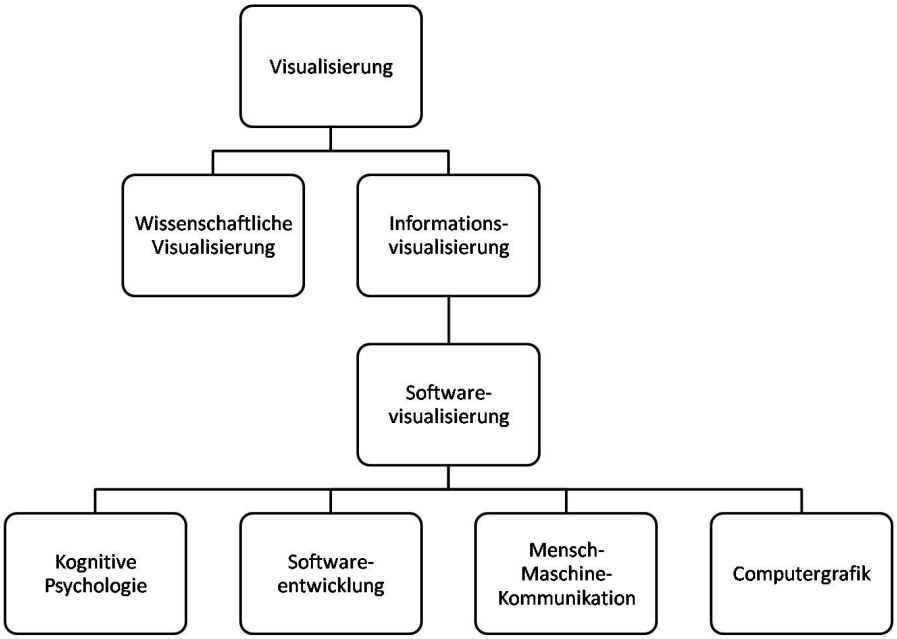


Abbildung 2-1: Einordnung der Softwarevisualisierung

2.1.2 Definition

In der Literatur existiert eine Reihe von Definitionen der Softwarevisualisierung:

> *"Program visualisation uses graphics to illustrate some aspect of the program or it's run-time execution, where the program is specified in a conventional, textual manner."* [Myers, 1990]

> *"Software visualisation is the use of the crafts of typography, graphic design, animation and cinematography with modern human- computer interaction technology to facilitate both the human understanding and effective use of computer software."* [Price et al., 1993]

> *"Program visualisation is a mapping, or transformation, of a program to a graphical representation."* [Roman und Cox, 1993]

> *"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 und Munro, 1999]

Die oben angeführten Definitionen sind sich inhaltlich sehr ähnlich. Sie unterscheiden sich lediglich in ihrer Spezifität. Die Gemeinsamkeit aller Definitionen besteht in der Beschreibung der Mittel und der Ziele der Softwarevisualisierung. Unter Verwendung graphischer Repräsentation bestimmter Aspekte von Software soll das Verständnis gefördert und gleichzeitig Einsicht in die Systeme gegeben werden. Dabei beziehen sich die Autoren speziell auf die Visualisierung von Programmen und Algorithmen, genauer gesagt, auf die Struktur und das Verhalten von Softwaresystemen. Diese Definitionen sind historisch bedeutsam, denn sie bilden das Fundament des Gebietes der Softwarevisualisierung. Jedoch decken sie nicht alle relevanten Aspekte von Software ab. Eine wesentlich weiter gefasste Definition liefert Reiss:

> *"[...] the development and evaluation of methods for graphically representing different aspects of software, including its structure, its abstract and concrete execution, and its evolution."* [Reiss, 2005]

Unter Beibehaltung der oben beschriebenen Mittel und des Zieles bezieht er einen weiteren wesentlichen Aspekt von Softwaresystemen mit ein, nämlich die Evolution. Diehl adaptiert die allgemeinere Auffassung der Softwarevisualisierung und fasst diese wie folgt zusammen:

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

Den beiden letztgenannten Auffassungen wird im Rahmen der Arbeit gefolgt. Demnach zielt die Softwarevisualisierung auf das Verstehen von Software ab, indem wesentliche Teile von Softwaresystemen respektive Artefakte sichtbar gemacht werden. Artefakte sind Zwischen- oder Endergebnisse, die während des Softwareentwicklungsprozesses entstehen. Sie werden benutzt, um projektspezifische Informationen festzuhalten oder zu übermitteln und lassen sich den verschiedenen Kategorien Struktur, Verhalten oder Evolution zuordnen [Diehl, 2007, S. 3f].

Artefakte der Kategorie "Struktur" umfassen die *statischen Informationen* eines Softwaresystems, die ohne dessen Ausführung zur Verfügung stehen. Hierzu zählen unter anderem Quellcode, Datenstrukturen, statische Aufrufgraphen, die Organisation des Systems in Module oder explizite Modelle über Struktur und Verhalten wie beispielsweise Klassendiagramme oder Sequenzdiagramme.
Die Kategorie "Verhalten" bezieht sich auf *dynamische Informationen*, die während der Programmausführung erzeugt werden. Die Ausführung kann als Abfolge von Programmzuständen gesehen werden, wobei jeder einzelne Zustand Informationen über den aktuellen Quellcode sowie die zu verarbeitenden Daten enthält. In Abhängigkeit von der Programmiersprache kann es sich dabei um Funktionsaufrufe oder die Kommunikation zwischen Objekten handeln.
Artefakte der Kategorie "Evolution" aggregieren *historisierte Informationen* über die zeitliche Entwicklung von Softwaresystemen. Diese Informationen lassen sich beispielsweise über Versionsverwaltungssysteme beziehen. Darunter fallen zum Beispiel CVS[1]- (engl. *concurrent versions system*) oder SVN[2]-Daten (engl. *subversion*).

2.1.3 Aufgaben

Es lassen sich drei verschiedene Aufgabenbereiche identifizieren, in denen die Verständnisgewinnung von Softwaresystemen eine besondere Rolle spielt [Bohnet et al., 2006, S. 4]:

- Entwurf und Entwicklung neuer Softwaresysteme
- Wartung, Erweiterung und Wiederverwendung existierender Softwaresysteme
- Verwaltung und Überwachung des Softwareentwicklungsprozesses

Die Visualisierung unterstützt analytische und konstruktive Tätigkeiten in der Softwareentwicklung. Häufig dienen CASE-Werkzeuge (engl. *computer-aided software engineering*) den Entwicklern zur Planung und bei dem Entwurf eines Softwaresystems. Eine zentrale Rolle spielen dabei Modelle, die in Diagrammform größtenteils statische und dynamische Informationen über die Systemarchitektur und die Systemausführung darstellen.
Softwarewartung und Reengineering einschließlich Reverse-Engineering zielen auf die funktionale Erweiterung eines Softwaresystems, Fehlerbehebung und Optimierung. Visualisierungen statischer, dynamischer sowie historisierter Informationen helfen, diese Aufgaben zeit- und kosteneffektiv zu bewältigen. Auf diese Weise können beispielsweise fehlerbehaftete oder funktionsüberladene Teile des Systems sichtbar gemacht und lokalisiert werden.
Die historisierten Informationen verbinden den zum Beispiel den Quelltext mit den Entwicklern und geben damit Aufschluss über die zeitliche Entwicklung des Softwaresystems. Auf dieser Basis lassen sich Visualisierungen für das Management entwickeln. Dadurch können ein Monitoring des des Softwareentwicklungsprozesses erfolgen und Entscheidungsunterstützung gegeben werden.

2.1.4 Taxonomien

Eine Taxonomie klassifiziert und gruppiert Teile eines Themengebietes anhand relevanter Charakteristiken. Zur Strukturierung der Forschung und der Anwendungen hinsichtlich der Softwarevisualisierung eignen sich Taxonomien.

[1] http://www.nongnu.org/cvs
[2] http://subversion.tigris.org

Myers [1990] führte die erste Taxonomie Ende der 80er Jahre ein. Er unterteilt die Programmvisualisierung in die Kategorien "Daten", "Quellcode" und "Algorithmus", wobei deren Visualisierung statisch oder dynamisch erfolgen konnte. In diesem Kontext bedeutet statisch, dass sich die Anzeige auf bestimmte Zustände des Systems beschränkt, während die dynamische Visualisierung eine kontinuierliche Animation bezeichnet.
Drei Jahre danach beschrieben Price et al. [1993] eine erweiterbare, hierarchische Taxonomie bestehend aus den sechs Hauptkategorien "Bereich" (engl. *scope*), "Inhalt" (engl. *content*), "Form" (engl. *form*), "Methode" (engl. *method*), "Interaktion" (engl. *interaction*) und "Effektivität" (engl. *effectiveness*).
Etwa zur selben Zeit entwickelten Roman und Cox [1993] eine ähnliche Taxonomie mit den Kategorien "Bereich", "Abstraktionsgrad" (engl. *abstraction level*), "Spezifikationsmethode" (engl. *specification method*), "Benutzungsschnittstelle" (engl. *interface*) und "Präsentation" (engl. *presentation*).
Stasko und Patterson [1991] identifizierten eine zusätzliche Eigenschaft, den "Automatisierungsgrad" (engl. *level of automation*). Dieser trifft Aussagen darüber, in welchem Verhältnis der manuelle und maschinelle Anteil zur Erzeugung der Visualisierung steht.
Ohne detaillierte Informationen über die Inhalte der angeführten Kategorien zu haben, wird bereits durch die Bezeichnungen deutlich, dass es relativ viele Redundanzen und Überschneidungen zwischen den Taxonomien gibt. Dies spiegelt den damaligen Mangel an Kooperation und Koordination in der Forschung zur Softwarevisualisierung wider. Einen Versuch, dem entgegen zu wirken, unternahmen Maletic et al. [2002]. Ihre aufgabenorientierte Taxonomie kombiniert die wichtigsten Bestandteile der oben angeführten Taxonomien und ergänzt diese um ausgewählte Sachverhalte. Die Entsprechungen der einzelnen Taxonomien sind in Tabelle 2-1 gegenübergestellt. Die Taxonomie von Maletic et al. charakterisiert Softwarevisualisierungswerkzeuge durch die fünf Dimensionen "Aufgabe" (engl. *task*), "Zielgruppe" (engl. *audience*), "Ausrichtung" (engl. *target*), "Darstellung" (engl. *representation*) und "Medium".

Maletic et al. [2002]	**Roman und Cox [1993]**	**Price et al. [1993]**
Aufgabe	-	Effektivität (Unterkategorie Zweck (engl. *purpose*))
Zielgruppe	-	Effektivität (Unterkategorie Zweck)
Ausrichtung	Bereich Abstraktionsgrad	Bereich Inhalt
Darstellung	Spezifikationsmethode Benutzungsschnittstelle Präsentation	Form Methode Interaktion Effektivität
Medium	-	Form

Tabelle 2-1: Zusammenhang verschiedener Taxonomien zur Softwarevisualisierung [Maletic et al., 2002]

Die erste Dimension spezifiziert, welche konkreten *Aufgaben* durch ein Softwarevisualisierungswerkzeug unterstützt werden sollen. Ausgehend von einem Verständnisgewinn eines oder meh-

rerer Aspekte von Software durch die Visualisierung können verschiedene Bereiche gefördert werden. Dazu gehören Entwicklungsaktivitäten (Programmierung, Debugging, Testen etc.), Wartungsaktivitäten (Fehlererkennung, Reengineering, Reverse-Engineering) oder das Softwareprozessmanagement.
Basierend auf der definierten Aufgabe ergeben sich unterschiedliche *Zielgruppen*. Maletic et al. differenzieren hier zwischen Bildung und Industrie. Im ersten Fall entsprechen die Benutzer Schülern und Lehrern beziehungsweise Professoren, Dozenten, wissenschaftlichen Mitarbeitern und Studenten. Im zweiten Fall ist die Visualisierung für die Interessensbeteiligten bei der Softwareentwicklung relevant. Hierzu zählen zum Beispiel Entwickler, Manager oder Kunden. Darüber hinaus können die Softwarevisualisierungswerkzeuge auf unterschiedliche Fähigkeitsprofile zugeschnitten sein.
In der *Ausrichtung* wird festgelegt, um welche Art von Artefakten der Software es sich bei der Visualisierung handelt. Eine zentrale Rolle nimmt dabei die Skalierbarkeit ein, das heißt, es muss darauf geachtet werden, das richtige Medium und eine entsprechende Darstellungs- beziehungsweise Visualisierungstechnik zu verwenden, um bei steigenden Datenmengen ein Optimum an Performanz und Ästethik beizubehalten.
In Abhängigkeit zu den genannten Dimensionen sowie dem noch ausstehenden Medium muss eine Art der *Darstellung* gewählt werden, die dem Benutzer die intendierte Information bestmöglich übermittelt. Der Visualisierungsprozess im Allgemeinen und das Abbilden der Informationen auf grafische Repräsentationen im Speziellen nehmen eine zentrale Rolle in der Softwarevisualisierung ein und werden deshalb in den beiden nächsten Abschnitten 2.1.5 und 2.1.6 gesondert betrachtet. Zudem sollte ein Softwarevisualisierungswerkzeug die von Shneiderman [1996] aufgestellten Kriterien erfüllen:

- Vermittlung eines Überblicks (engl. *overview*)
- Bereitstellung einer Zoom-Funktion (engl. *zoom*)
- Filter-Funktion (engl. *filter*)
- Details auf Nachfrage (engl. *details-on-demand*)
- Anzeige von Beziehung zwischen den Objekten (engl. *relate*)
- Aufzeichnung und Wiedergabe getätigter Aktionen (engl. *history*)
- Extraktion von und Anfragen an Objekte (engl. *extract*)

Es gibt eine Vielzahl an *Medien*, die für Visualisierungen verwendet werden können. Die Möglichkeiten reichen von Papier und Stiften über Computerbildschirme und Projektoren bis hin zu immersiven, virtuellen Umgebungen. Jedes Medium besitzt unterschiedliche Eigenschaften und ist demnach für verschiedene Aufgaben mehr oder weniger gut geeignet. Aus diesem Grund sollte die Wahl des Mediums entsprechend der zu realisierenden Aufgabe erfolgen.

In der Arbeit wird die Taxonomie von Maletic et al. in Verbindung mit dem von Stasko und Patterson beschriebenen Automatisierungsgrad verwendet. Demnach ergeben sich die folgenden sechs Dimensionen, deren Ausprägung durch die jeweils angegebene Frage erfasst werden kann:

- Aufgabe - Wofür wird die Visualisierung gebraucht?
- Zielgruppe - Wer wird die Visualisierung benutzen?
- Ausrichtung - Was wird visualisiert?
- Darstellung - Wie wird es dargestellt?
- Medium - Wo wird die Visualisierung dargestellt?
- Automatisierungsgrad - Wie wird die Visualisierung erstellt?

2.1.5 Visualisierungspipeline

Die Visualisierungspipeline spezifiziert eine Prozesskette zur Erzeugung visueller Repräsentationen aus Daten. Die konventionelle Pipeline von Haber und McNabb [1990] setzt sich sich aus drei grundlegenden Phasen zusammen. Den Ausgangspunkt nach der Datenerfassung bildet die Datenselektion (engl. *filtering*). Es schließt sich die Erzeugung eines Geometriemodells (engl. *mapping*) an. Im letzten Schritt werden die zur Darstellung benötigten Bilder generiert (engl. *rendering*). Dos Santos und Brodlie [2004] stellen der konventionellen Pipeline den Prozess der Datenanalyse (engl. *data analysis*) voran, um damit multivariate und multidimensionale Probleme besser beschreiben zu können. Abbildung 2-2 zeigt die resultierende Visualisierungspipeline.

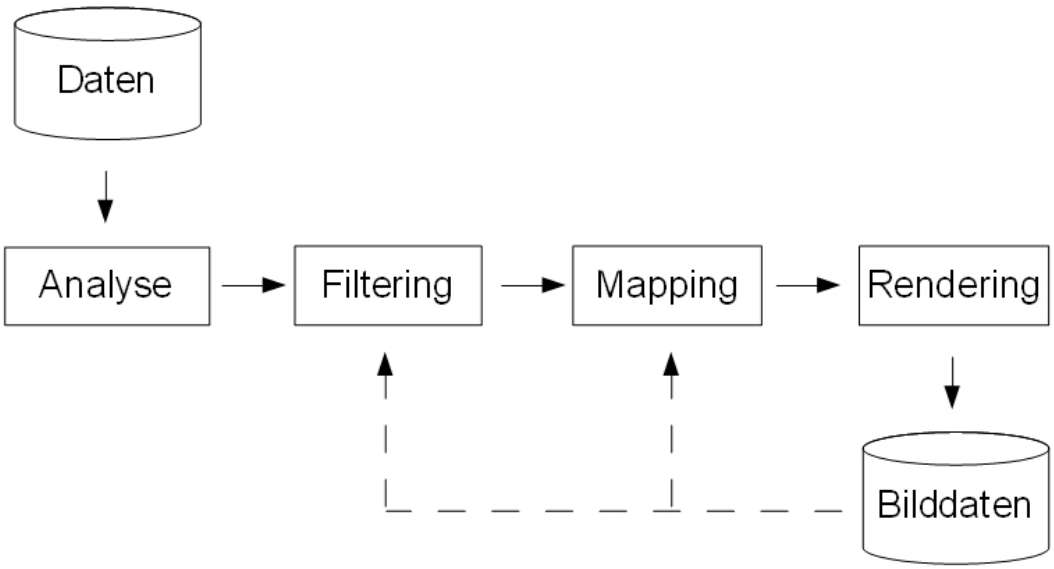


Abbildung 2-2: Visualisierungspipeline (ähnlich [dos Santos und Brodlie, 2004])

In der *Analyse* erfolgt die Aufbereitung der Daten für die nachfolgenden Visualisierungsschritte realisiert. Dazu gehören zum Beispiel Operationen zur Korrektur, Vervollständigung oder Reduzierung der Daten [Schumann und Müller, 2000, S. 15-16]. So werden fehlerhafte Daten identifiziert und von der weiteren Verarbeitung ausgeschlossen. In manchen Fällen kann es von Vorteil sein, fehlende Werte durch Interpolation zu ergänzen. Bei Daten mit sehr vielen Werten kann eine Reduzierung durch Glättung sinnvoll sein. Die aufbereiteten Daten werden dann im

nächsten Schritt, dem *Filtering*, selektiert. Hier werden Teilmengen der Daten ausgewählt, die sichtbar gemacht werden sollen. Das anschließende *Mapping* stellt das Kernstück im Visualisierungsprozess dar. Die selektierten Daten werden in diesem Schritt auf geometrische Primitive abgebildet und mit Attributen wie zum Beispiel Farbe, Größe und Position versehen. An dieser Stelle wird die Ausdrucksmächtigkeit und die Effektivität der Visualisierung im hohen Maße beeinflusst. Ursprünglich wurden beide Kriterien für zweidimensionale Abbildungen verwendet. Sie können aber auch für dreidimensionale Abbildungen adaptiert werden [Maletic et al., 2002]. Der Visualisierungsprozess schließt mit dem *Rendering* ab. Hier erfolgt die Umwandlung der geometrischen Daten in Bilddaten.
Bei interaktiven Visualisierungen hat der Benutzer die Möglichkeit, Einfluss auf den Visualisierungsprozess zu nehmen. Basierend auf bereits erzeugten graphischen Ausgaben kann der Benutzer in einem bestimmten Rahmen selbst entscheiden, was und wie er etwas visualisiert haben möchte. Die Methode wird als *computational* oder *visual steering* bezeichnet [Johnson et al., 1999; Mulder et al., 1999] und ist in Abbildung 2-2 mittels gestrichelter Pfeile dargestellt. Durch Interaktion wird die Wahrscheinlichkeit, expressive und effektive Visualisierungen zu erzeugen, wesentlich erhöht.

Im Kontext der Softwarevisualisierung bilden die Artefakte beziehungsweise die statischen, dynamischen und historisierten Informationen den Ausgangspunkt des Visualisierungsprozesses, die dann gemäß den beschriebenen Schritten verarbeitet werden.

2.1.6 Visualisierungstechniken

Die Darstellung der abstrakten Daten einschließlich ihrer Beziehungen untereinander sollte in einer natürlichen und intuitiven Weise realisiert werden. Zur Abbildung der Beziehungen zwischen Entitäten eignen sich Graphen. Um dem Betrachter den Zugang zur Visualisierung zu erleichtern, kann auf Metaphern zurückgegriffen werden.

2.1.6.1 Graphzeichnen

Graphen sind in der Informatik eine der am häufigsten verwendeten Abstraktionen. Beispiele aus der Softwareentwicklung umfassen Datenflussdiagramme, Aufrufgraphen oder Klassenhierarchien. Zur Erstellung von Datenbankmodellen dienen Entity-Relationship-Diagramme und beim Entwurf von Echtzeitsystemen kommen Petrinetze und Zustandsübergangsdiagramme zum Einsatz. Alle aufgezählten Modelle sind graphenbasiert und beschreiben die Beziehungen zwischen Elementen. Im Graphzeichnen (engl. *graph-drawing*) geht es um das automatische Zeichnen von Graphen mit dem Ziel, die Zusammenhänge von Elementen verständlich zu machen.
Graphen bestehen aus einer Menge von Knoten, die durch Kanten miteinander verbunden sein können, und werden anhand verschiedener Eigenschaften klassifiziert [Battista et al., 1999, S. 3ff, 42]. Ein Graph kann *gerichtet* oder *ungerichtet* sein, das heißt, die Kanten besitzen eine Richtung oder nicht. Gerichtete Graphen werden auch *Digraphen* genannt. Wenn Graphen einen Zyklus enthalten, werden sie als *zyklisch* bezeichnet. Ansonsten sind sie *azyklisch*. Ein Graph ist *zusammenhängend*, wenn alle Knoten direkt oder indirekt über Kanten miteinander verbunden sind. Ein *Baum* ist ein zusammenhängender azyklischer Graph. Bei einem *gewurzelten Baum* sind die Kanten üblicherweise gerichtet und es existiert ein Knoten, der nur ausgehende Kanten besitzt. Dieser Knoten wird als *Wurzel* bezeichnet. Alle anderen Knoten

besitzen nur eine eingehende Kante. Knoten, die keine ausgehenden Kanten besitzen werden als *Blätter* bezeichnet.
Nachfolgend werden zwei Verfahren zum Zeichnen von Graphen vorgestellt. Diese sind hauptsächlich von der Art der Graphen abhängig. Für ungerichtete Graphen eignen sich kraftgerichtete (engl. *force-directed*) Verfahren und bei Bäumen hierarchische (engl. *hierarchical*) Verfahren [Battista et al., 1999, S. 22f, 29f].

Kraftgerichtete Verfahren

Bei kraftgerichteten Verfahren steht ein Kräftemodell im Mittelpunkt. Dieses wirkt auf die Knoten des Graphen und konfiguriert deren Position. Hierbei kann zwischen Abstoßungs- und Anziehungskräften unterschieden werden. Knoten, die durch Kanten verbunden sind, ziehen sich an und alle anderen Knoten stoßen sich ab. Zunächst werden die Knoten zufällig platziert. Aufgrund der wirkenden Kräfte verändern sich die Positionen solange, bis ein Zustand des Kräftemodells mit der geringsten Energie gefunden ist. Zu den bekanntesten Implementierungen des kraftgerichteten Verfahrens gehören der *spring-embedder* von Eades [1984], der *Kamada-Kawai*-Algorithmus von Kamada und Kawai [1989] und das *force-directed placement* von Fruchterman und Reingold [1991]. Die kraftgerichteten Verfahren lassen sich sowohl im zweidimensionalen als auch im dreidimensionalen Raum anwenden.

Hierarchische Verfahren

Hierarchische Verfahren setzen sich aus mehreren Phasen zusammen [Sugiyama et al., 1981]. Zuerst wird jeder Knoten entsprechend seiner Lage in der Hierarchie einer bestimmten Ebene (engl. *layer*) zugeordnet. Danach werden die Knoten in einer Ebene so angeordnet, dass die Anzahl der Kantenkreuzungen minimiert wird. Anschließend werden den Knoten die Positionen zugewiesen. Die Zuweisung kann dabei von verschiedenen Kriterien wie beispielsweise Flächenminimierung, Kantenlängenminimierung oder Symmetrie beeinflusst werden. Ein klassischer hierarchischer Algorithmus zum Zeichnen eines Baumes im zweidimensionalen Raum ist der *Reingold-Tilford*-Algorithmus [Reingold und Tilford, 1981]. Dreidimensionale Äquivalente sind *cone* und *cam trees* [Robertson et al., 1991]. Der Unterschied zwischen beiden besteht darin, dass die Ebenen bei einem *cone tree* vertikal und bei einem *cam tree* horizontal ausgerichtet sind. In Abbildung 2-3 sind beide Varianten dargestellt.

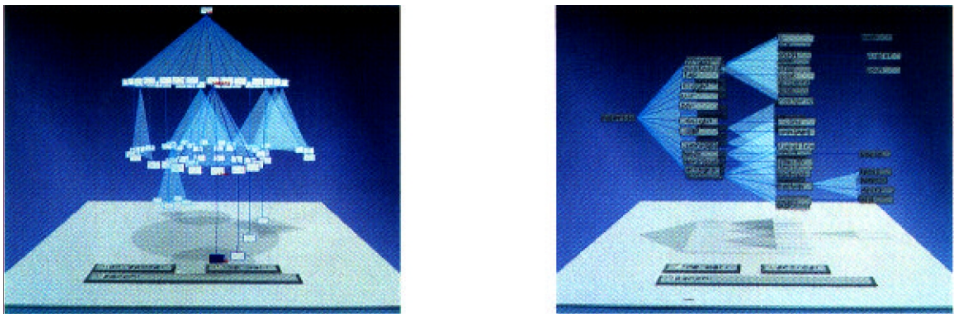


Abbildung 2-3: Cone Tree und Cam Tree [Robertson et al., 1991]

2.1.6.2 Geschachtelte Visualisierungen

Eine alternative Darstellung von Hierarchien im dreidimensionalen Raum bietet der Ansatz von Rekimoto und Green [1993]. Sie beschreiben das Prinzip der *information cubes*. Bei dieser Technik werden hierarchische Informationen als geschachtelte Würfel (engl. *cube*) dargestellt. Der äußere Würfel entspricht dabei dem Element der obersten Hierarchieebene. Die Elemente der weiteren Ebenen werden als verkleinerte Würfel innerhalb der Würfel ihrer übergeordneten Ebenen platziert. Abbildung 2-4 zeigt ein Beispiel für eine geschachtelte Visualisierung.

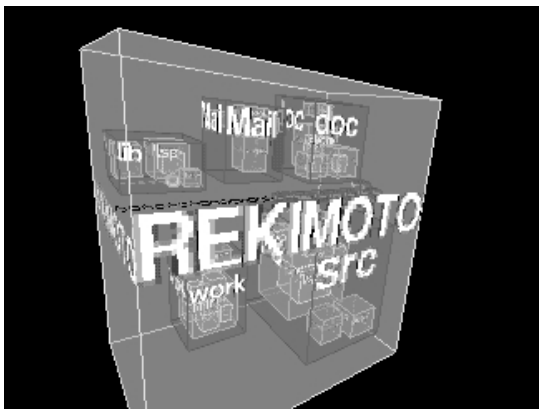


Abbildung 2-4: Information Cube [Rekimoto und Green, 1993]

2.1.6.3 Visuelle Metaphern

> *"The essence of metaphor is understanding and experiencing one kind of thing in terms of another."* [Lakoff und Johnson, 1980, S. 5]

Laut Lakoff und Johnsen besteht das Wesen einer Metapher darin, einen Sachverhalt in Begriffen eines anderen Sachverhaltes zu verstehen. Übertragen auf die Visualisierung bedeutet das, dass der zu visualisierende Sachverhalt, also die abstrakten Daten durch geeignete graphische Repräsentationen, sogenannte *visuelle Metaphern*, veranschaulicht werden [Diehl, 2007, S. 31]. Die Schreibtisch-Metapher (engl. *desktop metaphor*) gehört zu einer der bekanntesten Metaphern. Vertraute Gegenstände aus der realen Welt wie beispielsweise Ablage, Postfach oder Papierkorb besitzen in der virtuellen Welt gleichnamige Entsprechungen. Damit soll den Anwendern der Zugang zum virtuellen Arbeitsplatz erleichtert werden.
Bei visuellen Metaphern kann zwischen abstrakten und natürlichen Metaphern unterschieden werden [Gracanin et al., 2005]. Während sich *abstrakte* Metaphern aus einfachen zwei- oder dreidimensionalen, geometrischen Primitiven zusammensetzen, werden bei *natürlichen* Metaphern real existierende Gebilde (engl. *entity*) verwendet. In der Softwarevisualisierung gibt es einige auf Metaphern basierende Ansätze. Wie den Abbildungen 2-5 und 2-6 zu entnehmen ist, werden in *Imsovision* [Maletic et al., 2001] und *Software Landscape* [Balzer und Deussen, 2004] abstrakte Metaphern zur Darstellung der Struktur von Softwaresystemen eingesetzt.

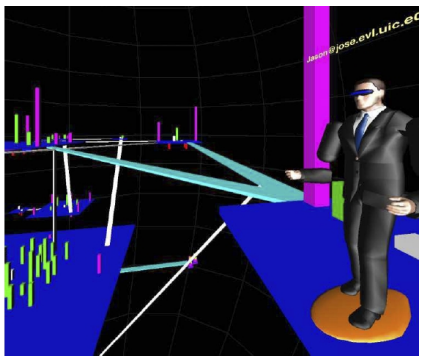


Abbildung 2-5: Imsovision [Maletic et al., 2001]

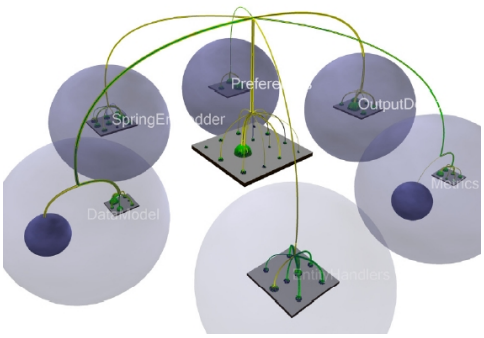


Abbildung 2-6: Software Landscape [Balzer und Deussen, 2004]

Es gibt aber auch Ansätze mit natürlichen Metaphern. In Software World [Knight und Munro, 2000], EvoSpaces [Dugerdil und Alam, 2008] und CodeCity [Wettel und Lanza, 2007] wird die Stadt-Metapher zur Visualisierung von Struktur, Verhalten und Metriken von Softwaresystemen eingesetzt. In *MetricsVisualizer3D* [Graham et al., 2004] werden Metriken auf eine Solarsystem-Metapher und in *CocoViz* [Boccuzzo und Gall, 2007] auf Alltagsgegenstände wie Tisch oder Haus abgebildet. Zur Visualisierung von zeitlichen Verläufen setzen Theron et al. die Baumring-Metapher ein [Therón, 2006]. Die Abbildungen 2-7 und 2-8 zeigen *CodeCity* beziehungsweise *MetricsVisualizer3D*.

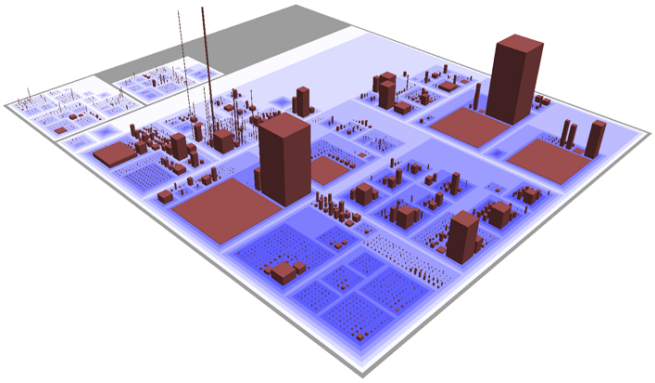


Abbildung 2-7: Code City [Wettel und Lanza, 2007]

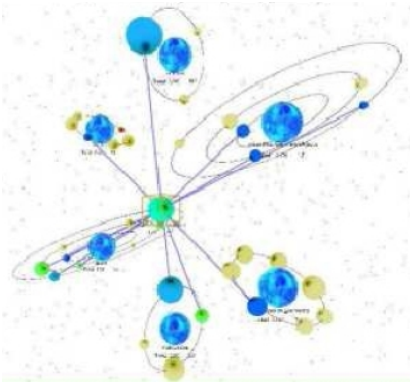


Abbildung 2-8: MetricsVisualizer3D [Graham et al., 2004]

Bei der Wahl einer visuellen Metapher spielen mindestens zwei Kriterien eine wichtige Rolle [Gracanin et al., 2005]. Zum einen muss die Metapher *konsistent* sein, das heißt, verschiedene Aspekte des Softwaresystems dürfen nicht auf dieselben Elemente einer Metapher abgebildet werden. Analog darf ein Aspekt nicht verschiedene Repräsentationen besitzen. Zum anderen muss die Metapher genügend Möglichkeiten bieten relevante Aspekte darzustellen. Dies wird als *semantische Vielfalt* (engl. *semantic richness*) bezeichnet. In diesem Zusammenhang ist es zusätzlich notwendig, die Grenzen der Metapher zu bestimmen, um deren Komplexität einzuschränken.

2.2 Generatives Paradigma

> *"Generative Programming (GP) is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge."* [Czarnecki und Eisenecker, 2000, S. 5]

Die Kernaussage dieser Definition lässt sich wie folgt zusammenfassen: Anstatt auf die Entwicklung eines einzelnen Systems zielt die generative Softwareentwicklung auf die Entwicklung einer Klasse von Systemen respektive einer Systemfamilie (engl. *system family*). Eine *Systemfamilie* umfasst eine Menge von Systemen, die in architektonischer Hinsicht einander ähnlich genug sind, um aus einer gemeinsamen Menge von Komponenten zusammengesetzt werden zu können [Czarnecki und Eisenecker, 2000, S. 31]. Mithilfe einer domänenspezifischen Sprache (engl. *domain specific language*, DSL), werden die Anforderungen an das zu erstellende Produkt beschrieben. Eine *domänenspezifische Sprache* ist spezialisiert, problemorientiert und dient zur Bestellung konkreter Mitglieder einer Systemfamilie [Czarnecki und Eisenecker, 2000, S. 137]. Die Spezifikation wird dann an einen Generator weitergegeben, der das Produkt durch die Kombination elementarer und wiederverwendbarer Komponenten automatisch erzeugt. *Komponenten* sind Bausteine, aus denen sich die verschiedenen Systeme einer Familie zusammensetzen lassen [Czarnecki und Eisenecker, 2000, S. 9]. Ein Generator ist ein Programm, das ausgehend von einer Spezifikation ein System erzeugt [Czarnecki und Eisenecker, 2000, S. 333ff]. Hierbei lassen sich vier essentielle Aufgaben eines Generators identifizieren:

- Überprüfung der Spezifikation einschließlich der Ausgabe von Warn- oder Fehlermeldungen
- Bedarfsweise Vervollständigung der Spezifikation durch Standardvorgaben
- Durchführung von Optimierungen
- Generierung

Das Generat ist ein Mitglied der Systemfamilie und basiert auf der gemeinsamen Systemfamilienarchitektur. Dabei ist es unerheblich, ob es sich um ein Zwischen- oder ein Endprodukt handelt.
Eine *Domäne* (engl. *domain*) wird im Kontext des generativen Paradigmas als abgegrenzter Wissensbereich verstanden. Der Bereich enthält einerseits fachliches Wissen, also Konzepte und Terminologien, die von Anwendern verstanden werden und anderseits technisches Wissen zur Erstellung von Systemen. Weiterhin steht eine Domäne immer in Beziehung zu den Interessensbeteiligten (engl. *stakeholder*) [Czarnecki und Eisenecker, 2000, S. 34].

2.2.1 Prozesse

In der generativen Softwareentwicklung lassen sich zwei Entwicklungsprozesse identifizieren: die Domänenentwicklung (engl. *domain engineering*) und die Anwendungsentwicklung (engl. *application engineering*) [Czarnecki und Eisenecker, 2000, S. 20 ff]. Ein weiterer Prozess, das Management, dient zur Steuerung der beiden Kernprozesse [Czarnecki, 2005, S. 328]. Dieser wird aber in der Arbeit nicht weiter betrachtet. Alle Prozesse werden iterativ durchlaufen. Das gesamte Prozessmodell ist in Abbildung 2-9 dargestellt.

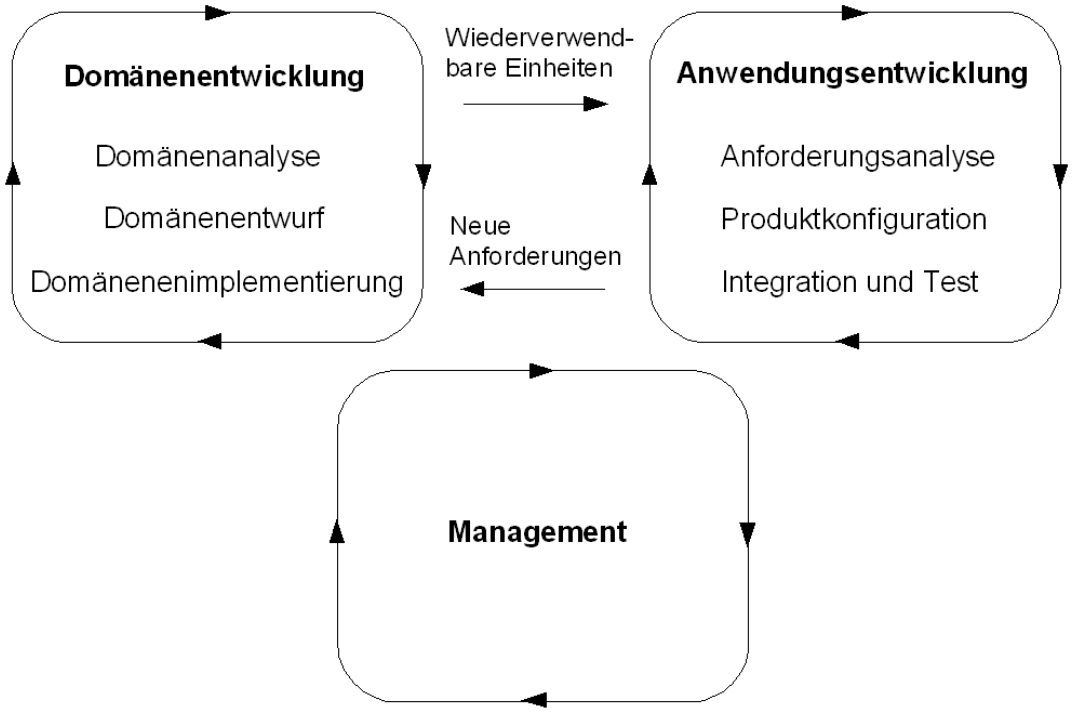


Abbildung 2-9: Prozessmodell der generativen Softwareentwicklung (ähnlich [Czarnecki, 2005, S. 329])

Die Domänenentwicklung kann als Entwicklung *zur* Wiederverwendung verstanden werden. In diesem Prozess werden die wiederverwendbaren Einheiten erstellt. Hierzu zählen unter anderen die Systemfamilienarchitektur, Komponenten, Generatoren und DSLs. In Anlehnung an die konventionelle Softwareentwicklung wird zwischen den Phasen Domänenanalyse, Domänenentwurf und Domänenimplementierung unterschieden. Das Zusatzwort *Domänen-* unterstreicht dabei, dass es sich nicht um ein Einzelsystem sondern um eine Familie von Systemen handelt. In der Domänenanalyse (engl. *domain analysis*) wird der Umfang der Domäne festgelegt und alle relevanten Informationen über die Domäne werden in einem Domänenmodell (engl. *domain model*) strukturiert erfasst. Das Domänenmodell repräsentiert die gemeinsamen und variablen Eigenschaften der Systeme aus der Domäne sowie die Abhängigkeiten zwischen den variablen Eigenschaften. Weiterhin definiert es die Semantik der Eigenschaften und der Konzepte. Das Domänenmodell besitzt im Allgemeinen die folgenden Bestandteile:

- Domänendefinition - Dient der Ein- und Abgrenzung einer Domäne.
- Domänenlexikon - Definiert das Vokabular einer Domäne.
- Konzeptmodelle - Beschreiben die Konzepte einer Domäne beispielsweise in Form von Diagrammen oder rein textuell.
- Merkmalmodelle - Aggregieren Informationen über gemeinsame und variable Merkmale einer Domäne sowie deren Abhängigkeiten.

Merkmalmodelle (engl. *feature models*) eignen sich sehr gut zur Abbildung von Gemeinsamkeiten und Variabilität von Konzepten im Rahmen der Domänenanalyse. In dieser Arbeit wird

darauf nicht näher eingegangen. Eine detaillierte Beschreibung ist zum Beispiel in [Czarnecki und Eisenecker, 2000, S. 88ff] zu finden.
Die Ergebnisse aus der Domänenanalyse bilden die Grundlage für den Domänenentwurf (engl. *domain design*). Dieser deckt einerseits die Entwicklung einer gemeinsamen Systemfamilienarchitektur und anderseits die Anfertigung eines Produktionsplanes ab. Der Produktionsplan enthält Informationen darüber, wie die einzelnen Komponenten spezifiziert und zu einem individuellen System zusammengesetzt werden können. Die generative Softwareentwicklung zielt dabei auf eine vollautomatische Montage. In der Domänenimplementierung (engl. *domain implementation*) werden die wiederverwendbaren Einheiten, also Architektur, Komponenten und Produktionsplan implementiert. Zum Produktionsplan gehören unter anderen domänenspezifische Sprachen und Generatoren. Der Produktionsplan und die Komponenten werden mit geeigneten Techniken umgesetzt, die durch eine gewählte Technikprojektion vorgegeben werden.
Die Anwendungsentwicklung kann als Entwicklung *mit* Wiederverwendung aufgefasst werden. Die wiederverwendbaren Einheiten aus dem Prozess der Domänenentwicklung werden benutzt, um Systeme zu erzeugen. In der Analysephase werden die Anforderungen erfasst und damit ein System spezifiziert. Werden die Anforderungen von dem bisherigen Domänenmodell nicht abgedeckt, so wird der Prozess der Domänenentwicklung unter Berücksichtigung dieser erneut durchlaufen. Schließlich wird das System entsprechend der Konfiguration mit einem Generator automatisch zusammengesetzt und kann danach getestet und ausgeliefert werden.

2.2.2 Generatives Domänenmodell

Das Ergebnis der Domänenentwicklung ist ein generatives Domänenmodell. Es besteht aus den drei Teilen Problemraum (engl. *problem space*), Lösungsraum (engl. *solution space*) und Konfigurationswissen (engl. *configuration knowledge*) [Czarnecki und Eisenecker, 2000, S. 131f]. Abbildung 2-10 veranschaulicht das generative Domänenmodell.

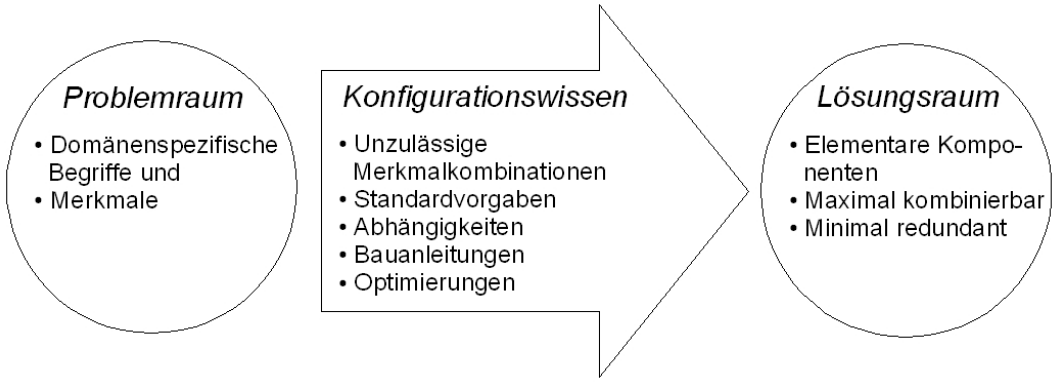


Abbildung 2-10: Elemente des generativen Domänenmodells (ähnlich [Czarnecki und Eisenecker, 2000, S. 132])

Der Problemraum beinhaltet Begriffe und Merkmale zur Spezifikation von Systemfamilienmitgliedern in Form einer oder mehrerer DSLs. Der Lösungsraum umfasst die Implementierungskomponenten und eine gemeinsame Systemfamilienarchitektur. Die Komponenten sollten minimal redundant und maximal kombinierbar sein. Das Konfigurationswissen bildet die Elemente des Problemraumes auf die Elemente des Lösungsraumes ab und ist in der Regel als Genera-

tor implementiert. Der Abbildungs- beziehungsweise Generierungsprozess berücksichtigt Informationen über unzulässige Merkmalkombinationen, Standardvorgaben sowie Abhängigkeiten gemäß den Bauanleitungen. Darüber hinaus werden bei Bedarf Optimierungen durchgeführt.
Zwischen dem Problem- und dem Lösungsraum gibt es verschiedene Abbildungsvarianten, die in Abbildung 2-11 dargestellt sind [Czarnecki, 2005, S. 332f]. Das generative Domänenmodell kann rekursiv durchlaufen werden, das heißt, der Lösungsraum eines Modells bildet gleichzeitig den Problemraum eines anderen. Damit ergibt sich eine verkettete Abbildung wie in (a). Weiterhin können mehrere Problemräume entweder aus zusammengesetzten DSLs wie in (b) oder aus alternativen DSLs wie in (d) auf einen Lösungsraum abgebildet werden. Andererseits kann ein Problemraum auf mehrere sich ergänzende wie in (c) oder alternative Lösungsräume wie in (e) abgebildet werden.

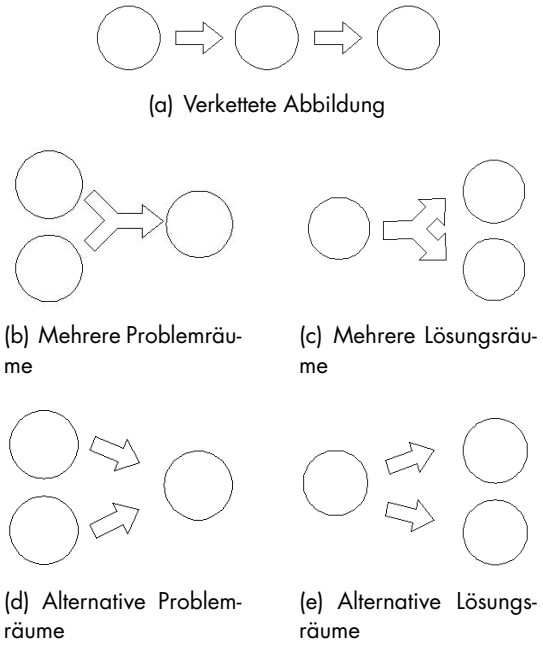


Abbildung 2-11: Verschiedene Abbildungsvarianten zwischen Problem- und Lösungsraum [Czarnecki, 2005, S. 333]

2.2.3 Technikprojektion

Die Elemente des generativen Domänenmodells können mit verschiedenen Techniken realisiert werden [Czarnecki, 2005, S. 335]. Die Abbildung der Elemente auf ein Paradigma, eine Programmiersprache oder eine Plattform wird als Technikprojektion (engl. *technology projection*) bezeichnet.
DSLs können beispielsweise in die Programmiersprache C++ eingebettet sein und besitzen damit textuellen Charakter. Sie können aber auch dialogbasiert sein. Für Generatoren können Schablonen- und Frameprozessoren oder Transformationssysteme verwendet werden. Komponenten können als Klassen oder Funktionen, als generische Komponenten wie in der *Standard Template Library* von C++ oder durch Komponentenmodelle wie zum Beispiel *JavaBeans*,

ActiveX oder *CORBA* implementiert werden. Weitere Technikprojektionen sind in [Czarnecki, 2005, S. 335] beschrieben.

2.3 Modellgetriebenes Paradigma

> *"Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen."* [Stahl et al., 2007, S. 11]

Im Kontext des modellgetriebenen Paradigmas werden Modelle zu zentralen Artefakten der Softwareentwicklung. Sie dienen nicht mehr nur zu Dokumentationszwecken wie bei der modellbasierten Softwareentwicklung sondern erhalten die Bedeutung von Quelltext [Stahl et al., 2007, S. 3]. Modelle sind formal so spezifiziert, dass aus ihnen automatisch Implementierungen von Teilen oder ganzen Softwaresystemen durch werkzeuggestützte Transformationen erzeugt werden können.
Zunächst werden die konzeptionellen Grundlagen des modellgetriebenen Vorgehens erläutert. Dafür wird ein Modellbegriff eingeführt und die Beziehung zwischen Modellen und Metamodellen werden genauer betrachtet. Es schließt sich die Vorstellung des von der *Object Management Group* (OMG) forcierten Standards Model Driven Architecture (MDA) an, da dieser als Basis der anderen modellgetriebenen Ansätze verstanden werden kann. Eine pragmatische Umsetzung von MDA bildet MDSD. Hier sind insbesondere Transformationen und deren technische Umsetzung mittels Werkzeugen von Bedeutung. Abschließend wird der Ansatz zur modellgetriebenen Visualisierung, der auf MDA beziehungsweise MDSD aufbaut, vorgestellt.

2.3.1 Grundlagen der Modellierung

Im Rahmen der Softwareentwicklung wird die Modellbildung eingesetzt, um einen Gegenstandsbereich in einem Beschreibungsmodell abzubilden [Strahringer, 1998]. Im Folgendem bezeichnet ein *Modell* die Beschreibung eines Systems oder eines Systemteils unter Verwendung einer wohldefinierten Sprache. Eine wohldefinierte Sprache umfasst eine Sprache mit einer formal definierten Syntax in Verbindung mit einer dazu angegebenen Semantik [Kleppe et al., 2003, S. 16]. Die Syntax dient zur Deklaration der Modellelemente und beschreibt deren Beziehungen untereinander. Dabei weist die Semantik der Kombination dieser Modellelemente eine Bedeutung zu.
Unter dem Begriff Metamodell kann vereinfachend ein Modell verstanden werden, das ein anderes Modell beschreibt. Diese Definition ist aber noch nicht hinreichend genau. Strahringer [1998] entwickelte ausgehend von der Sprachstufentheorie der Logik einen konkreteren Metamodellbegriff. Sie bezeichnet den Vorgang der Bildung von Metamodellen als *Metaisierung*. Je nachdem welcher Aspekt, in dem Metamodell abgebildet wird, kann zwischen der sprach- und der prozessbasierten Metaisierung unterschieden werden. Für diese Arbeit ist lediglich die sprachbasierte Metaisierung von Bedeutung. Die Bildung von Modellen, die Prozesse zur Konstruktion von Modellen repräsentieren, sei an dieser Stelle nur der Vollständigkeit halber erwähnt.
Die Beziehung zwischen Metamodell und Modell beziehungsweise Metasprache und Objektsprache wird durch die Einführung von Metaebenen ausgedrückt. Abbildung 2-12 veranschaulicht diese Modellhierarchie.

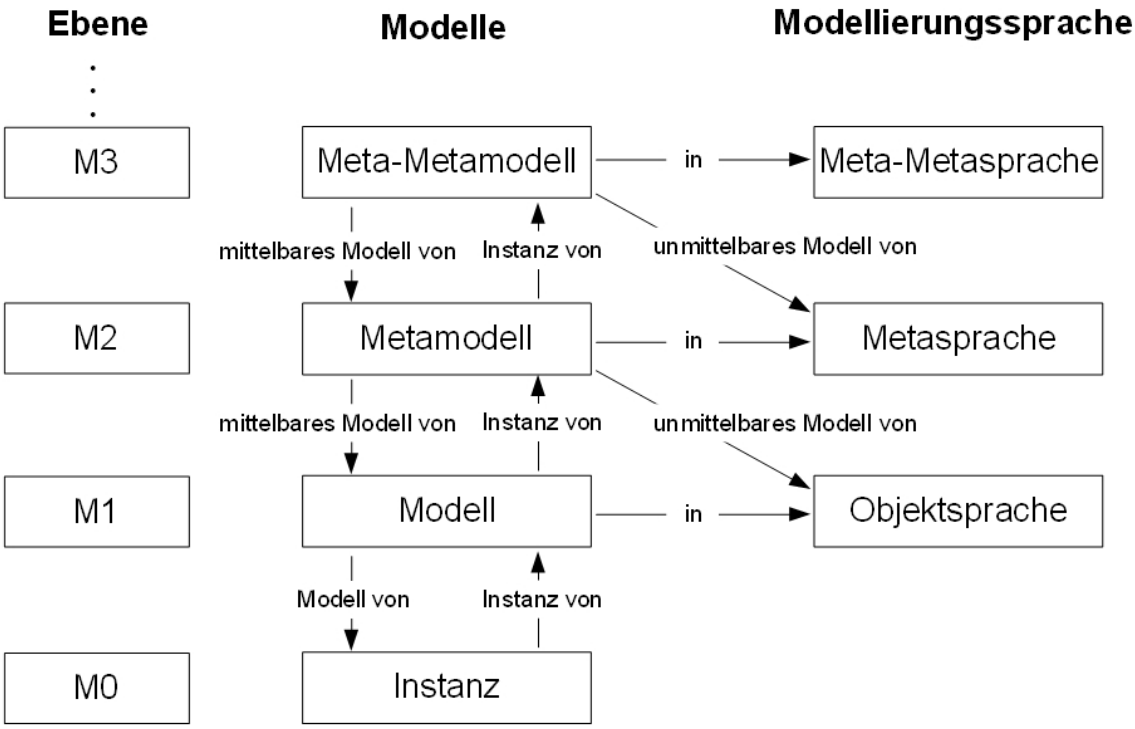


Abbildung 2-12: Zusammenhang zwischen Modell, Metamodell und Meta-Metamodell (ähnlich [Strahringer, 1998]; [Kleppe et al., 2003, S. 89])

Die Sprache, in der das Modell auf unterster Ebene formuliert ist, wird von Strahringer als Objektsprache bezeichnet. Sie wird auf einer Metaebene, also einem Abstraktionsniveau oberhalb des Modells, durch ein Metamodell beschrieben. Dieses Prinzip lässt sich rekursiv auf beliebig viele Metaebenen anwenden. Demnach legt das Metamodell eines Metamodells die Modellierungssprache zur Formulierung des Metamodells fest. Dieses wird, bezogen auf das Modell auf unterster Abstraktionsebene, als Meta-Metamodell bezeichnet. Im Kontext der MDA hat sich zur Differenzierung der verschiedenen Ebenen eine bestimmte Notation etabliert. Sie werden mit Mn, wobei gilt $n \in \{0, ..., \infty\}$, bezeichnet [Kleppe et al., 2003, S. 89]. So befindet sich auf M0 beispielsweise ein System zur Laufzeit, auf M1 das Modell, auf M2 das Metamodell und auf M3 das Meta-Metamodell. Aus softwaretechnischer Sicht kann der Zusammenhang zwischen den Modellen über den Begriff der Instanzierung gebildet werden [Kleppe et al., 2003, S. 85]. Folglich ist ein Modell die Instanz eines Metamodells, das wiederum eine Instanz des Meta-Metamodells ist. Die rekursive Deklaration von Modellen durch Metamodelle auf der nächst höheren Metaebene findet in der Praxis durch den Mechanismus der Selbstbeschreibung seine Beendigung [Frankel, 2003, S. 108]. Hierbei erklärt sich ein Modell innerhalb einer Ebene selbst. Es ist in diesem Fall sowohl Modell als auch Metamodell.

2.3.2 Model Driven Architecture

Die *Object Management Group*[3] ist ein internationales Konsortium, das 1989 gegründet wurde und dem mittlerweile weltweit ungefähr 800 Firmen angehören. Die OMG forciert die Ent-

[3] http://www.omg.org

wicklung von herstellerunabhängigen und systemübergreifenden Standards zur Verbesserung der Interoperabilität und Portabilität von Softwaresystemen. Die Standards der OMG zum modellgetriebenen Ansatz werden unter dem Namen *Model Driven Architecture* zusammengefasst. Der Grundgedanke von MDA besteht darin, die Spezifikation der Systemfunktionalität von der Spezifikation ihrer Implementierung auf einer gegebenen Plattform zu trennen [Object Management Group, 2003, S. 2-2]. Es werden Richtlinien und Standards zur Verfügung gestellt, die auf eine Strukturierung der Spezifikationen in Form von Modellen abzielen. Die Trennung funktionaler und technischer Aspekte verspricht eine bessere Wartbarkeit von Softwaresystemen, soll zur Unabhängigkeit vom Technikwandel führen und die Wiederverwendbarkeit fördern.
MDA steht in engen Zusammenhang mit anderen von der OMG entwickelten Standards. Zu den wichtigsten gehören die Meta Object Facility (MOF), die Unified Modeling Language (UML) und XML Metadata Interchange (XMI).
Die fachliche Spezifikation erfolgt in einem plattformunabhängigen Modell (engl. *platform independent model*, PIM). Das PIM enthält dabei nur die fachliche Logik und abstrahiert vollständig von technischen Details. Über eine PIM-zu-PSM-Transformation wird das plattformunabhängige Modell in ein plattformspezifisches Modell (engl. *platform specific model*, PSM) überführt. Das PSM enthält dabei zusätzlich Informationen über die Zielplattform. Unter Plattformen werden im Kontext der MDA Ausführungsumgebungen verstanden, die ihre Funktionen über Schnittstellen anbieten [Object Management Group, 2003, S. 2-3]. Plattformen sind beispielsweise Betriebssysteme (Unix, Windows etc.), Programmiersprachen (C++, Java, C#) oder Middleware-Technologien (CORBA, .NET, J2EE) [Gruhn et al., 2006, S. 26]. Mit einer PSM-zu-Quelltext-Transformation wird auf Basis des PSM Quelltext generiert.
Damit die Modelle ineinander überführt werden können, müssen deren Metamodelle auf einer gemeinsamen Sprache basieren. Diese Meta-Metasprache wird durch die MOF deklariert. Sie bildet das Meta-Metamodell im Kontext der MDA und stellt eine Menge von Konstrukten, also das Vokabular, zur Erstellung von Metamodellen bereit [Frankel, 2003, S. 105]. Tabelle 2-2 ordnet die MOF in die Hierarchie ein.

Ebene	**Modelle**	**Modellierungssprache**
M3	MOF	`MOF::Class`, `MOF::Attribute`, ...
M2	UML-Metamodell	`UML::Class`, `UML::Attribute`, ...
M1	UML-Modell	`Class Person`, `Attribute name, alter`
M0	Instanz	Max Muster ist 30 Jahre

Tabelle 2-2: Metaebenen in der MDA (ähnlich [Frankel, 2003, S. 105])

Im engen Zusammenhang hierzu steht die UML als MOF-konforme Modellierungssprache. Modelle im Sinne der MDA sind in der Regel UML-Modelle. Zur Serialiserung und zum Austausch dieser Modelle wird XMI verwendet. Für die Definition von Transformationen zwischen den oben genannten Abstraktionsebenen sieht die OMG den Standard MOF Query/View/Transformation (QVT) vor, der sich derzeit in der Entwicklung befindet [Object Management Group, 2007]. QVT umfasst eine Spracharchitektur mit deklarativen und imperativen Sprachen, die die Basis für Modelltransformationen bilden sollen.

2.3.3 Modellgetriebene Softwareentwicklung

Wie im vorherigen Abschnitt dargestellt, sind noch nicht alle Standards der MDA vollständig ausgereift beziehungsweise in der Praxis umsetzbar. Stahl et al. [2007] beschreiben, aufbauend auf den Konzepten der MDA, einen pragmatischen Ansatz zur Umsetzung der modellgetriebenen Softwareentwicklung. Sie schlagen ein iteratives und agiles Vorgehen vor und vermitteln bewährte Praktiken (engl. *best practices*). Die Autoren bezeichnen MDA als Standardisierungsinitiative zum Thema MDSD [Stahl et al., 2007, S. 7]. An dieser Stelle erfolgt keine umfassende Darstellung von MDSD. Vielmehr werden nachfolgend die verschiedenen Arten von Transformationen und die Anforderungen an Werkzeuge zur Realisierung beschrieben, um damit die Grundlage für das nächste Kapitel zu schaffen.

2.3.3.1 Transformationen

Im Wesentlichen werden zwei Arten von Transformationen unterschieden: Modell-zu-Quelltext-Transformationen (engl. *model-to-code transformations*, M2C) und Modell-zu-Modell-Transformationen (engl. *model-to-model transformations*, M2M) [Stahl et al., 2007, S. 33].
Modell-zu-Quelltext-Transformationen werden eingesetzt, um aus Modellen lauffähigen Quelltext zu erzeugen. Dieser Vorgang wird auch als Generierung bezeichnet. Dazu werden Schablonen (engl. *templates*) eingesetzt, die den zu generierenden Quelltext aus Modellelementen ableiten.
Bei Modell-zu-Modell-Transformationen oder kurz Modelltransformationen werden ein oder mehrere Quellmodelle auf ein Zielmodell abgebildet. Sie lassen sich in drei verschiedene Kategorien unterteilen: Modelltransformation, Modellmodifikation und Modellverwebung (engl. *model-weaving*) [Stahl et al., 2007, S. 33, S. 199f].
Die Modelltransformation beschreibt eine Überführung eines Quellmodells in anderes Zielmodell, wobei beide Modelle ein unterschiedliches Metamodell besitzen. Es findet also ein Metamodellwechsel statt. Das Quellmodell bleibt dabei unverändert.
Im Gegensatz dazu werden bei einer Modellmodifikation Elemente des Quellmodells verändert oder hinzugefügt. Das Zielmodell entspricht also dem modifizierten Quellmodell. Das Zielmodell bleibt aber trotzdem eine Instanz des ursprünglichen Metamodells.
In einigen Fällen sind Informationen auf verschiedene Modelle verteilt, die zusammengeführt werden müssen. In diesem Zusammenhang ist es irrelevant, ob es sich um Modelle handelt, die Instanzen des selben Metamodells sind oder nicht. Die Modellverwebung verbindet mindestens zwei Quellmodelle miteinander und erzeugt daraus ein Zielmodell.

2.3.3.2 Werkzeuge

Modelle stellen die zentralen Artefakte dar und erhalten die Bedeutung von Quelltext. Dieser Wechsel der Abstraktionsebene bei der Entwicklung von Software bedingt gleichzeitig eine Anpassung der Werkzeuge zur modellgetriebenen Softwareentwicklung. Eine geeignete Werkzeugunterstützung ist zwingende Voraussetzung, um die Vorteile von Modellen zu maximieren und deren Entwicklungs- und Wartungsaufwand zu minimieren.
Kent [2002, S. 292-294] fasst die Anforderungen an solche Werkzeuge zusammen: Die optimale Werkzeugunterstützung beginnt bereits bei der Modellerstellung. Diese kann textuell oder grafisch mittels Modelleditoren erfolgen. Hierbei sollte das Werkzeug statische Syntaxprüfung sowie Vervollständigungsfunktion bieten, um den Modellierer zu unterstützen. Zudem müs-

sen Mittel bereitgestellt werden, um die Wohlgeformtheitskriterien von Modellen festlegen und überprüfen zu können. Prüfungen auf Modellebene sind unerlässlich vor und zwischen Transformationen. Nur wenn die Quellmodelle fehlerfrei sind, können daraus qualitativ hochwertige und korrekte Zielmodelle erzeugt werden. Dies trifft natürlich ebenfalls für den generierten Quelltext zu. Das Werkzeug muss weiterhin in der Lage sein, Modell-zu-Modell- und Modell-zu-Quelltext-Transformationen zu realisieren. Bei all diesen Anforderungen sollte es aber trotzdem flexibel und konfigurierbar sein. Die unterschiedlichen Aufgaben, wie beispielsweise das Prüfen oder die Transformation von Modellen, müssen sich auf einfache Art und Weise steuern und arrangieren lassen.

Verschiedene Werkzeuge, die die angeführten Anforderungen unterstützen, sind in [Czarnecki und Helsen, 2006] aufgelistet. Zu diesen zählt openArchitectureWare. Es ist seit der letzten Version in die Entwicklungsumgebung Eclipse integriert und wird im anschließenden Kapitel im Abschnitt 3.3 vorgestellt.

2.3.4 Modellgetriebene Visualisierung

Model Driven Visualization (MDV) ist eine Vorgehensweise zur Erzeugung von Visualisierungen mittels Modellen und Transformationen [Bull, 2006]. MDV baut konzeptionell auf MDSD und MDA auf und stellt eine Möglichkeit zur Formalisierung der einzelnen Prozesse der Visualisierungspipeline dar.

Den Ausgangspunkt des Ansatzes bilden Artefakte mit Information über Software. Im Idealfall liegen diese bereits in strukturierter Form vor. Andernfalls müssen geeignete Mittel zur Extraktion der relevanten Informationen bereitgestellt werden. Ziel ist es, die Informationen automatisiert in eine geeignete Repräsentation zu überführen. Als Beispiele hierfür seien Graphen für Abhängigkeitsbeziehungen, Bäume für Vererbungshierarchien oder einfache Diagramme für Metriken genannt. Um dies zu erreichen werden auf einem höheren Abstraktionsniveau Transformationen beschrieben, die die Quellmodelle mit den zu visualisierenden Informationen in Zielmodelle für bestimmte Visualisierungstechniken überführen. Eine DSL dient dabei zur Spezifikation der Abbildung des Quellmodells auf das Zielmodell. In Abbildung 2-13 ist das Architekturmodell dieses Ansatzes dargestellt.

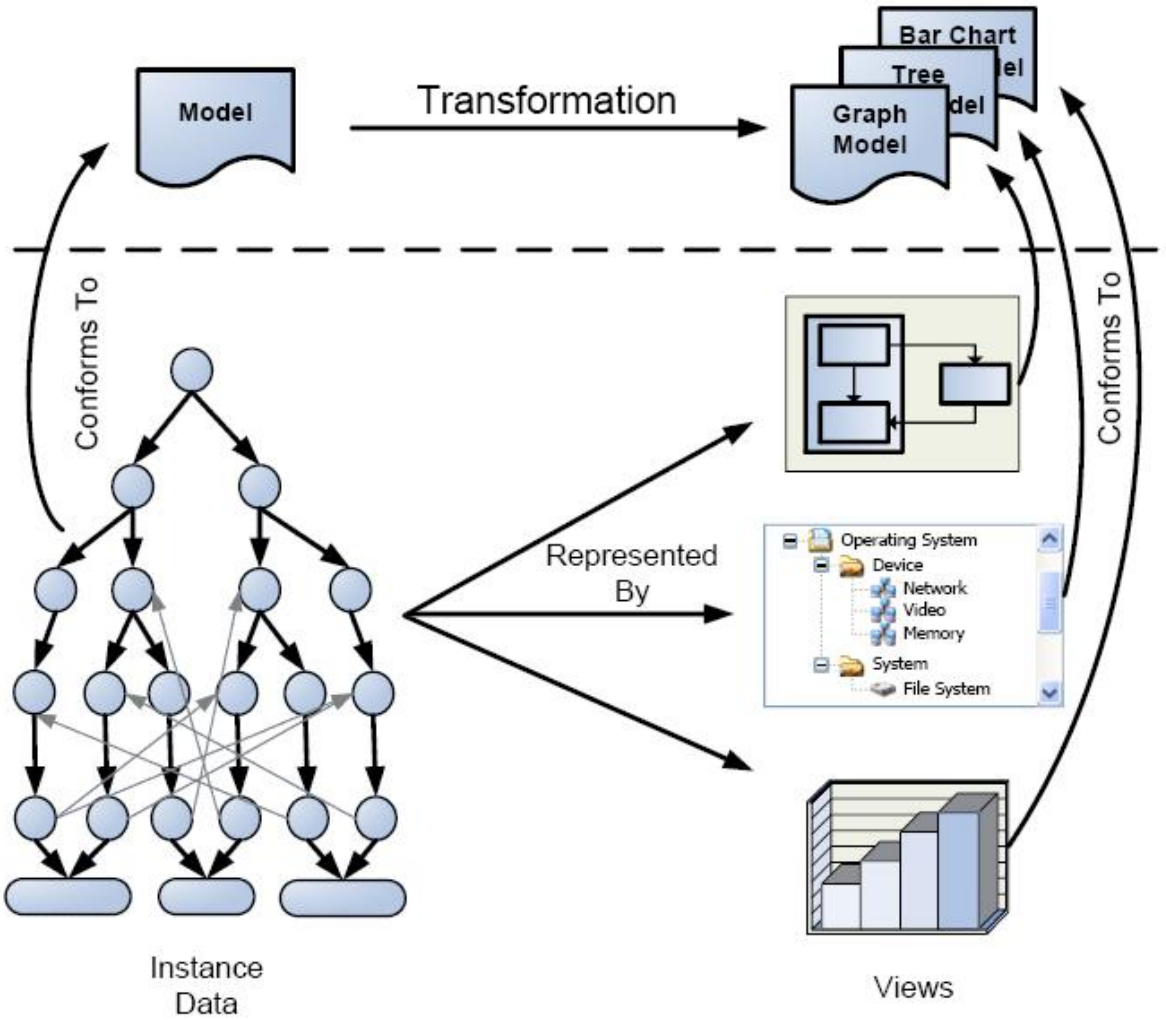


Abbildung 2-13: Architekturmodell der MDV [Bull, 2006]

3 Eclipse als Werkzeug

> *"The Eclipse platform itself is a sort of universal tool platform - it is an IDE for anything and nothing in particular."*[4]

Das oben angeführte Zitat beschreibt sehr treffend, worum es sich bei Eclipse handelt. Eclipse ist demnach eine Entwicklungsumgebung für alles Mögliche und nichts Spezielles. Die vielfältigen Einsatzmöglichkeiten ergeben sich aus der zu Grunde liegenden Plugin-Architektur. Die Aufgabe des relativ kleinen Eclipse-Kerns besteht darin, Plugins zur Laufzeit zu laden und auszuführen, wobei diese Plugins die eigentliche Funktionalität von Eclipse bereitstellen [Daum, 2008, S. 398].
Seit November 2001, nach der Offenlegung des Quelltextes durch IBM[5], steht Eclipse der Open-Source-Gemeinde zur Verfügung. Anfang 2004 kam es zur Gründung der *Eclipse Foundation*[6], einer rechtlich selbstständigen, nichtgewinnorientierten Organisation, die seitdem für die Koordination der Entwicklung von Eclipse verantwortlich ist. Das Ziel der von der Eclipse Foundation forcierten Projekte besteht darin, eine freie Entwicklungsplattform zu schaffen, die erweiterbare Frameworks, Werkzeuge und eine Laufzeitumgebung zum Erstellen, Verteilen und Verwalten von Software während ihres gesamten Lebenszyklus bietet.
Für die Implementierung des Prototyps kommen die *Plugin Development Environment* (PDE), Teile des *Eclipse Modeling Framework* (EMF) und *openArchitectureWare* (oAW) zum Einsatz. Auf diese Projekte wird in den nächsten Abschnitten näher eingegangen.

3.1 Plugin Development Environment

Die PDE gehört neben der *Eclipse-Plattform* und den *Java Development Tools* (JDT) zu dem *Software Development Toolkit* (SDK) von Eclipse. Die Umgebung liefert alle notwendigen Mittel zur Entwicklung von Plugins.

3.1.1 Plugins

Ein Plugin besteht im Minimalfall aus einem Manifest (`META-INF/MANIFEST.MF`), in dem Informationen für dessen Ausführung enthalten sind. Hierunter zählen Name und Version des Plugins sowie andere zum Ablauf benötigte Plugins. Der Ablaufkern basiert auf der Open Services Gateway Initiative (OSGi), die das Hinzufügen und Entfernen von Plugins zur Laufzeit arrangiert [Daum, 2008, S. 683 ff]. Im erweiterten Manifest (`plugin.xml`) sind Erweiterungen (engl. *extensions*) des Plugins und Erweiterungspunkte (engl. *extension points*) für andere Plugins definiert. Über Erweiterungen wird die zusätzliche Funktionalität, die ein Plugin mit sich bringt, in die Plattform eingebunden. Im Gegensatz dazu stellen Erweiterungspunkte Integrationsmöglichkeiten für andere Plugins dar. Weitere optionale Bestandteile von Plugins sind

[4] http://www.eclipse.org/platform/overview.php
[5] http://www.ibm.com/de
[6] http://www.eclipse.org/org

der Java-Bytecode zur Realisierung der Funktionalität, üblicherweise als Java-Archiv (JAR-Datei) enthalten, und Ressourcen wie zum Beispiel Icons, Hilfeseiten oder internationalisierte Zeichenketten. Bei der Entwicklung von Plugins spielen noch zwei weitere Konzepte eine wesentliche Rolle: Fragmente und Features.

3.1.2 Fragmente

Ein Fragment ist Bestandteil eines bestimmten Plugins. Es dient dazu, das Plugin durch zusätzliche Inhalte oder Funktionalität anzureichern. So können beispielsweise Sprachpakete oder andere Ressourcen nachträglich dem fertig entwickelten Plugin hinzugefügt werden. Im Wesentlichen deckt sich der Aufbau von Fragmenten mit dem bereits beschriebenen Aufbau von Plugins. Der grundlegende Unterschied liegt darin, dass der Lebenszyklus eines Fragments vom dazugehörigen Plugin gesteuert wird [Daum, 2008, S. 501]. Darüber hinaus spezifiziert das erweiterte Manifest (`fragment.xml`) des Fragments die Kopplung beider Komponenten.

3.1.3 Features

Ein Feature hingegen bündelt zusammengehörige Plugins und deren Fragmente zu einem gesamten Produkt. Ein Beispiel für ein Feature sind die JDT, in denen ein Quelltexteditor für Java, ein Debugger und eine Konsole zusammengefasst sind. Das Manifest (`feature.xml`) eines Features enthält neben den Referenzen auf die Plugins Update-, Copyright- und Lizenzinformationen des Produktes. Abbildung 3-1 verdeutlicht den Zusammenhang zwischen Plugins, Fragmenten und Features noch einmal.

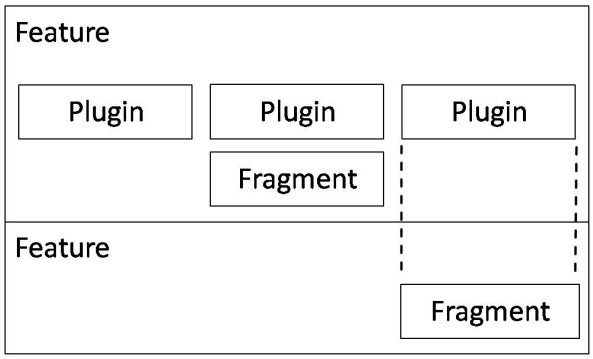


Abbildung 3-1: Zusammenhang zwischen Plugins, Fragmenten und Features (ähnlich [Daum, 2008, S. 499])

3.2 Eclipse Modeling Framework

EMF ist Bestandteil des *Eclipse Modeling Projects* und bietet unter Nutzung der Standards UML, MOF und XMI einen pragmatischen Ansatz zur Umsetzung von MDA. Im Allgemeinen dient EMF der Modellierung sowie der automatisierten Erzeugung von Java-Quelltext aus strukturierten Modellen. Näheres hierzu ist zum Beispiel in [Budinsky et al., 2004] zu finden. In dieser Arbeit beschränkt sich der Fokus der Betrachtung auf deren Metamodelle beziehungsweise das Meta-Metamodell, denn diese bilden einen möglichen Ausgangspunkt zur Erstellung des Generators für die dreidimensionale Softwarevisualisierung.

3.2.1 Ecore

Das Meta-Metamodell für alle Modelle im EMF wird als *Ecore* bezeichnet und basiert auf einer Untermenge von MOF, der Essential Meta-Object Facility (EMOF). Das Ecore-Modell ist auch ein EMF-Modell - es beendet durch die Selbstbeschreibung die Rekursion der Metamodelldefinitionen. Das Klassenmodell in Abbildung 3-2 gibt einen Überblick über die wichtigsten Bestandteile des Ecore-Modells und deren Beziehungen zueinander. Das komplette Modell mit Attributen und Methoden ist im Anhang A hinterlegt.

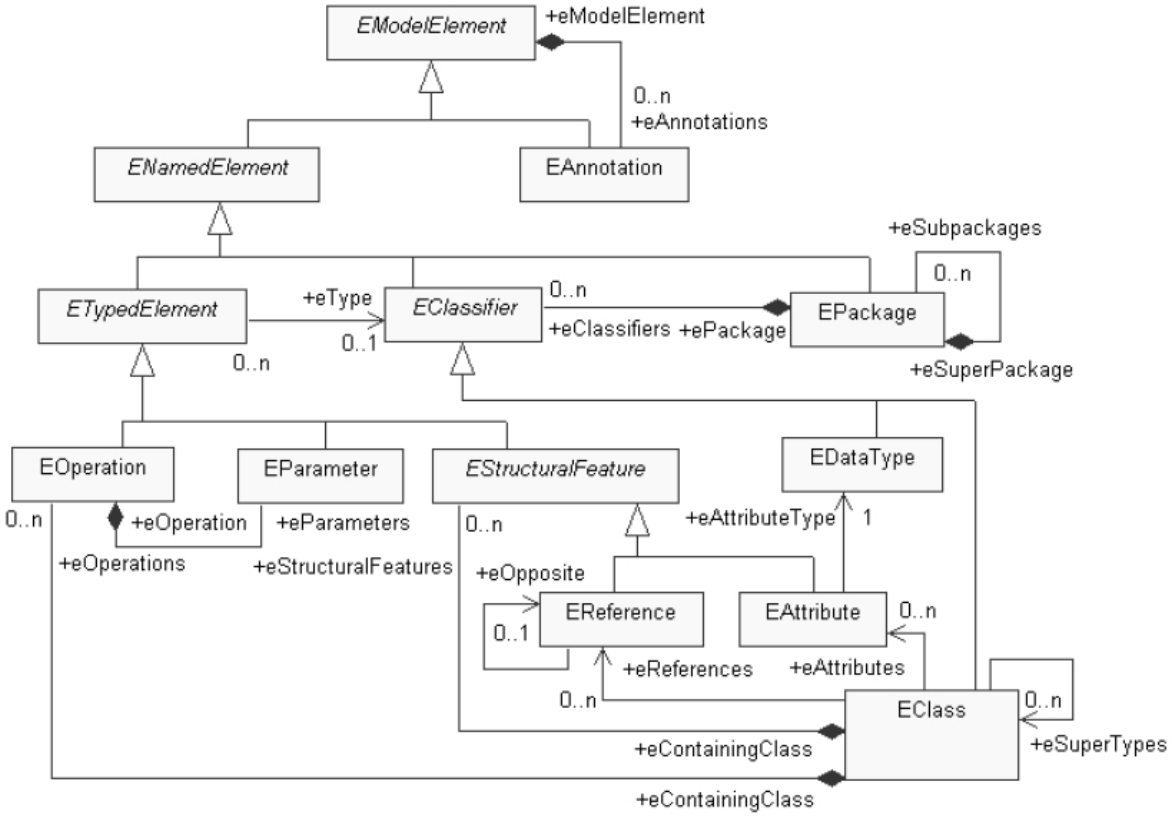


Abbildung 3-2: Klassenmodell von Ecore (ähnlich [Budinsky et al., 2004])

Das Diagramm zeigt, dass Metamodellelemente in Beziehung zueinander stehen und voneinander erben können. Alle Elemente außer `EAnnotation` besitzen einen Namen, da sie von `ENamedElement` abgeleitet sind, und werden entsprechend ihrer Semantik durch weitere Beziehungen zu anderen Elementen näher beschrieben [Budinsky et al., 2004, S. 97ff]. Die Klasse `EPackage` repräsentiert Pakete auf Metamodellebene und besitzt direkte Referenzen auf Ober- bzw. Unterpakete und indirekte Referenzen auf die sich im Paket befindenden Klassen. `EClass` definiert Klassen und hält Referenzen auf Methoden und Attribute innerhalb dieser Klassen. Des Weiteren werden Vererbungsbeziehungen durch Referenzen spezifiziert. `EReference` bildet Assoziationen oder Aggregationen zwischen Klassen ab. Der Typ einer Referenz entspricht der Klasse am Assoziationsende. Bidirektionale Beziehungen müssen in Ecore durch zwei Referenzen modelliert werden [Budinsky et al., 2004, S. 102]. `EAttribute` steht auf Metamodellebene für Attribute und besitzt eine Referenz auf deren Typ. `EOperation` repräsentiert

Methoden mit Referenzen auf die Parameter. Genau genommen handelt es sich dabei aber lediglich um Methodenköpfe beziehungsweise Schnittstellen. Über die Klasse `EAnnotation` können alle hier angeführten Modellelemente mit Kommentaren versehen werden. Primitive Datentypen aber auch komplexe Datentypen wie Klassen oder Schnittstellen werden auf Metamodellebene durch `EDataType` dargestellt.

3.2.2 Core-Modell

Instanzen des Ecore-Modells werden als Core-Modelle bezeichnet [Budinsky et al., 2004, S. 95]. Im Folgenden werden die Begriffe Ecore-basiertes Modell und Core-Modell synonym verwendet. Zur Erstellung von Core-Modellen stehen im EMF vier Alternativen zur Verfügung. Entweder erfolgt die Erzeugung durch den Import von UML-Modellen oder von XML-Schemata. Darüber hinaus kann aus annotiertem Java-Quelltext ein Core-Modell generiert werden. Die in das Core-Modell aufzunehmenden Klassen, Methoden bzw. Attribute sind hierfür mit einem entsprechenden Kommentar (`@model`) zu versehen. Der durch das EMF mitgelieferte Modelleditor unterstützt aber auch die manuelle Montage eines Core-Modells. Sowohl Core-Modelle als auch das Ecore-Modell besitzen die Endung `.ecore` und werden mit XML respektive XMI serialisiert.

3.3 openArchitectureWare

Das modulare Generator-Rahmenwerk openArchitectureWare[7] bietet eine Sammlung von Werkzeugen zur Unterstützung der modellgetriebenen Entwicklung und ist Bestandteil des *Generative Modeling Technologies*-Projektes (GMT) von Eclipse. Das Werkzeug ist in Java implementiert und liegt aktuell in der Version 4.3 vor.
Im Zentrum von oAW steht die *workflow engine*, die die einzelnen Transformations- und Generatorläufe orchestriert. Sie setzt sich aus Ablauf-Komponenten (engl. *workflow components*) zusammen, die jeweils einen Teilprozess des Generators ausführen. Zu den typischen Prozessen gehören das Einlesen und Instanzieren, die Prüfung sowie die Transformation von Modellen oder das Generieren von Quelltext. Mit einem entsprechenden Modell-Instanzierer kann das Werkzeug jedes Modell einlesen. Neben einigen anderen werden Modelle auf der Basis von EMF, XML und zahlreiche Ausgabeformate von UML-Werkzeugen unterstützt. Zur Prüfung von Modellen dient eine deklarative Sprache namens "Check". Modell-zu-Modell-Transformationen und Modellerweiterungen werden mit der funktionalen Sprache "Xtend" umgesetzt. Modell-zu-Quelltext-Transformationen lassen sich in der Schablonensprache "Xpand" definieren.

3.3.1 Typsystem und gemeinsame Sprachelemente

Die Sprachen Xtend, Check und Xpand basieren alle auf dem selben Typsystem (engl. *type system*) und einer gemeinsamen Ausdruckssprache (engl. *expression language*). Dadurch können unterschiedliche Operationen auf Modellen wie zum Beispiel Prüfungen oder Transformationen mit der selben Syntax durchgeführt werden.
Die Abstraktionsschicht auf der Basis einer Programmierschnittstelle (engl. *application programming interface*, API) wird als Typsystem bezeichnet [Efftinge et al., 2008, S. 57]. Zu den wichtigsten eingebauten Typen zählen `Object`, `String`, `Boolean`, `Integer`, `Real` und

[7] http://www.eclipse.org/gmt/oaw

`Void` für Ausdrücke ohne Rückgabewert. Weiterhin gibt es Typen für Mengen, wobei zwischen `Collection`, `List` und `Set` unterschieden wird. Neben den eingebauten Typen kann das Typsystem durch Registrierung eines oder mehrerer Metamodelle um die darin definierten Typen erweitert werden. Wird zum Beispiel in einem Ecore-basierten Metamodell der Typ `EigenerTyp` in dem Paket `typen` vereinbart, dann kann über den vollständig qualifizierten Namen `typen::EigenerTyp` auf den definierten Typ zugegriffen werden [Efftinge et al., 2008, S. 57]. Das Werkzeug oAW ist dabei nicht auf Ecore beschränkt. Es werden Metamodelle auf der Basis von UML2, Java und seit der Version 4.3 zusätzlich XML unterstützt.
Die auf dem Typsystem aufbauende Ausdruckssprache ist eine Mischung aus der Programmiersprache Java und der Object Constraint Language (OCL) [Efftinge et al., 2008, S. 61ff]. Die Sprache bietet arithmetische Operatoren (+,−,⋆,/), logische Operatoren (`&&`, `||`, `!`) und Vergleichsoperatoren (==, `!=`, >, <, >=, <=). Des Weiteren existieren Operationen speziell für Mengen zur Bildung von Teilmengen oder Listen, zur Überprüfung und zur Sortierung der Elemente. Beispiele von Mengenoperationen sind in Listing 3-1 angeführt.

```
1  // Bildung von Teilmengen
2  {1, 2, 3, 4}.select(i | i > 2) // {3, 4}
3  {1, 2, 3, 4}.reject(i | i > 2) // {1, 2}
4  menge.typeSelect(String)  // Alle Elemente vom Typ String
5
6  // Bildung von Listen
7  menge.collect(e | e.name)  // Liste mit Elementnamen
8
9  // Überprüfung
10 {1, 2, 10}.forAll(i | i < 10) // Falsch
11 {1, 2, 10}.exists(i | i < 10 ) // Wahr
12
13 // Sortieren
14 menge.sortBy(e | e.name)
```

Listing 3-1: Beispiele für Mengenoperationen

Für Kontrollstrukturen stehen drei Varianten zur Verfügung. Einfache bedingte Anweisungen lassen sich mit dem ternären Operator (Bedingung `?` Wahr **`:`** Falsch) oder durch die Schlüsselwörter `if`, `then` und `else` umsetzen. Eine Mehrfachauswahl wird durch das Schlüsselwort `switch` eingeleitet. Dabei werden die einzelnen Alternativen mit `case` und der Standardfall mit `default` deklariert. Listing 3-2 veranschaulicht die bedingten Anweisungen an Beispielen.

```
1  (name != null) ? name : "Unbekannt"
2
3  if (name != null) then name else "Unbekannt"
4
5  switch (name) {
6    case "Meier" : name + " ist bekannt."
7    case "Müller" : name + " ist bekannt."
8    case "Schulze" : name + " ist bekannt."
9  default : name + " ist unbekannt."
10 }
```

Listing 3-2: Beispiele für Kontrollstrukturen

Verkettete Ausdrücke (engl. *chain expression*) werden durch den Pfeiloperator (->) miteinander verbunden [Efftinge et al., 2008, S. 67]. Die einzelnen Ausdrücke werden dann sequentiell ausgewertet, aber nur das Ergebnis der letzten Auswertung wird zurückgegeben.
In den zwei folgenden Abschnitten werden die Sprachen Xtend und Check näher betrachtet. Auf die Beschreibung von Xpand wurde verzichtet, da diese im Prototyp nicht eingesetzt wird. Weitere Informationen zu Xpand stehen in [Efftinge et al., 2008, S. 80ff].

3.3.1.1 Xtend

Mit Xtend lassen sich Erweiterungen von Modellen realisieren und Modell-zu-Modell-Transformationen durchführen [Efftinge et al., 2008, S.69ff]. Die Sprache ist rein funktional, das heißt, alle Operationen entsprechen Funktionen. Die Dateien, in denen die Operationen zusammengefasst werden, müssen die Endung `.ext` besitzen und im Java-Klassenpfad (engl. *java class path*) enthalten sein. Die Funktionen werden als Erweiterungen (engl. *extensions*) bezeichnet, da sie die Typen der Metamodellelemente mit Operationen anreichern, ohne dabei die Typdefinition zu ändern.

Erweiterungen

Eine Funktion in Xtend besteht, ähnlich wie eine Methode in Java, aus einem Funktionskopf und einem Funktionskörper. Der Funktionskopf bestimmt den Rückgabetyp, einen Funktionsnamen und besitzt eine Parameterliste. Der Funktionskörper verarbeitet die Parameter entsprechend den darin enthaltenen Anweisungen. Die Angabe des Rückgabetyps kann in den meisten Fällen entfallen, da dieser über Typinferenz (engl. *type inference*) ermittelt werden kann [Efftinge et al., 2008, S. 71f]. Eine Ausnahme bilden rekursive Funktionen. Hier muss der Rückgabetyp angegeben werden. Das Schlüsselwort `cached` vor dem Rückgabetyp führt zu einer Pufferung der Rückgabewerte [Efftinge et al., 2008, S. 72f]. Bei einem Funktionsaufruf mit identischen Parametern liefert eine solche Funktion immer dasselbe Ergebnis zurück. Damit können Leistungsverbesserung erzielt werden. Listing 3-3 zeigt in den Zeilen 2 bis 3 ein Beispiel für die Definition einer Erweiterung. Weiterhin veranschaulicht es, wie die zuvor vereinbarte Erweiterung aufgerufen werden kann.

```
1 // Definition einer Erweiterung
2 String getterName(BezeichnetesElement e):
3   "get" + e.name.firstUpper();
4
5 // Aufruf als Funktion
6 getterName(bezeichnetesElement)
7
8 // Aufruf in der member syntax
9 bezeichnetesElement.getterName()
```

Listing 3-3: Definition und Aufruf einer Erweiterung

Entweder erfolgt der Aufruf wie in Zeile 6 als normale Funktion oder wie in Zeile 9 mit der sogenannten *member syntax* [Efftinge et al., 2008, S. 71]. Bei der *member syntax* wird der erste Parameter implizit durch das Objekt ersetzt, welches den Aufruf initiiert hat. Alle weiteren Parameter können auf konventionelle Weise in der Parameterliste übergeben werden. Beide Varianten unterscheiden sich zwar syntaktisch, führen aber zum gleichen Ergebnis.

Neben den in der Ausdruckssprache verfassten Erweiterungen können Operationen auch an statische Java-Methoden delegiert werden [Efftinge et al., 2008, S. 73]. Dadurch lassen sich die Beschränkungen der Ausdruckssprache aufheben. Somit werden beispielsweise komplexere Operationen oder der Zugriff auf externe Ressourcen möglich. In Listing 3-4 ist ein Beispiel für die Verbindung einer Erweiterung mit einer statischen Java-Methode dargestellt.

```
1  // Erweiterung, die auf eine Java-Methode verweist
2  Void ausgeben(String s) :
3    JAVA erweiterungen.Konsole.ausgeben(java.lang.String);
4
5  // Java-Klasse mit Methode zur Konsolenausgabe
6  package erweiterungen;
7
8  public class Konsole {
9    public final static void ausgeben(String string) {
10     System.out.println(string);
11   }
12 }
```

Listing 3-4: Erweiterungen mit Java

Mit dem Schlüsselwort `JAVA` in Zeile 3 wird im Funktionskörper der Erweiterung auf die vollständig qualifizierte Java-Methode verwiesen. Beide Funktionen müssen dabei die gleiche Signatur besitzen. Die externe Java-Methode übernimmt die Zeichenkette und gibt sie auf der Konsole aus. Die Erweiterung kann analog zu den beschriebenen Aufrufmöglichkeiten verwendet werden.

Modell-zu-Modell-Transformationen

Im Rahmen von Modelltransformationen spielt zusätzlich zu den bereits aufgezählten Sprachelementen das Schlüsselwort `create` eine wesentliche Rolle [Efftinge et al., 2008, S. 74ff]. Eine Erweiterung mit `create` verhält sich so ähnlich wie eine Erweiterung mit `cached`, das heißt, die Rückgabewerte werden gepuffert. Der Unterschied besteht im Zeitpunkt der Objekterstellung und folglich in der Verfügbarkeit des Objektes. Während bei `cached` das Objekt erst nach dem Verlassen des Funktionskörpers in den Speicher gelegt wird, steht das Objekt bei `create` sofort nach dessen Erzeugung, also vor dem Verlassen des Funktionskörpers, zur Verfügung. Dies kann in bestimmten Fällen dazu führen, dass das Objekt nicht vollständig initialisiert ist. Anhand von Listing 3-5 soll eine Modell-zu-Modell-Transformation nachvollzogen werden. Ziel ist es, einen Teil eines Ecore-basierten Metamodells in ein anderes Metamodell zu transformieren.

```
1  import ecore;
2  import metamodell;
3
4  Paket transformation(EPackage pkg):
5    zuPaket(pkg);
6
7  create Paket zuPaket(EPackage pkg) :
8    this.klassen.addAll(pkg.eClassifieres.zuKlasse());
9
```

```
10 create Klasse zuKlasse(EClass cls) :
11  this.attribute.addAll(cls.eReferences.zuReferenz());
12
13 create Referenz zuReferenz(EReference ref) :
14  this.setzeTyp(ref.eType.zuKlasse());
```

Listing 3-5: Modell-zu-Modell-Transformation mit Xtend

Zu Beginn werden beide Metamodelle importiert. Die Funktion `transformation` in Zeile 4 kann als Einstiegspunkt der Transformation angesehen werden. Sie wird von einer Ablauf-Komponente aufgerufen und gibt nach der Transformation ein Objekt vom Typ `Paket` zurück. Die Angabe von `Paket` ist in diesem Fall optional. In den folgenden Zeilen wird die Abbildung von `EPackage` auf `Paket`, `EClass` auf `Klasse` und `EReference` auf `Referenz` beschrieben. Hierbei aggregiert ein `Paket` Elemente vom Typ `Klasse` in einer Liste namens `klassen`. Eine `Klasse` enthält Elemente vom Typ `Referenz` in einer Liste namens `attribute`, wobei eine `Referenz` einen Verweis auf die referenzierte Klasse besitzt.

3.3.1.2 Check

Die Sprache Check stellt Mittel bereit, um Modelle auf bestimmte Wohlgeformtheitskriterien zu prüfen [Efftinge et al., 2008, S. 69]. Dies kann an jeder beliebigen Stelle im Generatorlauf erfolgen wie zum Beispiel nach dem Einlesen oder nach der Transformation eines Modells. Die Datei, in der die Kriterien als Regeln definiert werden, muss die Endung `.chk` besitzen und im Java-Klassenpfad enthalten sein. In Listing 3-6 ist ein Beispiel für eine einfache Modellprüfung angeführt.

```
1  import eigene::typen;
2  extension erweiterungen::EineErweiterung;
3
4  context EigenerTyp ERROR
5  "Der Name eines Typs ist nicht gesetzt." :
6  name != null;
7
8  context EigenerTyp WARNING
9  "Der Name " + name + " ist zu kurz" :
10 name.length > 2;
```

Listing 3-6: Modellprüfung mit Check

An erster Stelle steht der Import des Metamodells. Bei Bedarf kann auf Erweiterungen, die in Xtend verfasst sind, zurückgegriffen werden. Diese werden mit dem Schlüsselwort `extension` inkludiert. Eine Regel wird mit dem Schlüsselwort `context` eingeleitet. Damit wird der zu prüfende Typ des Modells deklariert. Je nachdem, welche Bedeutung die Regel besitzt, folgt eine Auszeichnung mit `ERROR` oder `WARNING`. Jede Regel besitzt weiterhin eine Nachricht und einen booleschen Ausdruck. Evaluiert der Ausdruck zu falsch, hängt der weitere Ablauf von der Auszeichnung ab. Bei `ERROR` wird die spezifizierte Nachricht in der Konsole ausgegeben und Generatorlauf wird abgebrochen. Bei `WARNING` wird nur die Nachricht ausgegeben.

3.3.2 Workflow

Die Konfiguration der *workflow engine* erfolgt durch XML-Dateien mit der Endung `.oaw`. Sie legen fest, in welcher Reihenfolge die einzelnen Ablauf-Komponenten ausgeführt werden. Mit oAW werden bereits vorgefertigte Ablauf-Komponenten beispielsweise zum Einlesen und Instanzieren, zum Prüfen und zur Transformation von Modellen oder zur Generierung von Quelltext ausgeliefert. Es besteht aber auch die Möglichkeit, eigene Ablauf-Komponenten zu erstellen. Hierzu muss die Schnittstelle mit der Bezeichnung `WorkflowComponent` implementiert werden [Efftinge et al., 2008, S. 47].
Listing 3-7 zeigt die wesentlichen Bestandteile und Konzepte, die bei einer Ablaufbeschreibung zum Einsatz kommen.

```
1  <workflow>
2    <property name='wurzelVerzeichnis' value='.'/>
3    <property file='${wurzelVerzeichnis}/meine.properties'/>
4    <!-- ... -->
5    <component class="org.eclipse.mwe.emf.Reader">
6      <uri value="${modell}" />
7      <modelSlot value="ecoreModell" />
8    </component>
9    <component class="oaw.check.CheckComponent">
10     <metaModel class="org.eclipse.m2t.type.emf.EmfRegistryMetaModel"/>
11     <checkFile value="Checks" />
12     <emfAllChildrenSlot value="ecoreModell" />
13   </component>
14   <!-- ... -->
15 </workflow>
```

Listing 3-7: Bestandteile einer Ablaufbeschreibung

Die Ablaufbeschreibung wird mit einer XML-basierten Konfigurationssprache realisiert. Dieser liegt das *dependency injection* Muster [Fowler, 2004] zu Grunde [Efftinge et al., 2008, S. 48]. Alle Ablauf-Komponenten sind in einer `workflow`-Direktive zusammengefasst. Die einzelnen Komponenten werden in `component`-Direktiven gekapselt und besitzen eine wohldefinierte Schnittstelle. Bei der Konfiguration steht das von *Apache Ant*[8] entlehnte Konzept der Eigenschaften (engl. *property*) zur Verfügung [Efftinge et al., 2008, S. 49]. In den Zeilen 2 und 3 sind einfache beziehungsweise in Dateien ausgelagerte Eigenschaften angeführt. Nachdem eine Eigenschaft definiert wurde, kann sie überall in der Ablaufbeschreibung über `${Eigenschaftsname}` referenziert werden [Efftinge et al., 2008, S. 49]. Die Kommunikation zwischen den einzelnen Ablauf-Komponenten erfolgt über sogenannte *slots* [Efftinge et al., 2008, S. 53]. Im Prinzip sind *slots* globale Variablen, auf die alle Komponenten Zugriff haben. Sie realisieren die Weitergabe des Modells von einer Komponente zur anderen. Dies ist beispielsweise notwendig, wenn ein eingelesenes Modell wie in Zeile 7 für die weitere Verarbeitung an eine andere Komponente wie in Zeile 12 zur Prüfung weitergeben werden soll.
Es gibt vier verschiedene Varianten, einen beschriebenen Transformations- oder Generatorlauf zu starten [Efftinge et al., 2008, S. 55]. Innerhalb von *Eclipse* erfolgt der Start über das Kontextmenü *Run as* → *oAW Workflow*. Weiterhin kann die Ausführung über die Kommandozeile oder

[8] http://ant.apache.org

mit einem *Ant*-Skript angestoßen werden. Um aus einer eigenen Anwendung auf die *workflow engine* zuzugreifen, steht die Schnittstelle `WorkflowRunner` zur Verfügung.

4 Extensible 3D als 3D-Technik

In diesem Kapitel werden zunächst Techniken zur dreidimensionalen Visualisierung vorgestellt. Anschließend erfolgt die Begründung der Auswahl von X3D sowie eine Einführung in diese Technik. Den Abschluss bildet eine Betrachtung verschiedener Ansätze zur Softwarevisualisierung mit X3D.

4.1 Wahl der Technologie

Die Auswahl der Technologie zur dreidimensionalen Softwarevisualisierung wurde maßgeblich von den eingangs festgelegten Zielen beeinflusst. Demnach wird ein plattform- und anwendungsunabhängiges, standardisiertes Format für 3D-Inhalte benötigt. Im Hinblick auf die modellgetriebene Erstellung der 3D-Inhalte wäre ein XML-basiertes Format von Vorteil, da in diesem Fall das Metamodell implizit durch eine Dokumenttypdefinition (engl. *document type definition*, DTD) oder ein XML-Schema gegeben wäre. Nachfolgend werden alternative Techniken betrachtet. Im Anschluß werden diese bewertet und die Entscheidung für X3D unter Einbeziehung einer von Anslow et al. durchgeführten Evaluation begründet.

OpenGL

Die *Open Graphics Library*[9] (OpenGL) ist eine Spezifikation, die es ermöglicht über eine plattform- und programmiersprachenunabhängige API zwei- und dreidimensionale Grafiken zu erzeugen. Die Schnittstelle bietet etwa 250 Befehle, um aus einfachen Primitiven komplexe 3D-Szenen zu erzeugen. OpenGL erfordert die Implementierung einer kompletten Anwendung zur Erstellung einer Visualisierung.

Open Inventor

Open Inventor[10] ist ein freies Werkzeug zur Erzeugung von dreidimensionalen Grafiken unter Verwendung von OpenGL. Es umfasst neben einer C++-Klassenbibliothek zur Erstellung ein standardisiertes Austauschformat zur Speicherung von 3D-Inhalten. Die Programmierung von reinen OpenGL-Anwendungen ist sehr zeitintensiv. Open Inventor bietet bereits vordefinierte Objekte beispielsweise für dreidimensionale geometrische Primitive.

Java 3D

Java 3D[11] stellt über eine Java-Klassenbibliothek Schnittstellen zur Verfügung, um dreidimensionale Grafiken zu erzeugen, zu manipulieren und darzustellen. Die Darstellung kann innerhalb einer Java-Anwendung oder eines Java-Applets realisiert werden. Wie bei OpenGL muss zur Visualisierung eine komplette Anwendung implementiert werden.

[9] http://www.opengl.org
[10] http://oss.sgi.com/projects/inventor
[11] https://java3d.dev.java.net

COLLADA

COLLAborative Design Activity[12] (COLLADA) ist ein XML-basiertes intermediäres Format. Es dient zum Austausch von 3D-Inhalten zwischen unterschiedlichen Werkzeugen wie beispielsweise 3D Studio Max, Blender oder Maya. COLLADA steht wie auch OpenGL unter der Aufsicht der *Khronos Group*, einem Industriekonsortium zur Erstellung und Verwaltung von offenen Standards im Multimedia-Bereich.

Die direkte Verwendung von OpenGL oder Java 3D erscheint zu schwergewichtig. Für eine Visualisierung müssen bei beiden Techniken komplette Anwendungen implementiert werden. Außerdem liegt Java 3D-Anwendungen ein binäres Format zugrunde und sie müssen zum Ablauf in eine Anwendung oder ein Applet eingebettet werden. Das primäre Ziel der Formate von Open Inventor und COLLADA liegt im Austausch von 3D-Inhalten und nicht in deren Darstellung. X3D ist mehr als nur ein Austauschformat. Mit X3D lassen sich beispielsweise auch Animationen und Benutzerinteraktionen modellieren. Die Navigationsmöglichkeiten wie Zoom, Laufen, Fliegen, Explorieren in der dreidimensionalen Umgebung werden durch die Ablaufumgebung, sogenannten X3D-Browsern, bereitgestellt und müssen nicht programmiert werden. Weiterhin ist X3D leichtgewichtig, besitzt ein XML-basiertes Format und ist als internationaler Standard anerkannt. Anslow et al. [2008] haben X3D als Technik zur dreidimensionalen Softwarevisualisierung nach einer selbst entwickelten Taxonomie evaluiert. Die Taxonomie korrespondiert mit der im Abschnitt 2.1.4 herausgearbeiteten Taxonomie. Die Untersuchung von X3D erfolgte entlang den Dimensionen Bereich, Form und Interaktion und resultierte in der Identifikation mehrerer Vor- und Nachteile von X3D. Zu den Nachteilen zählen die Autoren eine unzureichende Kontrolle der Visualisierungen durch Benutzer, das primitive Animationsmodell und die unzureichende Unterstützung von Layout- und Filtermöglichkeiten im Rahmen des Visualisierungsprozesses. Die beiden erstgenannten Nachteile sind im Rahmen dieser Arbeit vernachlässigbar. Eine Lösung der unzureichenden Unterstützung von Layout- und Filtermöglichkeiten wird im Abschnitt 6 bei der Beschreibung des Prototyps aufgezeigt. Die Vorteile von X3D umfassen das große Portfolio an graphischen Möglichkeiten, dessen Erweiterbarkeit und dass es auf XML beruht. X3D weist nicht die Probleme der anderen Technologien auf, denn es ist nicht schwergewichtig, setzt kein binäres Format voraus und ist nicht ausschließlich für den Austausch konzipiert. Infolge der angeführten Gründe fiel die Wahl der 3D-Technologie auf X3D.

4.2 Einführung in X3D

X3D ist eine XML-basierte Modellierungssprache zur Beschreibung dreidimensionaler virtueller Umgebungen und gilt als offizieller Nachfolger der Virtual Reality Modeling Language (VRML) [Web3D-Konsortium, 1997]. Das Buch von Brutzman und Daly [2007] gibt einen sehr guten Überblick über VRML und X3D und beschreibt viele der hier angeführten Inhalte ausführlicher. Die Entwicklung von X3D wird von mehreren unabhängigen Arbeitsgruppen vorangetrieben. Diese stehen unter der Aufsicht des *Web3D-Konsortiums*[13]. Das primäre Ziel liegt darin, einen anwendungsunabhängigen Standard für 3D-Modelle bereit zu stellen.

[12] http://www.khronos.org/collada
[13] http://www.web3d.org

X3D offeriert eine Vielzahl an Funktionalitäten. Hierunter zählen zwei- und dreidimensionale geometrische Primitive, Animation, Benutzerinteraktion und -navigation, räumlicher Sound und Video, physische Simulationen, Echtzeitkommunikation, Netzwerkfähigkeit und Unterstützung von Computer Aided Design (CAD). Darüber hinaus können zur Erweiterung der Standardfunktionalität benutzerdefinierte Datentypen, sogenannte Prototypen (engl. *prototypes*), vereinbart werden. Über Skripte lassen sich Programmier- und Skriptsprachen einbinden. Die Funktionalitäten bilden die Basis für den Einsatz von X3D in Architektur, Ingenieurwesen, Marketing, Training und Simulationen, Multimedia, Bildung, wissenschaftlicher und medizinischer Visualisierung sowie der Informationsvisualisierung respektive der Softwarevisualisierung. Ausgehend von der Spezifikation geben die folgenden Abschnitte einen Überblick über den Aufbau von X3D und die unterstützten Dateiformate. Anhand eines Beispiels werden die allgemeine Struktur einer X3D-Datei und der Szenegraph näher erläutert. Im Anschluss werden das den Animationen und Interaktionen zu Grunde liegende Ereignismodell sowie die Architektur eines X3D-Browsers vorgestellt.

4.2.1 Spezifikation

Die Spezifikation von X3D besteht aus den folgenden drei Teilen:

- ISO/IEC 19775:200x - umfasst eine technikunabhängige Beschreibung aller funktionalen Bestandteile von X3D.

- ISO/IEC 19776:200x - enthält die Abbildung der abstrakten Struktur auf unterstützte Dateiformate.

- ISO/IEC 19777:200x - beschreibt die Sprachanbindungen für Java und ECMAScript (engl. *European Computer Manufacturers Association*, ECMA).

Seit 2004 ist die technikunabhängige Beschreibung von X3D durch die *International Organization for Standardization*[14] (ISO) und die *International Electrotechnical Commission*[15] (IEC) als internationaler Standard anerkannt. Im Jahr 2005 folgte die Standardisierung der Dateiformate und ein Jahr später die Sprachanbindungen. Die Standards werden regelmäßig durch Zusatz- beziehungsweise Abänderungsanträge (engl. *amendments*) aktualisiert. Zur Spezifikation des logischen Aufbaus von X3D-Dateien stehen DTDs und XML-Schemata in den entsprechenden Versionen zur Verfügung. Die weiteren Ausführungen zu X3D beziehen sich auf ISO/IEC FDIS 19775-1.2:2008 und ISO/IEC 19776:2005 [Web3D-Konsortium, 2008].

4.2.2 Profile, Komponenten und Knoten

Dem Aufbau von X3D liegt ein modulares Konzept zu Grunde. Die gesamte Spezifikation ist in Profile unterteilt, die sich in Komplexität und Funktionalität unterscheiden. Ein *Profil* setzt sich aus einer vordefinierten Menge von Komponenten zusammen. Laut Spezifikation sind insgesamt sieben Profile definiert. Abbildung 4-1 gibt einen Überblick über die grundlegenden Profile und deren Verhältnis zueinander.

[14] http://www.iso.org
[15] http://www.iec.org

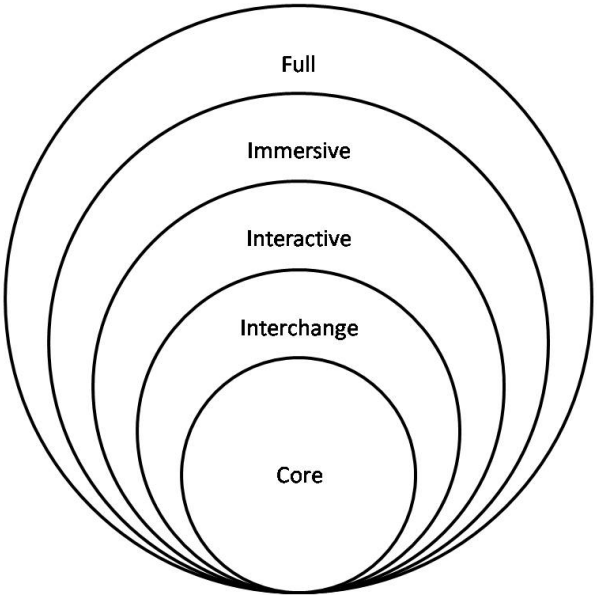


Abbildung 4-1: X3D-Profile (ähnlich [Brutzman und Daly, 2007, S. 13])

Core stellt Mittel bereit, um die minimal notwendigen Informationen einer X3D - Anwendung zu deklarieren. *Interchange* dient dem Austausch geometrischer Modelle zwischen unterschiedlichen Anwendungen. *Interactive* ermöglicht Interaktion und Navigation eines Benutzers mit bzw. in der dreidimensionalen Umgebung. *Immersive* entspricht der VRML - Spezifikation [Web3D-Konsortium, 1997] und *Full* bietet alle Komponenten, die in der X3D - Spezifikation [Web3D-Konsortium, 2008] vereinbart sind. Neben den genannten Profilen stehen noch *CADInterchange* und *MPEG-4 interactive* zur Verfügung. Beide Profile sind aber im Kontext dieser Arbeit nicht relevant.
Wie bereits erwähnt, setzen sich Profile aus Komponenten zusammen. Die Bausteine von Komponenten bilden Knoten (engl. *node*). Jede *Komponente* bietet mit den darin enthaltenen Knoten bestimmte Funktionalitäten. *Knoten* werden durch Felder (engl. *field*) näher beschrieben und können weitere Knoten enthalten. In Tabelle 4-1 ist ein Teil der Komponenten mit den dazugehörigen Knoten sowie einer kurzen Beschreibung aufgeführt.

Beschreibung	**Komponente**	**Knoten**
Unterstützung der Navigation und Orientierung des Benutzers in der Szene	Navigation	Viewpoint, NavigationInfo etc.
Kapselung von Kindknoten und Definition eines Koordinatensystems	Grouping	Anchor, Billboard, Collision, Group, LOD, Switch, Transform
Zweidimensionale geometrische Primitive	Geometry2D	Arc2D, Circle2D, Disk2D, Rectangle2D, TriangleSet2D etc.
Dreidimensionale geometrische Primitive	Geometry3D	Box, Cone, Cylinder, Sphere, Extrusion etc.

Beschreibung	Komponente	Knoten
Definition des Erscheinungsbildes geometrischer Primitive	Shape	Appearance, Material, Shape etc.
Einbindung von Texturen	Texturing	ImageTexture, MovieTexture etc.
Einbindung von Text	Text	FontStyle, Text
Zeitgesteuertes Auslösen von Ereignissen	Time	TimeSensor
Benutzergesteuertes Auslösen von Ereignissen	Pointing device sensor	CylinderSensor, PlaneSensor, SphereSensor, TouchSensor
Interpolation bei Animationen	Interpolation	ColorInterpolator, CoordinateInterpolator, OrientationInterpolator etc.
Komposition und Verlinkung verteilter Szenen	Networking	Anchor, Inline, LoadSensor
Einbindung von Sound	Sound	AudioClip, Sound
Einbindung von ECMAScript und Java	Scripting	Script
Gestaltung der Umgebung	Environmental effects	Background, Fog etc.
Ausleuchtung der Szene	Lighting	DirectionalLight, PointLight, SpotLight

Tabelle 4-1: Ausgewählte X3D-Komponenten

Die getroffene Auswahl an Komponenten und Knoten deckt bei weitem nicht alle spezifizierten Elemente ab. Sie dient lediglich einem groben Überblick. Eine detailliertere Betrachtung von Knoten erfolgt im Abschnitt 5.2.2 und eine Beschreibung der im Prototyp verwendeten Knoten im Abschnitt 6.2.1.

4.2.3 Dateiformate

Es existieren drei verschiedene Dateiformate, die zur Formatierung einer X3D-Datei benutzt werden können. Das erste Format verwendet XML und besitzt die Endung `.x3d`. Das zweite Format baut auf der klassischen VRML-Syntax auf und trägt die Endung `.x3dv`. Im Gegensatz zu diesen beiden Klartext-Formaten ist das dritte Format ein Binär-Format mit der Endung `.x3db`. In den verschiedenen Formaten wird dieselbe Szene in einem X3D-Browser konsistent dargestellt. Autorenwerkzeuge dienen neben der Erstellung von X3D-Dateien auch zur Konvertierung zwischen den verschiedenen Formaten. Eine aktuelle Liste der Autorenwerkzeuge ist der Supportseite[16] des *Web3D-Konsortiums* zu entnehmen.

4.2.4 Struktur

Die Struktur einer X3D-Datei ähnelt der einer XML-Datei. Szenegraphen, Knoten und Felder in X3D entsprechen Dokumenten, Elementen und Attributen in XML [Brutzman und Daly, 2007,

[16] http://www.web3d.org/x3d/content/examples/X3dResources.html

S. 21]. Anhand von Listing 4-1 wird die allgemeine Struktur einer XML-basierten X3D-Datei näher erläutert.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <X3D profile='Immersive' version='3.2' xmlns:xsd='http://www.w3.org
       /2001/XMLSchema-instance' xsd:noNamespaceSchemaLocation='http://www.
       web3d.org/specifications/x3d-3.2.xsd'>
3    <head>
4      <meta content='HelloWorld.x3d' name='title'/>
5      <meta content='Richard Mueller' name='author'/>
6      <meta content='21.07.2008' name='created'/>
7    </head>
8    <Scene>
9    <!--Szenegraph mit 3D-Objekten-->
28   </Scene>
29 </X3D>
```

Listing 4-1: Allgemeine Struktur einer X3D-Datei

In Zeile 1 erfolgt mit der XML-Deklaration die Auszeichnung der Datei als XML-Datei. Der Wurzelknoten wird in Zeile 2 definiert. Über die Felder des Wurzelknotens werden notwendige Informationen zur Version, zum Profil und zum XML-Schema bereit gestellt. Die Zeilen 3 bis 7 definieren den Header. Im Header können einerseits Metainformationen wie zum Beispiel Titel, Autor, Erstellungsdatum und andererseits zusätzlich zu den durch das Profil definierten Komponenten weitere Komponenten angegeben werden. Zeile 8 leitet den Szenegraph ein. Dieser enthält die eigentlichen 3D-Objekte.

4.2.5 Szenegraph

Das Herzstück jeder 3D-Anwendung ist der Szenegraph (engl. *scene graph*). Ein Szenegraph ist ein gerichteter, azyklischer Graph respektive ein Baum [Brutzman und Daly, 2007, S. 1], dessen Knoten den Objekten einer Szene entsprechen. Besitzen die Objekte eine oder mehrere Positionen in der virtuellen Welt, so lassen sie sich der Transformationshierarchie (engl. *transformation hierarchy*) zuordnen. Die Transformationshierarchie beschreibt die räumlichen Beziehungen der Objekte zueinander. Stehen die Objekte in Verbindung mit Ereignissen, so lassen sie sich dem Ereignismodell zuordnen. Bevor mit dem Beispiel fortgefahren wird, sei an dieser Stelle noch auf zwei allgemeine Konventionen bezüglich Koordinatensystem und Maßeinheiten hingewiesen. Jeder 3D-Szene liegt ein dreidimensionales, kartesisches, rechtshändiges Koordinatensystem zu Grunde. Rotationen erfolgen deshalb im mathematisch positiven Sinne, das heißt, gegen den Uhrzeigersinn. Die Maßeinheiten für Distanz-, Grad-, Zeit- und Farbangaben sind der Tabelle 4-2 zu entnehmen.

Kategorie	**Maßeinheit**
Lineare Distanz	Meter
Grad	Radiant
Zeit	Sekunden
Farbe	RGB ([0.,1.], [0.,1.], [0.,1.])

Tabelle 4-2: Maßeinheiten in X3D

Listing 4-2 erweitert Listing 4-1 und zeigt die Definition eines einfachen Szenegraphen im Detail.

```
8   <Scene>
9     <Viewpoint description='Hello, World!' position='0 0 10'/>
10    <Transform translation='0 0 0' rotation='0 1 0 2.5'>
11      <Shape>
12        <Sphere/>
13          <Appearance>
14            <ImageTexture url='earth-topo.png'/>
15          </Appearance>
16      </Shape>
17    </Transform>
18    <Transform translation='0 2 0'>
19      <Shape>
20        <Text string='Hello, World!'>
21          <FontStyle justify='"MIDDLE" "MIDDLE"'/>
22        </Text>
23        <Appearance>
24          <Material diffuseColor='0 0 1'/>
25        </Appearance>
26      </Shape>
27    </Transform>
28  </Scene>
29 </X3D>
```

Listing 4-2: Einfacher Szenegraph in X3D (ähnlich [Brutzman und Daly, 2007, S. 33])

Die initiale Kameraposition der Szene wird in Zeile 9 gesetzt. In den Zeilen 10 bis 17 wird eine Kugel mit einer Textur erzeugt. Die Kugel ist einem Gruppenknoten zugeordnet. Dieser bestimmt die Position des Objektes mit X=0, Y=0 und Z=0 und beschreibt eine Rotation um die Y-Achse um 2,5 Rad. Die Zeilen 19 bis 27 vereinbaren eine vertikal und horizontal zentriert ausgerichtete, blaue Zeichenkette an der Position X=0, Y=2 und Z=0. Abbildung 4-2 zeigt eine Baumdarstellung und Abbildung 4-3 gibt einen visuellen Eindruck der hier beschriebenen Szene.

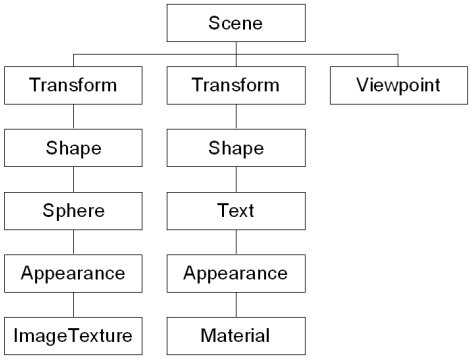


Abbildung 4-2: Szenegraph als Baumdarstellung

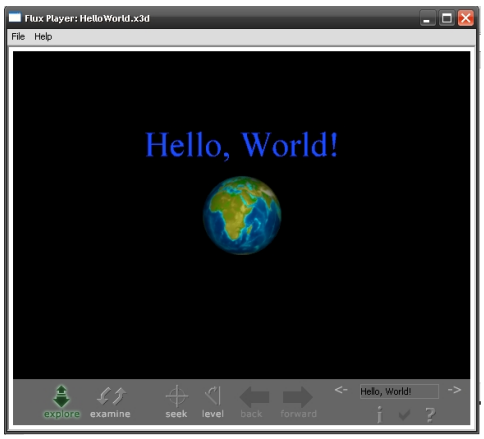


Abbildung 4-3: Beispielanwendung: Hello, World! mit X3D (ähnlich [Brutzman und Daly, 2007, S. 33])

4.2.6 Ereignismodell

Animationen und Interaktionen ermöglichen die Modifikation von Objekten einer Szene zur Laufzeit und folgen im Allgemeinen einem bestimmten Muster, das in Abbildung 4-4 dargestellt ist. Nach dem Auslösen eines Ereignisses (engl. *event*) durch einen Sensor oder einen Trigger wird in der Regel ein Zeitsensor (engl. *time sensor*) aktiviert, der Werte an einen Interpolator sendet. Die von dem Interpolator erzeugten Werte werden daraufhin zur Modifikation eines Zielknotens (engl. *target node*) verwendet. In jedem Schritt der Animation erfolgt die Weiterleitung der Ereignisse von einem zum nächsten Knoten über Pfade (engl. *routes*). Dieses Konzept wird als *Ereignismodell* (engl. *event model*) bezeichnet [Brutzman und Daly, 2007, S. 188ff].

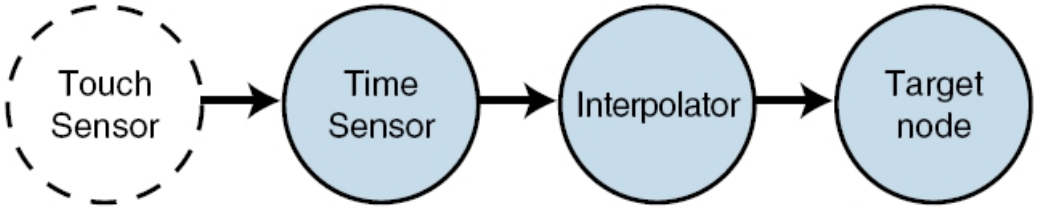


Abbildung 4-4: Ereignismodell von X3D [Brutzman und Daly, 2007, S. 189]

4.2.7 X3D-Browser

Die Interpretation, Ausführung und Darstellung einer Szene realisiert ein X3D-Browser. Dieser liest oder schreibt den Inhalt einer Szene. Nach dem Einlesen interpretiert ein Parser das entsprechende Format. Es werden Knoten erzeugt und an den Szenegraph-Manager weiter geleitet, der diese mit entsprechender Geometrie, Erscheinungsbild, Position und Orientierung zeichnet. Des weiteren kann der Szenegraph-Manager Ereignisse von Animations- oder Skriptknoten empfangen, die die gerenderte Szene manipulieren. Über eine Schnittstelle, das Scene

Access Interface (SAI), ist es möglich, zur Laufzeit auf die Objekte einer Szene zuzugreifen. Abbildung 4-5 verdeutlicht die Architektur eines X3D-Browsers.

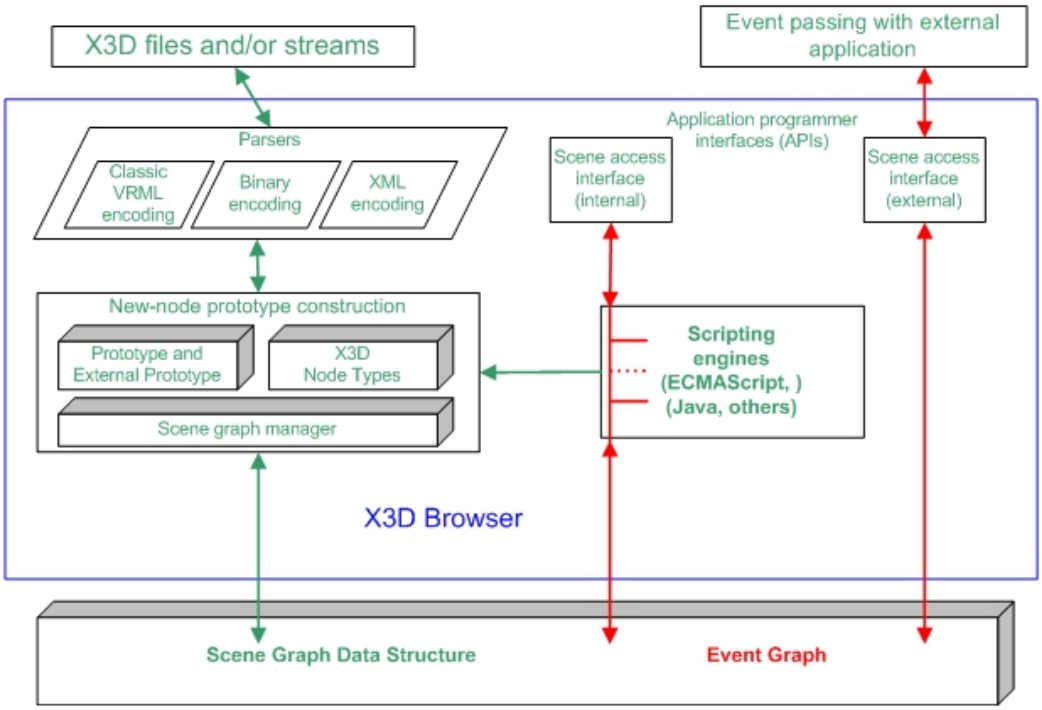


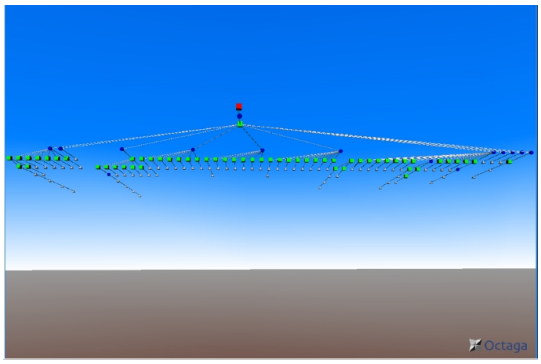
Abbildung 4-5: Architektur eines X3D-Browsers [Web3D-Konsortium, 2008]

Es gibt verschiedene Implementierungsmöglichkeiten für einen X3D-Browser. Er kann als Plugin in einen Web-Browser, wie zum Beispiel Internet Explorer oder Mozilla Firefox, eingebettet oder als eigenständige Anwendung umgesetzt sein. Die zur Verfügung stehenden X3D-Browser sind ebenfalls auf der bereits erwähnten Supportseite des *Web3D-Konsortiums* zu finden.

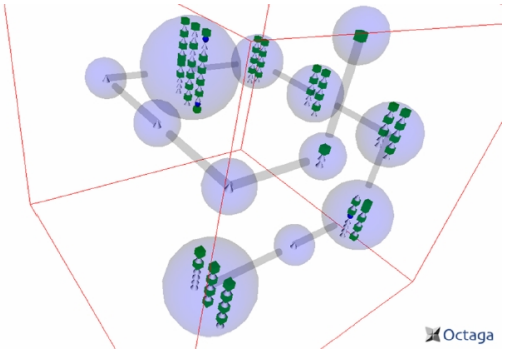
4.3 Ansätze zur Softwarevisualisierung

Die Verwendung von VRML beziehungsweise ihrem Nachfolger X3D im Rahmen der dreidimensionalen Softwarevisualisierung ist nicht neu. Vielmehr gibt es mehrere Ansätze, insbesondere zur Visualisierung der Struktur und des Verhaltens von Software mit diesen Technologien. Ein Ausschnitt dieser Ansätze wird nachfolgend dargestellt.
Anslow et al. [2008] nutzen X3D unter anderen zur Visualisierung von Aufrufgraphen. Sie haben weiterhin gezeigt, dass sich mit X3D auch dynamische Visualisierungen, beispielsweise in Form animierter Algorithmen, umsetzen lassen [Anslow et al., 2007]. Churcher et al. [1999] verwenden VRML in Verbindung mit XML zur Softwarevisualisierung. Sie visualisieren Vererbungsstrukturen, Klassenhierarchien, Klassenkohäsion und objektorientierte Metriken. Charters et al. [2002] setzen VRML und XML ein, um Software mit der Stadt-Metapher besser verständlich zu machen. Feijs und Jong [1998] verwenden VRML zur Visualisierung der Softwarearchitektur und benutzen dafür verschiedenfarbige Legosteine, wobei deren Beziehungen

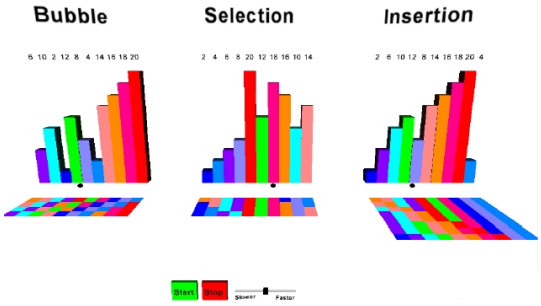
durch Pfeile gekennzeichnet sind. Auch im Umfeld von UML gibt es einige Bestrebungen, die dritte Dimension effektiv nutzbar zu machen. Gogolla et al. [1999] nimmt hierfür VRML. Mcintosh et al. [2005] untersuchen, inwieweit sich X3D zur Darstellung von UML-Diagrammen eignet. Sie haben gezeigt, dass sich Klassendiagramme in den dreidimensionalen Raum projizieren lassen. Darauf aufbauend beschreiben sie in [Mcintosh et al., 2008] wie Zustandsdiagramme dreidimensional visualisiert werden können. In Abbildung 4-6 (a) bis (h)sind einige Beispiele der hier beschriebenen Ansätze angeführt.



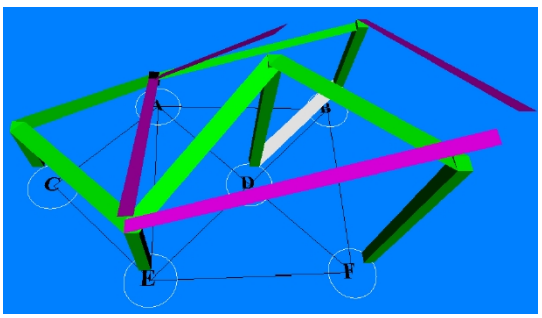
(a) Aufrufgraph in einer Software-Landschaft mit X3D [Anslow et al., 2007]



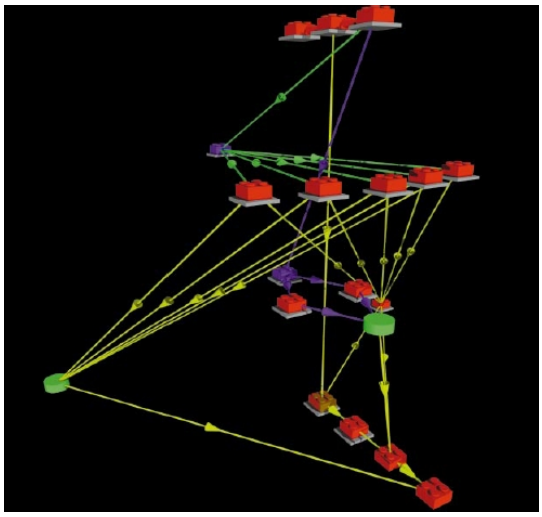
(b) Aufrufgraph als Information Cube mit X3D [Anslow et al., 2007]



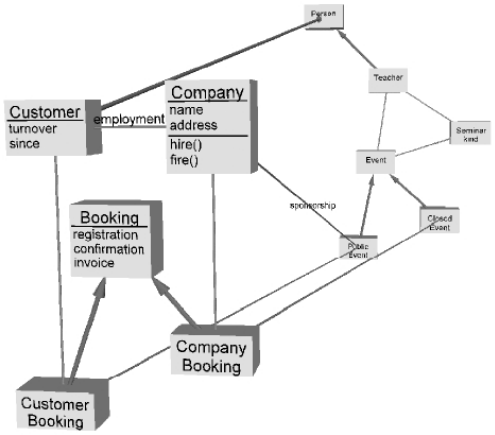
(c) Animierte Sortieralgorithmen mit X3D [Anslow et al., 2008]



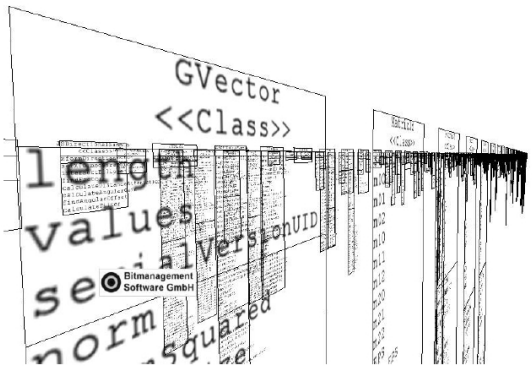
(d) Animierter Dijkstra-Algorithmus mit X3D [Anslow et al., 2007]



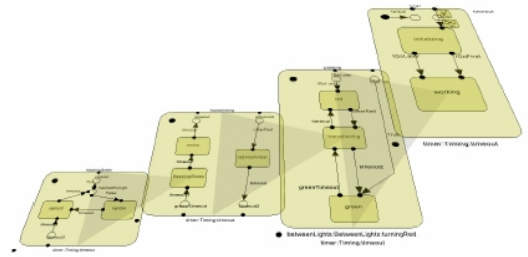
(e) Systemarchitektur mit Legosteinen mit VRML [Feijs und Jong, 1998]



(f) UML-Klassendiagramm mit VRML [Gogolla et al., 1999]



(g) UML-Klassendiagramm mit X3D [Mcintosh et al., 2005]



(h) UML-Zustandsdiagramm mit X3D [Mcintosh et al., 2008]

Abbildung 4-6: Ansätze zur dreidimensionalen Softwarevisualisierung mit X3D und VRML

5 Konzept zur generativen und modellgetriebenen Visualisierung

Ziel ist es, den Visualisierungsprozess zur Erzeugung von 3D-Modellen zu automatisieren. Um dies zu erreichen, wird in diesem Kapitel aufbauend auf den vorhergehenden Kapiteln ein allgemeines Konzept für einen Generator entwickelt. Die generative Softwareentwicklung ist darauf spezialisiert, eine Menge ähnlicher Softwaresysteme ausgehend von einer Spezifikation vollautomatisch zu erzeugen und bildet deswegen den Ausgangspunkt. Die Idee Visualisierungen auf der Basis von Modellen mittels Modelltransformationen zu erstellen, entspringt dem Ansatz der modellgetriebenen Visualisierung. In den folgenden Abschnitten werden ein generatives Domänenmodell zur dreidimensionalen Softwarevisualisierung entworfen und Techniken zur Umsetzung vorgeschlagen. Das Kapitel mündet in die Vorstellung einer Methode zur Entwicklung solcher Generatoren.

5.1 Adaption des generativen Domänenmodells

In Anlehnung an die Ein-Satz-Definition [Czarnecki und Eisenecker, 2000, S. 5] der generativen Softwareentwicklung, lässt sich das Ziel wie folgt formulieren: Der Visualisierungsprozess soll so gestaltet werden, dass ausgehend von einer Anforderungsspezifikation mittels Konfigurationswissen aus elementaren, wiederverwendbaren Implementierungskomponenten einer Modellfamilie in Form eines Szenegraphen ein hochangepasstes und optimiertes 3D-Modell nach Bedarf automatisch erzeugt werden kann. Im Unterschied zur ursprünglichen Definition handelt es sich bei dem Ergebnis nicht um ein System sondern um ein Modell, das die Struktur, das Verhalten oder die Evolution eines Softwaresystems visualisiert. Anstelle der manuellen Montage eines einzelnen Modells, können alle Modelle auf der Basis eines gemeinsamen generativen Domänenmodells automatisch erzeugt werden. Abbildung 5-1 veranschaulicht, wie sich dieses Ziel mithilfe der Terminologie des generativen Paradigmas entsprechend der Struktur des generativen Domänenmodells beschreiben lässt.

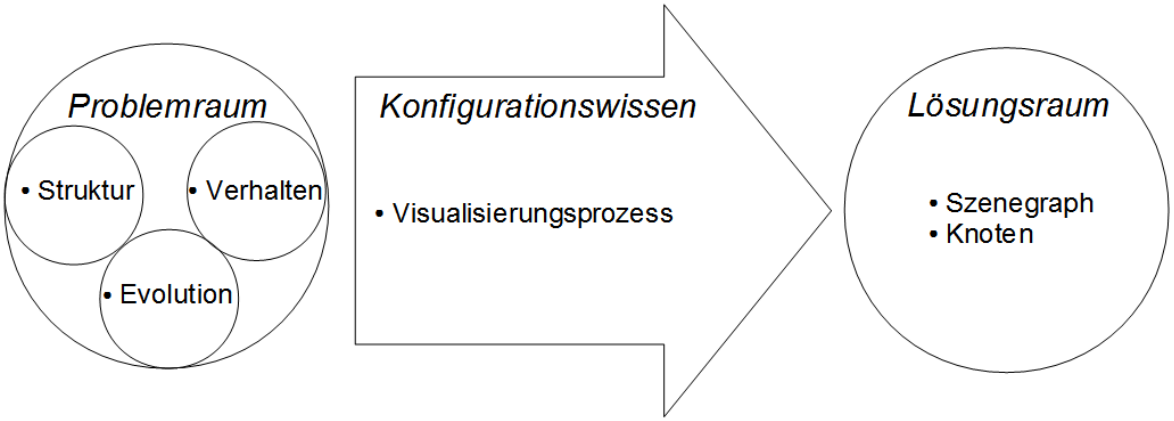


Abbildung 5-1: Generatives Domänenmodell zur Softwarevisualisierung in 3D

Der Problemraum enthält Mittel zur Spezifikation von Mitgliedern einer Systemfamilie. Hier wird festgelegt was auf welche Art und Weise visualisiert werden soll. Eine zentrale Rolle spielen dabei die Artefakte, die Informationen über ein Softwaresystem enthalten. Es lassen sich entsprechend den Kategorien von Artefakten die Domänen Struktur, Verhalten und Evolution identifizieren. Sie enthalten statische, dynamische oder historisierte Informationen über ein Softwaresystem. In Verbindung mit einer DSL lassen sich zum einen die Informationen selektieren, die visualisiert werden sollen. Zum anderen kann mit ihr bestimmt werden, wie die Informationen auf graphische Repräsentationen abgebildet werden. Die DSL entspricht der Anforderungsspezifikation und steuert das Filtering und das Mapping des Visualisierungsprozesses. Damit können Benutzer über die DSL Einfluss auf die Visualisierung nehmen, was im Kontext der Softwarevisualisierung dem *computational* oder *visual steering* entspricht.

Der Lösungsraum umfasst die Implementierungskomponenten einer Systemfamilie mit all ihren Kombinationsmöglichkeiten. Die gemeinsame Grundlage aller 3D-Modelle bilden Szenegraphen. Wie bereits im Abschnitt 4.2.5 beschrieben, handelt es sich dabei aus graphentheoretischer Sicht um einen Baum. Der Wurzelknoten enthält die Gesamtszene. Der Wurzel untergeordnet sind Kindknoten, die einzelne Objekte der Szene enthalten. Das Konzept des Szenegraphen entspricht folglich der Systemfamilie und die einzelnen Knoten den Implementierungskomponenten.

Das Konfigurationswissen bildet die Elemente des Problemraumes auf die Elemente des Lösungsraumes ab. Die Abbildung erfolgt in der Regel unter Verwendung eines Generators. Der Generierungsprozess ist in diesem Sinne auch gleichzeitig ein Visualisierungsprozess. Das bedeutet einerseits, dass die Visualisierung vollautomatisch abläuft und andererseits, dass bei der Generierung die einzelnen Prozessschritte gemäß der Visualisierungspipeline beachtet werden müssen. Zu dem Wissen über unzulässige Merkmalkombinationen, Standardvorgaben, Abhängigkeiten, Optimierungen und Bauanleitungen kommt demzufolge das Wissen über Analyse, Filterung und Mapping hinzu. Das Rendern fällt nicht in den Aufgabenbereich des Generators.

5.2 Technikprojektion

Damit dieses theoretische Modell in der Praxis umgesetzt werden kann, müssen konkrete Techniken für die Bestandteile des generativen Domänenmodells identifiziert werden. Müller [2008] hat einen Vergleich zwischen der generativen und modellgetriebenen Softwareentwicklung und eine Technikprojektion des Werkzeuges openArchitectureWare vollzogen. Bei der Technikprojektion [Müller, 2008, S. 44ff] hat der Autor die drei Bestandteile des generativen Domänenmodells auf die Technologien des Werkzeuges abgebildet. Die vorliegende Arbeit baut auf diesen Ergebnissen auf. Die Ergebnisse müssen dabei an die spezifische Problemstellung angepasst werden. Abbildung 5-2 fasst die Technikprojektion zusammen.

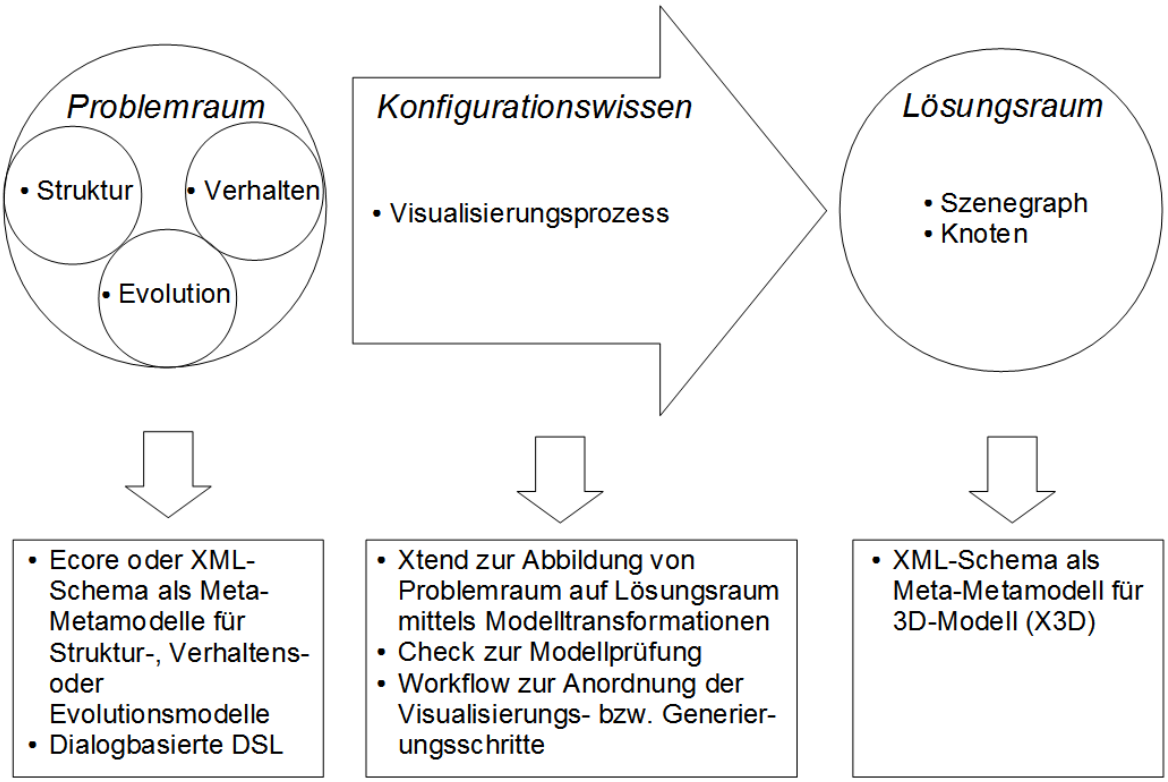


Abbildung 5-2: Technikprojektion des generativen Domänenmodells zur Softwarevisualisierung in 3D (ähnlich [Müller, 2008, S. 45])

Sowohl der Problemraum als auch der Lösungsraum werden jeweils mit Modellen beschrieben. Im Problemraum können dafür Ecore-basierte oder XML-basierte Modelle mit Informationen über Struktur, Verhalten oder Evolution über ein Softwaresystem verwendet werden. Mit einer dialogbasierten DSL werden die zur Visualisierung relevanten Informationen aus diesen Modellen selektiert. Zusätzlich dient die DSL dazu, das Mapping der Informationen auf die Implementierungskomponenten des zu erzeugenden 3D-Modells festzulegen. Das 3D-Modell gehört zum Lösungsraum. Der zu Grunde liegende Szenegraph wird durch X3D realisiert, dessen Metamodell als XML-Schema vorliegt. Das Konfigurationswissen verbindet beide Räume miteinander, indem es mittels Modelltransformationen die Modelle aufeinander abbildet. Dafür wird die Sprache Xtend eingesetzt. Weiterhin dient die Sprache Check für Modellprüfungen und die *Workflow*-Beschreibungen zur Definition der durchzuführenden Generierungs- beziehungsweise Visualisierungsschritte. In den folgenden drei Abschnitten werden die Techniken näher erläutert.

5.2.1 Problemraum

Den Ausgangspunkt für den Visualisierungsprozess stellen Artefakte über Struktur, Verhalten und Evolution dar, denn sie enthalten statische, dynamische und historisierte Informationen über ein Softwaresystem. Damit sie automatisch transformiert werden können, müssen die Informationen strukturiert in Form eines Modells vorliegen, dessen Syntax und Semantik durch

ein Metamodell definiert sind. An diesem Punkt stellt sich die Frage, wie man an ein solches Metamodell kommt. Hierfür gibt es zwei Alternativen: Entweder ist ein Metamodell bereits vorhanden oder es muss in der Domänenentwicklung selbst erstellt werden. Im Rahmen dieser Arbeit wird lediglich auf den ersten Fall näher eingegangen. Es sei in diesem Zusammenhang aber noch erwähnt, dass sich sowohl Ecore als auch XML zur manuellen Metamodellierung eignen.
In Abschnitt 3.2.1 wurde Ecore bereits vorgestellt. Es enthält Strukturinformationen über Pakete, Klassen, Methoden und Attribute sowie deren Beziehungen untereinander. Neben den in Abschnitt 3.2.2 beschrieben Alternativen, um an ein Core-Modell zu gelangen, sei an dieser Stelle noch auf eine weitere Möglichkeit verwiesen. Für den Fall, dass keine Modelle für das Softwaresystem zur Verfügung stehen, existiert ein Plugin[17] für Eclipse, mit dem Java-Quelltext in Core-Modelle überführt werden können.
In engem Zusammenhang mit dem Modell steht eine DSL. Sie wird benötigt, um zu spezifizieren, welche Modellelemente auf welche Weise visualisiert werden sollen.

5.2.2 Lösungsraum

Im Abschnitt 4.2.2 wurde der allgemeine Zusammenhang zwischen Profilen, Komponenten und Knoten dargestellt. An dieser Stelle werden die elementaren Bestandteile, also die Knoten, aus einem konzeptionellen Blickwinkel mit dem Ziel betrachtet, X3D als Systemfamilie zu charakterisieren. Darüber hinaus werden die Komponenten, die zur Konstruktion eines X3D-Modells dienen, näher beschrieben.

5.2.2.1 Systemfamilienarchitektur

Das Metamodell respektive die Architekturbeschreibung von X3D manifestiert sich in einer DTD[18] bzw. einem XML-Schema[19]. Der Entwurf des Metamodells orientiert sich an Konzepten aus dem objektorientierten Paradigma [Brutzman und Daly, 2007, S. 20]. Es lassen sich demnach ein einheitliches Typsystem, Vererbung, Datenkapselung sowie Objekte mit Struktur und Verhalten identifizieren.
Die Typhierarchie von X3D enthält abstrakte Schnittstellen und konkrete Knotentypen. Die abstrakten Schnittstellen definieren die Funktionalität, die von anderen Schnittstellen oder Knotentypen geerbt werden kann. Die Instanzierung der Schnittstellen als Objekte in einem Szenegraph ist nicht möglich. Im Gegensatz dazu werden konkrete Knotentypen von einer oder mehreren Schnittstellen abgeleitet und können instanziert werden. Die beiden abstrakten Schnittstellen, von denen fast alle Objekte erben, sind `X3DNode` und `X3DField`. Jedes Objekt besitzt einen Typnamen, wie zum Beispiel `SFBool` oder `Transform` und eine Implementierung, die seine Funktionalität realisiert.
Objekte, die von `X3DNode` abgeleitet sind, werden als Knoten bezeichnet. Knoten können mit einem Namen versehen werden, Felder aggregieren und Ereignisse senden beziehungsweise empfangen. Die Benennung eines Knotens erfolgt über die `DEF`-Direktive. Zur Referenzierung des Knotens an anderer Stelle wird die `USE`-Direktive verwendet.
Felder sind von `X3DField` abgeleitete Objekte. Bei Feldtypen mit dem Präfix `SF` (engl. *single field*) handelt es sich um einzelne Werte, wogegen Felder mit dem Präfix `MF` (engl. *multiple*

[17] http://code.google.com/p/java2ecore
[18] http://www.web3d.org/specifications/x3d-3.2.dtd
[19] http://www.web3d.org/specifications/x3d-3.2.xsd

field) für ein Array des Typs stehen. Abgesehen von `SFNode` und `MFNode` sind alle Feldtypen elementar. Diese Ausnahme ermöglicht die Schachtelung von Knoten. Felder beschreiben die Struktur von Knoten und enthalten Werte, die über Ereignisse gesendet oder empfangen werden. X3D kennt vier verschiedene Zugriffsarten auf Felder:

- `initializeOnly` - Bei der Erzeugung eines Knotens wird dem Feld einmalig ein Wert zugewiesen, wobei dieser zu jedem Zeitpunkt ausgelesen werden kann. Ereignisse spielen hierbei keine Rolle.
- `inputOnly` - Der Knoten empfängt Ereignisse, die den Wert des Feldes ändern. Das Auslesen des Wertes ist nicht möglich.
- `outputOnly` - Der Knoten sendet Ereignisse, wenn sich der Wert des Feldes ändert. Das Schreiben des Wertes ist nicht möglich.
- `inputOutput` - Der Knoten kann Ereignisse senden und empfangen und der Wert des Feldes kann zu jedem Zeitpunkt ausgelesen werden.

Bei den Feldern vom Typ `initializeOnly` und `inputOutput` müssen Standardwerte bereitgestellt sein.
Ein Ereignis besteht aus einem mit einem Zeitstempel versehenen Wert, der über eine `ROUTE`-Direktive zwischen den Feldern von unterschiedlichen Knoten übertragen wird. Ereignisse steuern also das Verhalten von Knoten. Die hier beschriebenen Merkmale sind für alle X3D-Modelle eine zwingende Voraussetzung und bilden damit die gemeinsame Systemfamilienarchitektur.

5.2.2.2 Komponenten

Neben den in der Architektur von X3D integrierten (engl. *built-in*) Knotentypen, die als elementare Komponenten verstanden werden können, gibt es noch zwei weitere Arten von Wiederverwendungseinheiten: Prototypen und komplette X3D-Dateien.
Die Spezifikation bietet einen Erweiterungsmechanismus zur Erstellung benutzerdefinierter Knotentypen, sogenannter Prototypen. Anhand eines Beispiels, entlehnt aus dem Anhang der Spezifikation [Web3D-Konsortium, 2008], wird nachfolgend die Definition und die Instanzierung eines Prototyps beschrieben. Im Beispiel geht es darum, einen Tisch aus einem Quader (engl. *box*) und vier Zylindern (engl. *cylinder*) zu definieren, wobei die Tischplatte und -beine jeweils mit unterschiedlichen Farben parametrisiert werden können.
Ein Prototyp setzt sich aus einer Deklaration, einer Schnittstelle und einer Implementierung zusammen. Wie Listing 5-1 zu entnehmen ist, werden zur Definition der Bestandteile die Direktiven `ProtoDeclare`, `ProtoInterface` und `ProtoBody` verwendet.

```
7  <ProtoDeclare name='TwoColorTable'>
8    <ProtoInterface>
9      <field name='legColor' type='SFColor' value='0 0 1'
10          accessType='initializeOnly'/>
11     <field name='topColor' type='SFColor' value='0 0 1'
12          accessType='initializeOnly'/>
13   </ProtoInterface>
14   <ProtoBody>
15     <Transform>
```

```
16      <Transform translation='0.0 0.6 0.0'>
17        <Shape>
18          <Appearance>
19            <Material DEF='TableTopMaterial'>
20              <IS>
21                <connect nodeField='diffuseColor' protoField='topColor'/>
22              </IS>
23            </Material>
24          </Appearance>
25          <Box size='1.2 0.2 1.2'/>
26        </Shape>
27      </Transform>
28      <Transform translation='-0.5 0.0 -0.5'>
29        <Shape DEF='Leg'>
30         <Appearance>
31           <Material DEF='LegMaterial' diffuseColor='1.0 0.0 0.0'>
32             <IS>
33               <connect nodeField='diffuseColor' protoField='legColor'/>
34             </IS>
35           </Material>
36         </Appearance>
37         <Cylinder height='1.0' radius='0.1'/>
38        </Shape>
39      </Transform>
40      <Transform translation='0.5 0.0 -0.5'>
41        <Shape USE='Leg'/>
42      </Transform>
43      <Transform translation='-0.5 0.0 0.5'>
44        <Shape USE='Leg'/>
45      </Transform>
46      <Transform translation='0.5 0.0 0.5'>
47        <Shape USE='Leg'/>
48      </Transform>
49    </Transform>
50  </ProtoBody>
51 </ProtoDeclare>
```

Listing 5-1: Deklaration, Schnittstelle und Implementierung eines Prototyps (ähnlich [Web3D-Konsortium, 2008])

Die Deklaration des Prototyps erfolgt in Zeile 7. In den Zeilen 8 bis 13 wird dessen Schnittstelle festgelegt. Sie besteht aus zwei Feldern, die die Farbwerte der Tischplatte beziehungsweise der Tischbeine enthalten. Zusätzlich wird der Standardwert blau für diese Felder festgelegt. In den nächsten Zeilen folgt die Implementierung. In diesem Fall kommen die integrierten Knotentypen `Box` für die Tischplatte und `Cylinder` für die Tischbeine zum Einsatz. Es wäre aber auch möglich, zur Montage des Prototyps auf andere, bereits definierte Prototypen zurückzugreifen. Zu Beginn wird die Tischplatte erstellt. Eine zentrale Rolle spielt hierbei die `IS`-Direktive in den Zeilen 20 bis 24 und 32 bis 34. Sie verbindet die öffentlichen Felder aus der Schnittstellendefinition mit den Feldern der im Prototyp gekapselten Knoten. Das erste Tischbein wird in den Zeilen 29 bis 38 erstellt und unter Verwendung der `DEF`-Direktive mit einem Bezeichner versehen. Die anderen drei Tischbeine referenzieren mit der `USE`-Direktive das erste Tischbein. Sie werden lediglich an anderer Stelle positioniert.
Listing 5-2 stellt die Instanzierung eines Prototyps dar.

```
52 <ProtoInstance name='TwoColorTable'>
53   <fieldValue name='legColor' value='1 0 0'/>
54   <fieldValue name='topColor' value='0 1 0'/>
55 </ProtoInstance>
```

Listing 5-2: Instanzierung eines Prototyps (ähnlich [Web3D-Konsortium, 2008])

Mit der Direktive `ProtoInstance` wird eine Instanz des Prototyps beziehungsweise des zweifarbigen Tischs erzeugt, wobei die Felder mit den Farbwerten grün für die Tischplatte und rot für die Tischbeine parametrisiert werden. Damit wird der Standardwert blau überschrieben. Wenn der Prototyp in einer externen X3D-Datei definiert wurde, muss dieser vor seiner Instanzierung über die `ExternProtoDeclare`-Direktive eingebunden werden. Der daraus resultierende Tisch ist in Abbildung 5-3 dargestellt.

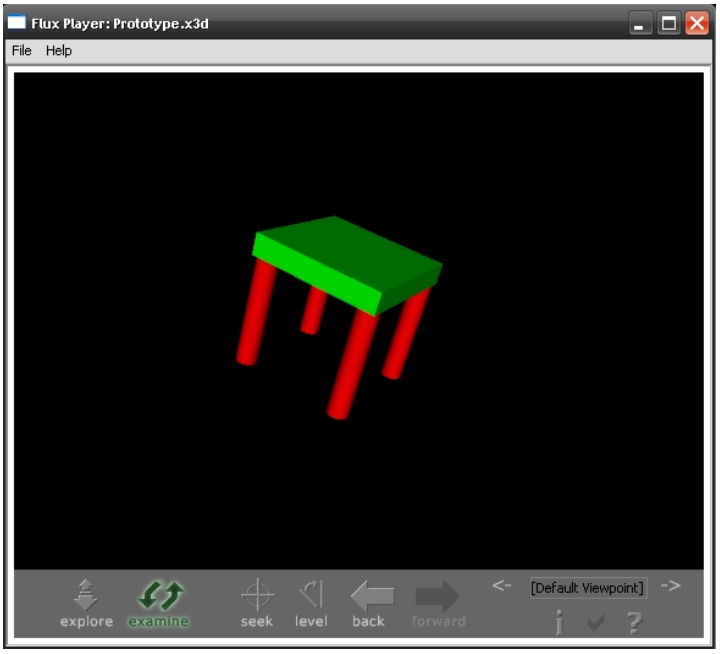


Abbildung 5-3: Zweifarbiger Tisch als Prototyp

Über `Inline`-Knoten lassen sich Inhalte einer externen X3D-Datei in den aktuellen Szenegraphen einbinden. In Verbindung mit `IMPORT`- und `EXPORT`-Direktiven können Ereignisse an den `Inline`-Knoten gesendet und von dem Knoten empfangen werden [Brutzman und Daly, 2007, S. 382f].

Im Sinne des generativen Paradigmas lassen sich demnach drei Komponenten identifizieren, die sich jeweils in ihrer Komplexität unterscheiden. Die kleinste Wiederverwendungseinheit bilden die integrierten Knotentypen. Prototypen lassen sich unter Verwendung dieser elementaren Komponenten oder anderer Prototypen erstellen, und schließlich ist es auch möglich, komplette X3D-Dateien wieder zu verwenden. Durch die Kombination dieser Komponenten lässt sich die Variabilität realisieren.

5.2.3 Konfigurationswissen

Als Techniken für das Konfigurationswissen wurden die Sprachen Xtend, Check und die *Workflow*-Beschreibung von oAW gewählt. Die Abbildung 5-4 konkretisiert den bisher abstrakt beschriebenen Generierungsprozess und zeigt die Verbindungen zur Visualisierungspipeline.

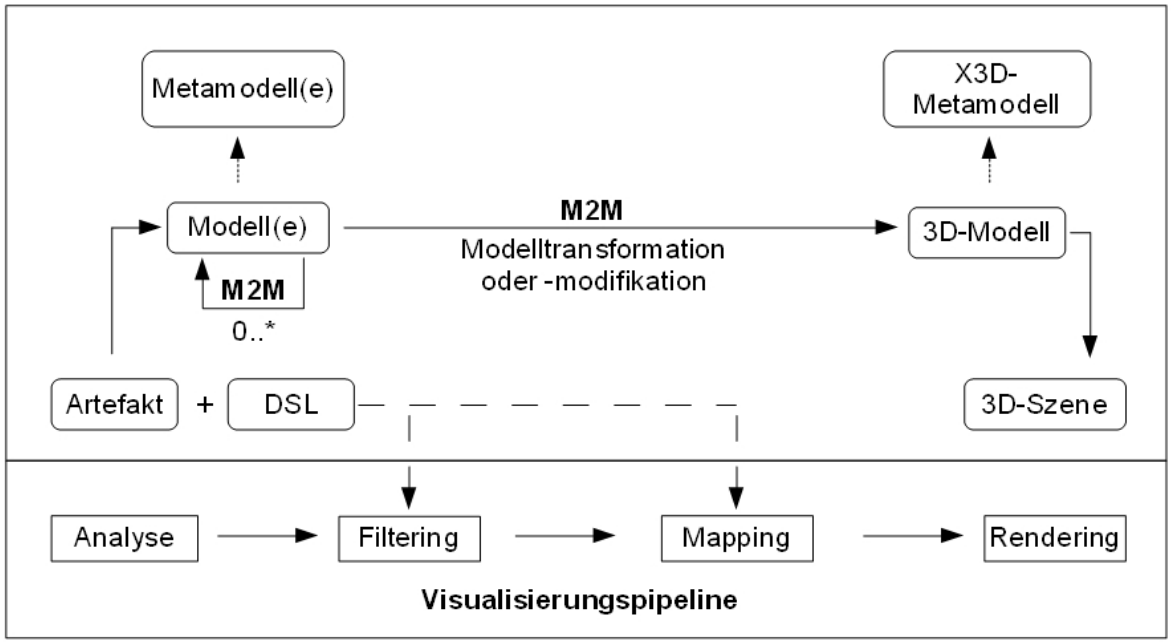


Abbildung 5-4: Visualisierungsprozess des Generators

Am Anfang der Prozesskette stehen Artefakte. Die darin enthaltenen Informationen werden auf einem höheren Abstraktionsniveau in Modellen repräsentiert. Mittels der Sprache Check ist es möglich, diese Modelle zu prüfen (*Analyse*). Die Benutzeranforderungen aus einer DSL werden dazu verwendet, um die zu visualisierenden Informationen zu selektieren (*Filtering*) und deren Abbildung auf graphische Repräsentationen zu steuern (*Mapping*). Die Abbildung wird mit mindestens einer in Xtend beschriebenen Modelltransformation realisiert. Eine einmal verfasste Transformation kann für alle Instanzen dieses Modells wiederverwendet werden. Die einzelnen Schritte des Visualisierungsprozesses lassen sich über die *Workflow*-Beschreibung anordnen und steuern. Am Ende der Prozesskette steht ein 3D-Modell, das mit einem entsprechenden Browser gerendert (*Rendering*) und betrachtet werden kann.

XSLT vs. Xtend

Bei einigen der im Abschnitt 4.3 angeführten Ansätze zur Softwarevisualisierung mit X3D fällt auf, dass XSLT zur Erzeugung der dreidimensionalen Visualisierungen verwendet wird [Anslow et al., 2007, 2008; Churcher et al., 1999; Mcintosh et al., 2005, 2008]. Der Einsatz von XSLT wird nachfolgend diskutiert.
XSL Transformation (XSLT) ist neben XPath und XSL Formatting Objects (XSL-FO) ein Bestandteil der XML Stylesheet Language[20] (XSL). XSLT ist speziell auf die Transformation

[20] http://www.w3.org/Style/XSL

von XML-Bäumen in XML-Bäume oder Text ausgelegt. XSLT-Templates oder auch XSLT-Stylesheets enthalten Transformationsregeln und sind dabei selbst nach den Regeln des XML-Standards aufgebaut. Über die Abfragesprache XPath wird die zu verarbeitende Knotenmenge eines Quell-Baumes selektiert, die dann gemäß den vorgegebenen Regeln mittels eines XSLT-Prozessors in ein Ziel-Baum überführt wird. Auf den ersten Blick scheint sich XSLT in Verbindung mit Xpath für die Verarbeitung von Modellen anzubieten, da die Modelle meistens in XML serialisiert sind. Bei genauerer Betrachtung werden jedoch auch die Nachteile der Verwendung von XSLT im Gegensatz zu oAW respektive der gemeinsamen Ausdruckssprache einschließlich Xtend und Check deutlich.
XML[21] ist ein Sprachstandard, mit dem Daten hierarchisch strukturiert und maschinenlesbar aufbereitet werden. Im Fokus bei der Entwicklung von XML stand die Maschinenlesbarkeit. Den XSLT-Stylesheets liegt aber die für Menschen schwer leserliche XML-Syntax zugrunde. Bei etwas umfangreicheren Transformationen werden die Stylesheets schnell unübersichtlich und schwer verständlich. In Bezug auf die Entwicklung, Wiederverwendung und Wartung von Stylesheets stellt dies einen erheblichen Nachteil dar.
Außerdem lassen sich mit Xtend die Modelltransformationen durch wesentlich weniger Quelltextzeilen (engl. *lines of code*, LOC) beschreiben. Dies liegt nicht zuletzt daran, dass die Ausgabe von XML bei oAW durch eine spezielle Ablauf-Komponente übernommen wird. Diese akzeptiert eine mit Xtend beschriebene Modelltransformation und erzeugt daraus valides und wohlgeformtes XML.
Im Rahmen von Modelltransformationen stehen Abfragen über Informationen aus Modellen im Mittelpunkt. Bei XSLT steht dafür XPath zur Verfügung. Diese Abfragesprache ist aber im Gegensatz zur Ausdruckssprache von oAW nicht statisch typisiert [Stahl et al., 2007, S. 149]. Dies hat zur Folge, dass der XSLT-Editor das Metamodell oder auch das XML-Schema nicht kennt. Aus diesem Grund werden potentielle Fehler erst bei der Ausführung der Transformation bemerkt. Weiterhin ist dadurch keine Möglichkeit der Autovervollständigung während der Entwicklung der Transformation gegeben.
Aus Gründen der besseren Lesbarkeit und Verbalisierung, Kompaktheit und der Typisierung wird der Einsatz von oAW mit Xtend und Check dem Einsatz von XSLT und XPath vorgezogen. Es sei an dieser Stelle angemerkt, dass dies nicht bedeutet, dass XSLT nicht zum Einsatz kommen kann. Ganz im Gegenteil, mit oAW ist es möglich, eigene Ablauf-Komponenten zu definieren, die eine XSL-Transformation enthalten.

5.3 Adaption des Prozessmodells

In ähnlicher Weise, wie das generative Domänenmodell zur Softwarevisualisierung adaptiert wurde, wird als nächstes das Prozessmodell angepasst. Auch hier bedarf es lediglich marginaler Veränderungen. Der Prozess der Domänenentwicklung entspricht im Wesentlichen demselben Verständnis wie im ursprünglichen Prozessmodell, das im Abschnitt 2.2.1 vorgestellt wurde. Dieser produziert wiederverwendbare Bestandteile in Form von DSLs, Generatoren einschließlich der Systemfamilie mit den Implementierungskomponenten. Der einzige Unterschied besteht darin, dass der Systemfamilie das Konzept eines Szenegraphen zu Grunde liegt. Wie bereits erläutert, handelt es sich bei dem Ergebnis nicht um ein System sondern ein 3D-Modell. Aus diesem Grund eignet sich als Bezeichnung für den zweiten Prozess eher der Begriff Visualisierungsprozess anstatt Anwendungsentwicklung. Bei diesem wird ausgehend von einer Anfor-

[21] http://www.w3.org/XML

derungsspezifikation unter Einsatz der wiederverwendbaren Einheiten vollautomatisch ein 3D-Modell erzeugt. Werden bestimmte Anforderungen durch die aktuelle Implementierung nicht unterstützt, so können diese durch die Rückkopplung in einer weiteren Iteration der Domänenentwicklung ihre Berücksichtigung finden. Abbildung 5-5 zeigt das resultierende Prozessmodell.

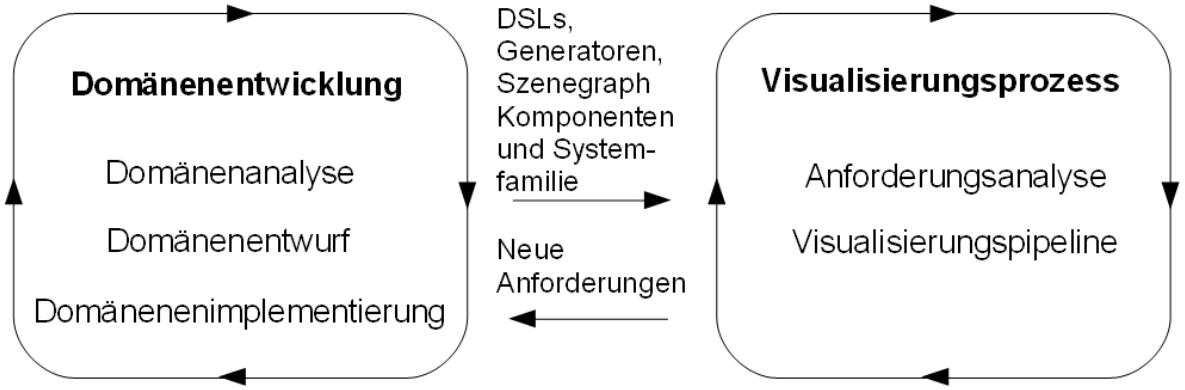


Abbildung 5-5: Prozessmodell zur Softwarevisualisierung in 3D

6 Prototyp

Unter Anwendung des im vorangegangenen Kapitel 5 entwickelten Konzepts wurde ein Generator prototypisch implementiert. Dieser ist als Plugin für Eclipse realisiert, welches aus Ecore-basierten Strukturmodellen nach Benutzervorgaben ein 3D-Modell im X3D-Format erzeugt. Als Visualisierungstechnik wird auf eine abstrakte Metapher zurückgegriffen. X3D bietet keine direkte Unterstützung für die dreidimensionale Anordnung der Elemente der Metapher. Um diesem Problem zu entgegnen, wird ein Graphenmodell als Zwischenschritt eingeführt. Dadurch ist es möglich, die Anordnung der Elemente der Metapher über Layout-Algorithmen zu bestimmen.
In den ersten drei Abschnitten dieses Kapitels wird der Entwicklungsprozess des Prototyps und der Entscheidungsfindungsprozess transparent dargestellt. Die Vorgehensweise folgt den Schritten der Domänenentwicklung. Anhand eines Beispiels wird danach die Funktionsweise näher erläutert. Schließlich werden der Prototyp bewertet und bestehende Probleme aufgezeigt.

6.1 Domänenanalyse

Am Anfang der Domänenentwicklung steht die Analyse. Hier werden die relevanten Domänen identifiziert und ein- und abgegrenzt. Dies erfolgt hauptsächlich durch die verwendeten Modelle respektive deren Metamodell. Für den Prototyp lassen sich die Domänen Struktur und Graph bestimmen. Im Folgenden wird dargestellt, durch welche Modelle diese Domänen im Prototyp repräsentiert werden.

6.1.1 Domäne Struktur

In der Analyse der Domäne Struktur geht es darum, festzulegen was im Rahmen des Prototyps unter der Struktur von Softwaresystemen zu verstehen ist. Eine erste Einschränkung erfährt der Strukturbegriff durch die Verwendung Ecore-basierter Modelle. Damit werden die Grenzen des Prototyps deutlich: Nur die Informationen, die in einem Ecore-basierten Modell abgebildet sind, können auch Gegenstand des Strukturbegriffs sein. Anders formuliert bedeutet das, dass die Grenzen der Domäne Struktur durch die Möglichkeiten und den Funktionsumfang von Ecore abgesteckt sind. Auf Ecore und seine Bestandteile sowie die Erstellung Ecore-basierter Modelle wurde bereits im Abschnitt 3.2 näher eingegangen. An dieser Stelle soll daher lediglich die Wahl von Ecore begründet und der Strukturbegriff geformt werden.
Ecore als Ausgangspunkt für den Visualisierungsprozess einzusetzen, bietet mehrere Vorteile. Zunächst ist festzustellen, dass es sich bei Ecore um einen etablierten Standard in Anlehnung an MOF handelt. Das Metamodell muss demnach nicht in aufwendiger Eigenarbeit selbst konzipiert werden, sondern es existiert bereits. Mit der Verwendung von Ecore wird von einer konkreten Programmiersprache abstrahiert. Es kann also theoretisch jedes Softwaresystem visualisiert werden, das in entsprechender Weise modelliert ist oder dessen Modell über eine Möglichkeit der in Abschnitt 3.2.2 aufgezeigten Methoden ableitbar ist. Ecore und Eclipse stehen in einem engen Zusammenhang zueinander. Eclipse enthält grafische und textbasierte

Editoren sowie Wizards, die bei der Erstellung und Wartung solcher Modelle wichtige Unterstützung bieten. Zudem ist oAW in der Lage, ohne große Vorarbeiten Ecore-basierte Modelle zu instanzieren und zu verarbeiten.
Durch die Festlegung der im Prototyp zu verwendenden Elemente von Ecore erfährt der Strukturbegriff eine weitere Konkretisierung. Zur Beschreibung der Struktur von Softwaresystemen werden die Entitäten Paket, Klasse, Methode, Attribut sowie die Beziehungen zwischen den Klassen verwendet, wobei bei den Beziehungen zwischen Vererbung und Assoziationen unterschieden wird. Ecore bietet zusätzlich statische Informationen über Parameter, Datentypen und Annotationen. Diese finden bisher aber keine Berücksichtigung.

6.1.2 Domäne Graph

Die Domäne Graph kann als Unterstützungsdomäne aufgefasst werden. Die Strukturelemente Paket, Klasse, Methode und Attribut sowie die Beziehungen zwischen den Klassen werden auf ein Graphenmodell abgebildet, um die Bestimmung ihrer Positionen und Größen im dreidimensionalen Raum an Werkzeuge zum Graphzeichnen zu delegieren. Nachfolgend werden potentielle Graphenformate und auch Werkzeuge zum Graphzeichnen auf ihre Eignung für den Einsatz im Prototyp untersucht. Der Entscheidungsprozess für ein Format und ein Werkzeug wird klar nachvollziehbar dargestellt.

6.1.2.1 Graphenformate

Ein geeignetes Format muss XML-basiert sein, damit es von oAW direkt verarbeitet werden kann. Weiterhin ist es notwendig, dass die Graphenelemente mit Attributen annotiert werden können. Auf diese Weise werden wichtige Informationen wie zum Beispiel der Name einer Klasse oder berechnete Positionen und Größen abgebildet. Zur Abbildung von Hierarchien soll eine geschachtelte Visualisierung zum Einsatz kommen. Dafür muss das Format hierarchische Graphen unterstützen. Zusammenfassend ergeben sich die folgenden Anforderungen an das Graphenformat:

- Format: XML-basiert
- Erweiterbarkeit: Unterstützung von Attributen
- Funktionalität: Unterstützung hierarchischer Graphen

GML und XGMML

Die *Graph Modeling Language*[22] (GML) erwuchs aus einer Idee für ein einheitliches Graphenformat auf dem *Symposium on Graph Drawing* im Jahr 1995. GML wurde an der Universität Passau entwickelt und ist ein textbasiertes Format zur Speicherung von Graphen. GML dient als Standardformat für *Graphlet* beziehungsweise dessen Nachfolger *Gravisto* und findet in zahlreichen anderen Werkzeugen zum Graphzeichnen Anwendung. Das Format enthält neben Mitteln zur Beschreibung der Struktur von Graphen Attribute, die die Strukturelemente näher beschreiben. So können beispielsweise Graphen, Knoten oder Kanten mit einer bestimmten Farbe, Form oder Größe versehen werden. Zusätzlich zu einfachen gerichteten und ungerichteten Graphen unterstützt GML auch hierarchische Graphen. Die *Extensible Graph Markup and*

[22] http://www.infosun.fim.uni-passau.de/Graphlet/GML/gml-tr.html

Modeling Language[23] (XGMML) bildet das XML-basierte Pendant zu GML und bietet daher dieselben Möglichkeiten wie das textbasierte Format. Sowohl GML als auch XGMML können mit einem Parser beziehungsweise einer XSL-Transformation ineinander überführt werden. Der Parser und das XSLT-Stylesheet sind über die Projektseite von XGMML beziehbar.

GraphML

Die *Graph Markup Language*[24] (GraphML) [Brandes et al., 2002] entstand auf dem *Symposium on Graph Drawing* im Jahr 2000 mit dem Ziel, ein einheitliches XML-basiertes Format für den Austausch von Graphen festzulegen. Über einen Erweiterungsmechanismus können die Graphen, Knoten und Kanten mit anwendungsspezifischen Daten versehen werden. GraphML unterstützt sowohl gerichtete und ungerichtete als auch gemischte Graphen. Neben hierarchischen Graphen ist es möglich, Hypergraphen abzubilden.

GXL

Die *Graph Exchange Language*[25] (GXL) [Holt et al., 2006] ist ein XML-basiertes Format und findet hauptsächlich im Kontext der Interoperabilität von Reengineering-Werkzeugen seine Anwendung. Die Strukturelemente Graphen, Knoten und Kanten können typisiert und attributiert werden. GXL unterstützt genauso wie GraphML gerichtete, ungerichtete, gemischte und verschachtelte Graphen sowie Hypergraphen. Allen Strukturelementen können Attribute zugeordnet werden, die zusätzliche Informationen wie zum Beispiel Koordinaten für ein Layout enthalten. GXL wurde als Standardaustauschformat für Werkzeuge im Reengineering auf dem *Dagstuhl Seminar on Interoperability of Reverse engineering tools*[26] festgelegt.

XWG

XML Wilma Graph[27] (XWG) [Dwyer, 2005] ist ein proprietäres XML-basiertes Format von WilmaScope. Es enthält neben den Strukturelementen eines Graphen zusätzliche Informationen über Layout und Darstellung und unterstützt hierarchische Graphen. Im Gegensatz zu den anderen Formaten ist XWG direkt auf die Visualisierung von Graphen ausgerichtet.

Von den angeführten Graphenformaten eignen sich auf den ersten Blick alle XML-basierten Formate, denn sie erfüllen die gestellten Anforderungen. Ein Format allein reicht jedoch noch nicht aus, denn es steht immer im Zusammenhang mit dem Werkzeug. Die Entscheidung für ein bestimmtes Graphenformat bedingt also noch die Untersuchung existierender Werkzeuge zum Graphzeichnen.

6.1.2.2 Werkzeuge zum Graphzeichnen

Die ursprüngliche Liste der begutachteten Werkzeuge war wesentlich länger. Die hier getroffene Auswahl enthält nur solche, die einer Open-Source-Lizenz unterliegen, also frei verfügbar sind, und auch dreidimensionale Layout-Algorithmen anbieten. Von zentraler Bedeutung bei

[23] http://www.cs.rpi.edu/~puninj/XGMML
[24] http://graphml.graphdrawing.org
[25] http://www.gupro.de/GXL
[26] http://www.dagstuhl.de/de/programm/kalender/semhp/?semnr=01041
[27] http://wilma.sourceforge.net

der Auswahl eines Werkzeuges ist das Portfolio der Layout-Algorithmen. Für die Visualisierung der Struktur von Software soll ein kraftgerichteter Algorithmus eingesetzt werden. Das Besondere dabei ist, dass dieser auf hierarchische Graphen angewendet werden können muss. Zudem spielt die Plattform auf der die Werkzeuge ausgeführt werden eine wichtige Rolle. Die Ausführung sollte unter Windows, Linux und Mac OS möglich sein. Im Hinblick auf den Im- und Export ist es notwendig, dass das Werkzeug ein XML-basiertes Graphenformat unterstützt. Für eine einfache Integration mit oAW wäre es von Vorteil, wenn es in der Programmiersprache Java implementiert ist. Dies stellt aber kein zwingendes Kriterium dar. Es muss lediglich eine Möglichkeit geben, auf die Funktionalität des Werkzeuges zuzugreifen. Dies kann auch über einen kommandozeilenbasierten Aufruf oder eine Schnittstelle geschehen. Die an das Werkzeug zum Graphzeichnen gestellten Anforderungen lassen sich wie folgt zusammenfassen:

- Layout: Kraftgerichtetes Layout für hierarchische Graphen
- Im- und Export: Unterstützung XML-basierter Graphenformate
- Plattform: Windows, Linux, Mac OS
- Sprache: Java (optional)

Die Untersuchungsergebnisse sind größtenteils das Resultat eigener Arbeit und werden nur an geeigneten Stellen durch die Arbeit von Kotzyba et al. [2008] ergänzt. Die Angaben zu den unterstützten Graphenformaten der einzelnen Werkzeuge sind nicht notwendigerweise komplett, denn es wurden nur die unter 6.1.2.1 vorgestellten Formate berücksichtigt.

Gravisto

Das *Graph Visualization Toolkit*[28] (Gravisto) ist ein interaktiver Grapheneditor. Gravisto wird an der Universität Passau entwickelt und ersetzt den Vorgänger *Graphlet*. Es gibt bisher noch keine veröffentlichte Version dieses Werkzeuges[29]. Gravisto bietet neben der Bearbeitung verschiedene Algorithmen zur Erzeugung und zur Visualisierung von Graphen. Die erweiterbare Architektur ermöglicht zusätzlich eine einfache Ausweitung der Funktionalität. Das präferierte Graphenformat dieses Werkzeuges ist GML.

Gluskap

Mit *Gluskap*[30] [Dyck et al., 2004b] können Graphen erzeugt, modifiziert und dreidimensional dargestellt werden. Ursprünglich wurde das an der Universität Lethbridge entwickelte Werkzeug in C++ implementiert, aber mit der Version 2.x erfolgte eine Portierung nach Python [Dyck et al., 2004a]. Die aktuelle Version ist 2.4. Das Angebot an Algorithmen zur Visualisierung ist im Vergleich zu anderen Werkzeugen relativ gering und es gibt keine direkten Erweiterungsmöglichkeiten. Dieses Werkzeug bietet Unterstützung für GML und GraphML.

[28] http://gravisto.fim.uni-passau.de

[29] Infolge einer Anfrage bei der Universität Passau wurde dem Autor dieser Arbeit der Zugang zur Beta-Version des Werkzeuges für Evaluationszwecke gewährt.

[30] http://www.cs.uleth.ca/~vpak/gluskap/download.html

WilmaScope

Die Entwicklung von *WilmaScope*[31] [Dwyer, 2005] begann im Jahr 2001 unter der Leitung von Tim Dwyer mit dem Ziel, dreidimensionale Repräsentationen von UML-Diagrammen zu erzeugen. Ein Jahr später wurde die ursprüngliche Anwendung mit dem Fokus auf interaktive und dreidimensionale Visualisierung von Graphen im Allgemeinen umgeschrieben. Das Werkzeug ist in Java implementiert und verwendet zur Bildgenerierung Java 3D. WilmaScope liegt eine modulare Architektur zugrunde. Geometrische Formen für Elemente von Graphen, Algorithmen zur Erzeugung, Modifikation und Analyse von Graphen werden dynamisch als Plugins geladen. Seit 2004 befindet sich das Werkzeug in der Version 3.1. Bei den Graphenformaten werden das proprietäre XWG und GML unterstützt. *Geometry for Maximum Insight*[32] (GEOMI) [Ahmed et al., 2006] kann als Weiterentwicklung von WilmaScope angesehen werden. Die Quellen von GEOMI ließen sich jedoch nicht fehlerfrei kompilieren, so dass lediglich WilmaScope untersucht werden konnte.

Tulip

Bei *Tulip*[33] [Auber, 2001] liegt der Fokus auf der Analyse und der Darstellung sehr großer Graphen. Laut der Projektseite des Werkzeuges können Graphen mit mehr als einer Million Elementen visualisiert werden. Tulip wurde an der Universität Bordeaux von David Auber entwickelt und liegt in der Version 3.0.2 vor. Es ist in C++ implementiert, wobei OpenGL zur dreidimensionalen Darstellung und Qt[34] für die Benutzungsoberfläche zum Einsatz kommen. Daneben gibt es auch eine kommandozeilenbasierte Variante von Tulip. Neben einem proprietären Format für Graphen unterstützt dieses Werkzeug GML. Das Funktionsangebot kann durch dynamische Laufzeitbibliotheken (engl. *dynamic link libraries*) erweitert werden.

VGJ

Das javabasierte Werkzeug *Visualizing Graphs with Java*[35] (VGJ) ist ein rudimentärer Grapheneditor und bietet einige wenige Algorithmen zur dreidimensionalen Visualisierung. Es akzeptiert Graphen in GML. Die Weiterentwicklung wurde im Jahr 1998 mit der Version 1.03 eingestellt.

GraphViz

Das von AT&T und den Bell Laboratories entwickelte Werkzeug *Graph Visualization*[36] (GraphViz) [Gansner und North, 2000], dessen Wurzeln bis Anfang der 90er Jahre zurück reichen, umfasst hauptsächlich Verfahren zur zweidimensionalen Visualisierung von Graphen. Die aktuelle Version ist die 2.20.3. Es gibt zwar einen einfachen Editor, aber im Grunde genommen ist GraphViz kommandozeilenbasiert. Mit der an die Programmiersprache C angelehnten Auszeichnungssprache *DOT* wird die visuelle Darstellung von Graphen gesteuert. Einerseits lässt sich mit ihr die Struktur eines Graphen beschreiben und andererseits können die Knoten und

[31] http://wilma.sourceforge.net
[32] http://www.cs.usyd.edu.au/~visual/valacon/geomi
[33] http://tulip.labri.fr
[34] http://trolltech.com
[35] http://www.eng.auburn.edu/department/cse/research/graph_drawing/graph_drawing.html
[36] http://www.graphviz.org

Kanten mit Attributen wie zum Beispiel Form oder Farbe versehen werden. Interpretiert wird *DOT* von den Renderern des Werkzeuges, die entsprechend ihres implementierten Algorithmus das Layout berechnen. Es gibt insgesamt fünf Renderer *dot*, *neato*, *fdp*, *twopi* und *circo*, die in C beziehungsweise C++ implementiert sind. Während alle anderen nur auf zweidimensionale Visualisierung ausgelegt sind, sind in *neato* dreidimensionale, kräftebasierte Verfahren implementiert. Mit dem Werkzeug können aber auch Graphen im GXL-Format geladen werden.

R

R[37] ist gleichzeitig eine Programmiersprache und eine Statistiksoftware. Das Projekt wurde 1992 von John Chambers und anderen an den Bell Laboratories entwickelt und liegt derzeit in der Version 2.7.2 vor. R offeriert eine Vielzahl an statistischen Verfahren und Methoden zur Erzeugung von Grafiken. Der Funktionsumfang kann durch Pakete erweitert und damit an spezifische Problemstellungen angepasst werden. Die Pakete *igraph* [Csardi, 2008] und *graph* [Gentleman et al., 2008] unterstützen die Graphenformate GML, GraphML sowie GXL und enthalten Methoden zur Analyse und Visualisierung von Graphen. Sowohl *igraph* als auch *graph* sind in C beziehungsweise C++ implementiert. Durch die Integration von R ergeben sich im Rahmen des Analyseprozesses interessante Möglichkeiten, denn in diesem Fall könnte auf alle zur Verfügung stehenden Verfahren zurückgegriffen werden.

Name	**Layout**	**Format**	**Plattform**	**Sprache**
Gravisto	nein	GML	Windows, Linux, Mac OS	Java
Gluskap	nein	GraphML, GML	Windows, Linux, Mac OS	Python
WilmaScope	ja	XWG, GML	Windows, Linux, Mac OS	Java
Tulip	ja	GML	(Windows), Linux, Mac OS	C++
VGJ	nein	GML	Windows, Linux, Mac OS	Java
GraphViz	ja	GXL	Windows, Linux, Mac OS	C/C++
R				
igraph	nein	GraphML, GML	Windows, Linux, Mac OS	C/C++
graph	nein	GXL	Windows, Linux, Mac OS	C/C++

Tabelle 6-1: Werkzeuge zur dreidimensionalen Visualisierung von Graphen

Tabelle 6-1 fasst die wichtigsten Informationen der beschriebenen Werkzeuge zusammen. Es wird deutlich, dass nur drei Werkzeuge in die engere Auswahl kommen, da alle anderen die

[37] http://www.r-project.org

Anforderung an das Layout nicht erfüllen. Um Tulip nutzen zu können, wird die kommandozeilenbasierte Variante dieses Werkzeuges benötigt. Hierfür müssen die Quellen von Tulip unter Windows kompiliert und erstellt werden, was dem Verfasser dieser Arbeit aufgrund eines nicht trivialen Bibliothekenproblems unmöglich war. Ein Eintrag[38] im Forum von Tulip blieb im Rahmen der Bearbeitungszeit dieser Arbeit unbeantwortet. Nach näherer Betrachtung von GraphViz stellte sich heraus, dass die Ergebnisse des Layout-Algorithmus vom Renderer *neato* unbefriedigend waren. Aus diesen Gründen fiel die Wahl letztendlich auf das Graphenformat XWG im Zusammenhang mit dem Werkzeug WilmaScope.

6.2 Domänenentwurf

Aus der Domänenanalyse ergibt sich das in Abbildung 6-1 skizzierte Domänenmodell für den Prototyp.

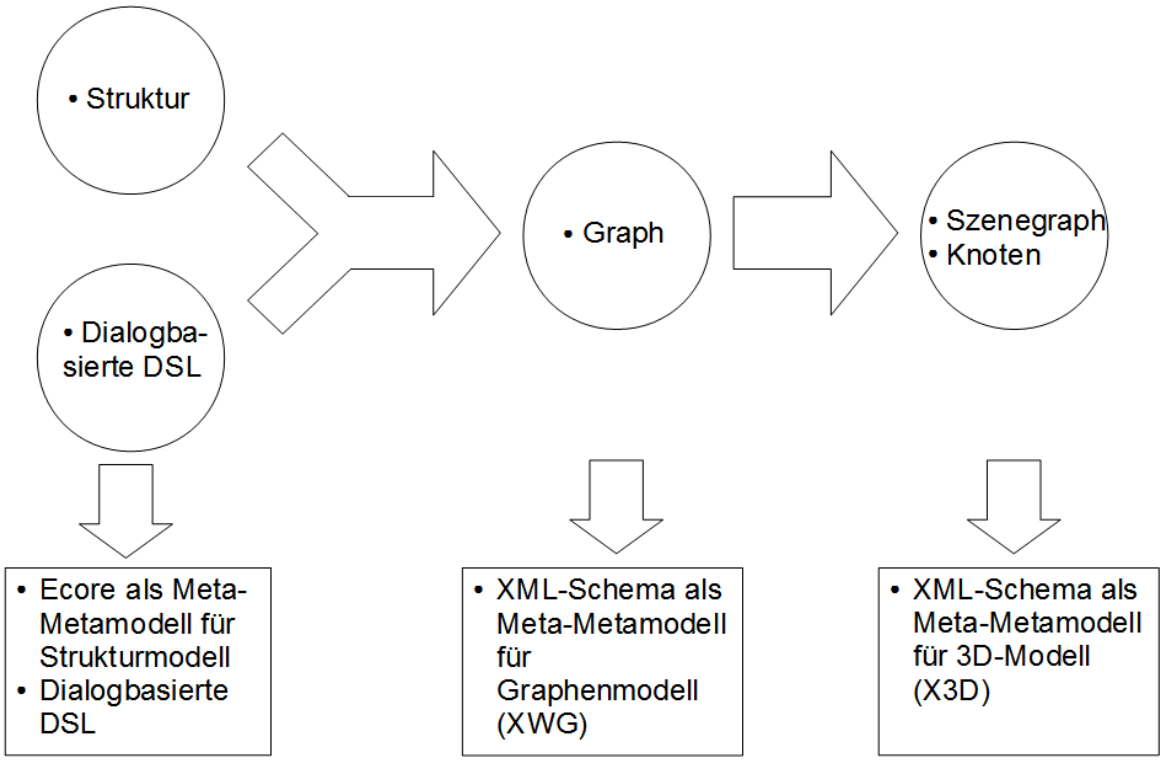


Abbildung 6-1: Generatives Domänenmodell des Prototyps einschließlich Techniken

Hierbei handelt es sich um eine Kombination zweier Abbildungsvarianten, die in Abschnitt 2.2.2 beschrieben sind. Zum einen setzt sich die Spezifikation aus dem Ecore-basierten Strukturmodell und den Benutzervorgaben über eine dialogbasierte DSL zusammen. Zum anderen kommt eine verkettete Abbildung zum Einsatz, da ein Graphenmodell basierend auf XWG zur Größen- und Positionsbestimmung der Elemente im dreidimensionalen Raum verwendet wird. Nachfolgend werden die Implementierungskomponenten des Szenegraphen und anschließend die Spezifizierung der DSL näher erläutert.

[38] http://sourceforge.net/forum/forum.php?thread_id=2114330&forum_id=206283

6.2.1 Implementierungskomponenten des Szenegraphen

Bei der Implementierung des Prototyps wurde auf integrierte Knotentypen zurückgegriffen. In diesem Abschnitt werden die verwendeten Knotentypen näher beschrieben. Im Anhang B ist deren genaue Spezifikation hinterlegt.

Gruppierung

Der `Transform`-Knoten aggregiert andere Knoten und definiert für diese ein eigenes Koordinatensystem. Die untergeordneten Knoten können verschoben (`transform`), rotiert (`rotate`) und skaliert (`scale`) werden. Der `Billboard`-Knoten dient dazu, seine untergeordneten Knoten in die Richtung des Betrachters auszurichten. Das spielt beispielsweise bei Texten eine wichtige Rolle, damit sie aus verschiedenen Perspektiven lesbar sind. Die Ausrichtung erfolgt aber nur in Bezug zur Z-Achse. Bei einer Rotation um die X- oder Y-Achse geht der Fokus verloren. Beide Knoten sind aus der Komponente *Grouping*.

Erscheinungsbild, Farbgebung und Beleuchtung

Der `Shape`-Knoten besteht im Wesentlichen aus den zwei Feldern `appearance` und `geometry`. Jedes Feld kann als Platzhalter für einen `Appearance`-Knoten oder einen `Geometry`-Knoten angesehen werden. Der `Appearance`-Knoten beschreibt das Erscheinungsbild von geometrischen Primitiven. Über das Feld `material` ist es möglich, einen `Material`-Knoten einzubinden. Darüber hinaus ist auch die Einbindung von Texturen über die Felder `texture` oder `textureTransform` möglich. Der `Material`-Knoten spezifiziert die Oberfläche von `Geometry`-Knoten gemäß dem *Phong-Beleuchtungsmodell* [Phong, 1975]. Demnach existieren zum Beispiel Felder, über die Farbe (`diffuseColor`), Beleuchtung (`ambientIntensity`, `shininess`) oder Transparenz (`transparency`) festgelegt werden. Alle drei Knoten sind in der Komponente *Shape* enthalten.

Geometrische Primitive

Zu den einfachen dreidimensionalen geometrischen Primitiven gehören die Knoten `Box`, `Sphere`, `Cone` und `Cylinder`. Mit ihnen lassen sich Quader, Kugel, Kegel und Zylinder darstellen. Alle Knoten besitzen das Feld `solid`, welches festlegt, ob die Rückseiten der Flächen gezeichnet werden. Entsprechend der Art des Körpers stehen Felder zur Verfügung, um dessen Größe zu definieren. Zum Beispiel werden die Breite, Höhe und Tiefe von einem Quader über das Feld `size` festgelegt.
Mithilfe des `Extrusion`-Knotens lässt sich eine Vielzahl an dreidimensionalen Körpern erzeugen. Im Prototyp wird dieser Knoten verwendet, um die Kanten zwischen Knoten im Sinne der Graphentheorie darzustellen. Hierzu wird ein zylinderähnlicher, schmaler Körper zwischen zwei Punkten im dreidimensionalen Raum erzeugt.
Sowohl `Extrusion` als auch die einfachen geometrischen Primitiven sind Bestandteil der Komponente *Geometry3D*.
Der `Text`-Knoten ermöglicht das Einbinden von Zeichenketten über das Feld `string`. Mit den Feldern des aggregierten `Fontstyle`-Knotens wird die Darstellung des Textes beeinflusst. Auf diese Weise können unter anderen Schriftart (`family`), Schriftstil (`style`), Schriftgröße (`size`), Position (`justify`) und Ausrichtung (`horizontal`) angepasst werden. Die Knoten `Text` und `Fontstyle` gehören zur Komponente *Text*.

Umgebung

Mit dem `Background`-Knoten werden über die Felder `groundColor` und `skyColor` die Farben für den Boden und den Himmel festgelegt. Damit steht für den Betrachter eine zusätzliche Orientierungshilfe zur Verfügung. Der Knoten ist in der Komponente *EnvironmentalEffects* enthalten.

6.2.2 Spezifizierung der DSL

Die Konfiguration des Visualisierungsprozesses erfolgt über eine dialogbasierte DSL. Die damit erstellte Konfiguration bildet zusammen mit dem Strukturmodell die Eingabe für den Generator. Abbildung 6-2 zeigt die Möglichkeiten, die der Benutzer spezifizieren kann.

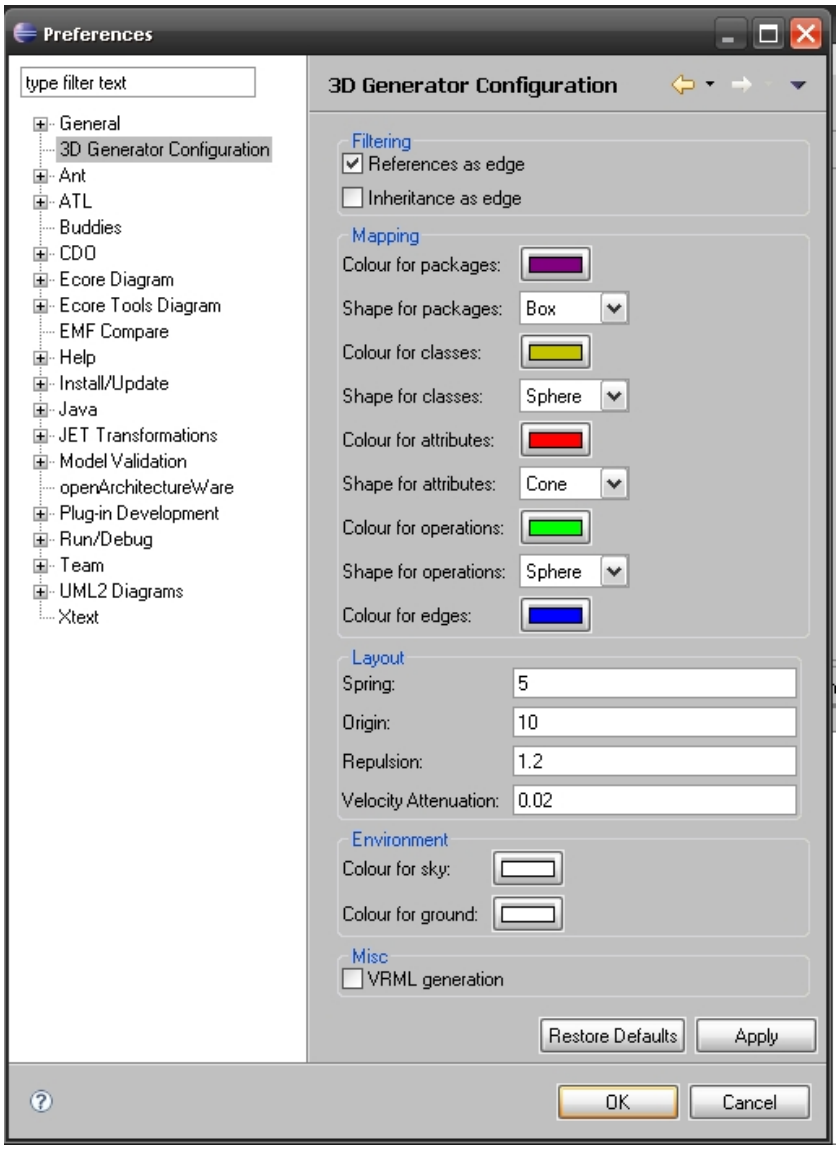


Abbildung 6-2: Dialog zur Konfiguration des Generators

Über den Filter wird bestimmt, welche Art von Beziehungen zwischen Klassen visualisiert werden. Die Alternativen umfassen Vererbungs- oder Assoziationsbeziehungen. Im Rahmen des Mappings können die geometrischen Primitive einschließlich ihrer Farbe für die Strukturelemente Paket, Klasse, Methode und Attribut ausgewählt werden. Zur Auswahl stehen hierfür Quader, Kugel, Kegel und Zylinder jeweils in Verbindung mit einem Farbwahldialog. Zudem kann die Farbe der Beziehungen festgelegt werden. Die Parameter für den Layout-Algorithmus müssen in einigen Fällen angepasst werden. So können Abstoßungs- und Anziehungskräfte nach Bedarf geändert werden. Zur besseren Orientierung in der virtuellen Welt besteht die Möglichkeit, die Farben für den Himmel und den Untergrund ebenfalls über einen Farbwahldialog anzugeben. Schließlich kann in diesem Dialog noch die Option zur zusätzlichen Generierung des 3D-Modells in VRML gesetzt werden. Für alle Einstellungen sind Standardvorgaben hinterlegt, die sich während des Entwicklungsprozesses des Prototyps als geeignet erwiesen haben.

6.3 Domänenimplementierung

Nachdem alle Domänen durch Modelle abgegrenzt und beschrieben sind, folgen die Ausführungen über deren Implementierung. In den beiden folgenden Abschnitten wird die Integration des Prototyps mit Eclipse sowie der Generator als Ganzes erklärt.

6.3.1 Integration mit Eclipse

Der Prototyp ist, wie Abbildung 6-3 zu entnehmen, in Eclipse eingebettet. Seine Bestandteile werden durch das Feature *X3D Generator Feature* aggregiert. Insgesamt gehören drei Plugins und zwei Fragmente zu dem Prototyp.
Der eigentliche Generator ist im Plugin *X3D Generator Core* enthalten. Hier befinden sich alle oAW-spezifischen Elemente wie die Metamodelle, Modellprüfungen, Modelltransformationen, die Konfigurationsdatei für die *workflow-engine* und die selbst definierten Ablauf-Komponenten. Dieses Plugin hängt von dem Plugin *X3D Generator Libraries* ab, welches die nötigen Laufzeitbibliotheken bereit stellt. Hierzu zählen WilmaScope für die Layout-Algorithmen sowie der XSLT-Prozessor Saxon[39] zur Umwandlung von X3D in VRML.
Die Integration des Generators mit Eclipse wird über das Plugin *X3D Generator Eclipse Integration* realisiert. Es dockt dafür an drei verschiedenen Erweiterungspunkten der Eclipse-Plattform an, die durch die Plugins *Eclipse Core Runtime* und *Eclipse UI* angeboten werden. Über den Erweiterungspunkt *Popup Menu* wird über einen Kontextmenüeintrag der Start des Generators initiiert und über *Preferences Page* kann der Visualisierungsprozess dialogbasiert konfiguriert werden. Zur persistenten Speicherung der Konfiguration dient dabei der Erweiterungspunkt *Preferences*.
Bei dem Aufruf der *workflow-engine* aus einem externen Plugin funktioniert die Konsolenausgabe aufgrund fehlender Abhängigkeiten zwischen *Apache Jakarta Log4j Plug-In* und *Apache Commons Logging Plug-In* nicht. Eine Möglichkeit diesem Problem entgegen zu treten, besteht in der Verwendung von Fragmenten und ist im oAW-Forum[40] beschrieben. Die Fragmente *Log4j Fragment* und *Apache Logging Fragment* sorgen also für die Informationsausgabe über die Konsole bei dem Generierungsprozess.

[39] http://saxon.sourceforge.net
[40] http://www.openarchitectureware.org/forum/viewtopic.php?showtopic=5258

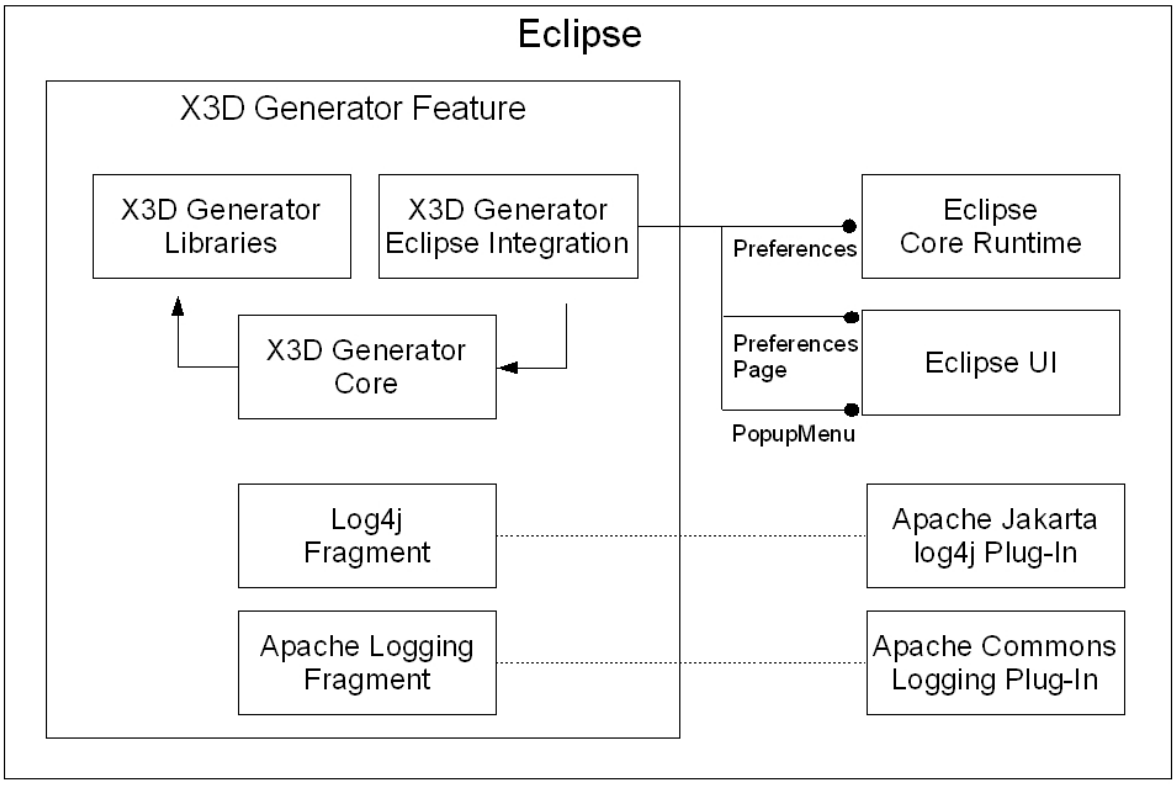


Abbildung 6-3: Einbettung des Prototyps in Eclipse

6.3.2 Implementierung des Generators

Den Ausgangspunkt des in Abbildung 6-4 skizzierten Visualisierungs- beziehungsweise Generierungsprozesses bildet ein Artefakt mit Informationen über die Struktur von Software. Für den Prototyp fiel die Wahl hierfür auf Ecore-basierte Modelle.

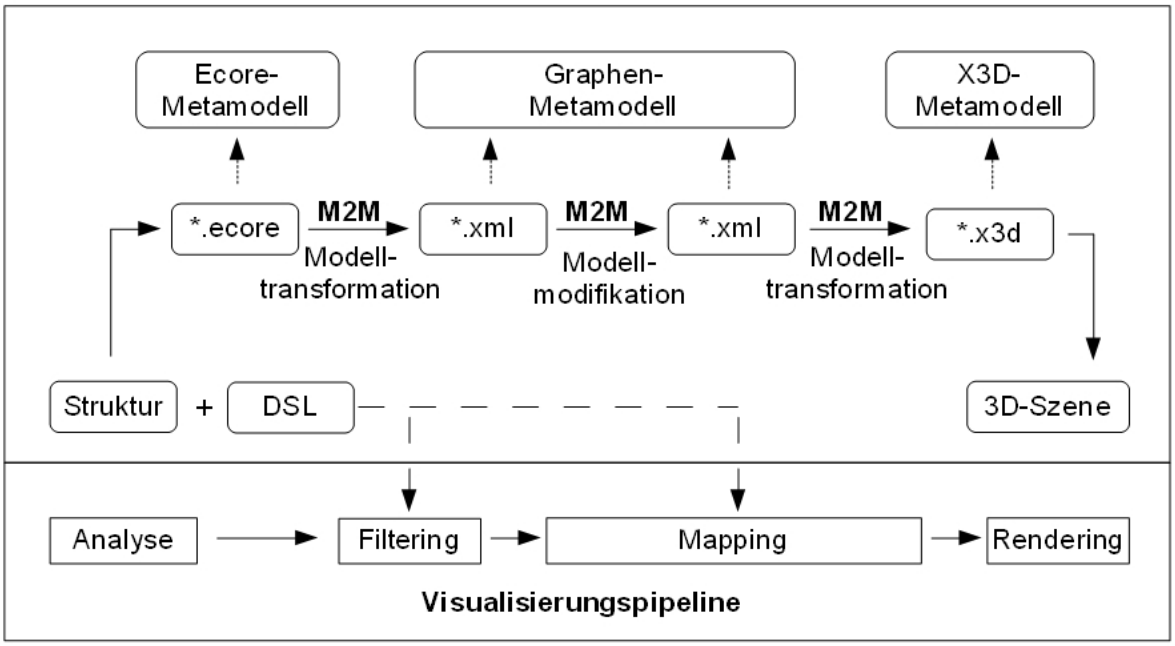


Abbildung 6-4: Visualisierungsprozess des Prototyps

Listing 6-1 zeigt die Schritte, die zur Initialisierung des Prozesses notwendig sind. Mit statischen Eigenschaften werden die Informationen über die einzelnen Metamodelle für X3D und XWG, das XSLT-Stylesheet zur Umwandlung von X3D in VRML sowie die Pfade zu den Xtend- und Check-Dateien bekannt gemacht. Die beiden Metamodelle liegen als XML-Schema vor. Da oAW für Ecore eine direkte Unterstützung bietet, muss es an dieser Stelle nicht explizit angegeben werden. Neben den statischen gibt es dynamische Eigenschaften wie zum Beispiel der Name des zu visualisierenden Modells. Diese werden zur Laufzeit durch das Plugin *X3D Generator Eclipse Integration* ermittelt und der *workflow-engine* übergeben.

```
1  <workflow>
2    <!-- Static properties -->
3    <property name="metamodel.x3d" value="x3d-3.2.xsd"/>
4    <property name="metamodel.xwg" value="WilmaGraph.xsd"/>
5    <property name="x3d2vrml.xslt" value="X3dToVrml97.xslt"/>
6    <property name="extensions.dir" value="
       org::x3d::generator::extensions::"/>
7    <property name="checks.dir" value="org::x3d::generator::checks::"/>
8
9    <!-- Set up workflow engine -->
10   <bean class="org.eclipse.mwe.emf.StandaloneSetup">
11     <platformUri value=".."/>
12   </bean>
13
14   <!-- Load model -->
15   <component class="org.eclipse.mwe.emf.Reader">
16     <useSingleGlobalResourceSet value="false"/>
17     <modelSlot value="ecoreModel"/>
```

```
18    <uri value="${model.uri}"/>
19  </component>
```

Listing 6-1: Initialisierung

Nachdem das Modell mit den Strukturinformationen geladen wurde, prüft der Generator das Modell im Rahmen der Analyse auf syntaktische und semantische Fehler. Im Fehlerfall bricht der Prozess an dieser Stelle ab. Warnungen hingegen werden auf der Konsole ausgegeben, führen aber nicht zum Abbruch. Die Listings 6-2 und 6-3 veranschaulichen den Aufruf der Komponente zur Modellprüfung und einige Regeln, die geprüft werden.

```
21  <!-- Check model -->
22  <component
23    class="org.openarchitectureware.check.CheckComponent">
24    <metaModel id="ecore" class="oaw.type.emf.EmfMetaModel"
25      metaModelPackage="org.eclipse.emf.ecore.EcorePackage" />
26    <checkFile value="${checks.dir}checks"/>
27    <emfAllChildrenSlot value="ecoreModel" />
28  </component>
```

Listing 6-2: Ablauf-Komponente zur Modellprüfung

Die Regeln sind beispielhaft implementiert und erfüllen keineswegs die Bedingung der Vollständigkeit. Sie entstanden während der Testphase des Prototyps. Dem iterativen Charakter des modellgetriebenen Ansatzes folgend, müssen diese noch ergänzt werden. In der aktuellen Implementierung wird neben der Länge des Namens von Modellelementen das Vorhandensein von referenzierten und erbenden Klassen geprüft.

```
1  import ecore;
2
3  context EClassifier WARNING
4  "The name " + name + " is too short." :
5  name.length > 2;
6
7  context EReference ERROR
8  "There is a reference from " + ((EClassifier)eContainer).name + " that
       has no target." :
9  ((EClass)eType).name != null;
10
11 context EClass ERROR
12 "The base class from " + name + " cannot be found." :
13 eSuperTypes.select(parent|parent.name == null).isEmpty;
```

Listing 6-3: Exemplarische Regeln zur Modellprüfung

Im Zuge der sich anschließenden Modell-zu-Modell-Transformationen wird das Ecore-basierte Strukturmodell auf ein XML-basiertes Graphenmodell mit den Elementen Graphen, Knoten und Kanten abgebildet. Die Entitäten Paket und Klasse werden in Graphen überführt. Hierbei bilden die Graphen von Klassen Subgraphen von Paketgraphen, um die Zugehörigkeit der Klasse zum Paket implizit kenntlich zu machen. Analog werden Methoden und Attribute auf Knoten im Graphen der angehörenden Klasse abgebildet. Die Beziehungen zwischen Klassen

werden durch Kanten repräsentiert. Für jede Beziehung wird an den Kantenenden, also innerhalb eines Klassengraphen, ein virtueller Knoten erzeugt. Welche Art von Beziehung im Fokus der Visualisierung stehen soll, kann über einen Filter gesteuert werden. Zur Charakterisierung der erzeugten Graphen, Knoten und Kanten werden diese auf der Basis von Schlüssel-Wert-Paaren mit Informationen über Bezeichner und Typ aus dem Strukturmodell angereichert. Für Knoten wird darüber hinaus eine eindeutige ID generiert. In Listing 6-4 ist der Aufruf dieser Modelltransformation dargestellt.

```
30  <!-- Load ecore and xwg metamodel and transform model to graph model
        -->
31  <component class="oaw.xtend.XtendComponent">
32    <metaModel idRef="ecore"/>
33    <metaModel id="xwg" class="org.openarchitectureware.xsd.XSDMetaModel
        ">
34      <schemaFile value="${metamodel.uri}${metamodel.xwg}" />
35    </metaModel>
36    <invoke value="${extensions.dir}ecore2xwg::ecore2xwg(ecoreModel)" />
37    <outputSlot value="graphModel" />
38  </component>
```

Listing 6-4: Ablauf-Komponente zur Modelltransformation: Ecore zu Graph

Ein Ausschnitt aus dieser Modelltransformation ist in Listing 6-5 zu sehen. In der Methode `toRootCluster()` wird ein Wurzelgraph erzeugt, der neben dem Graph für das Wurzelpaket alle Kanten enthält. Je nach dem, welche Art von Beziehung visualisiert werden soll, handelt es sich dabei um Vererbungs- oder Assoziationsbeziehungen. Die Abbildung von Paketen und Klassen auf Graphen erfolgt mit der überladenen Methode `toCluster()`. Jeder Graph wird dabei mit Informationen über Name und Art des Strukturelements sowie Parametern für den Layout-Algorithmus angereichert. Die Strukturinformationen werden aus dem geladenen Modell extrahiert. Die Parameter werden dynamisch über das Plugin *X3D Generator Eclipse Integration* aus den Einstellungen zur Konfiguration des Generators bezogen. Bei Paketen werden alle aggregierten Unterpakete und Klassen als Graphen hinzugefügt. Im Gegensatz dazu werden bei Klassen Attribute und Methoden als Knoten hinzugefügt. Die überladene Methode `toNode()` bildet Methoden und Attribute auf Knoten ab. Jeder Knoten enthält wie auch die Graphen Strukturinformationen. Darüber hinaus müssen Knoten eine eindeutige Kennung besitzen. Die Beziehungen zwischen Klassen werden mit der überladenen Methode `toEdge()` auf Kanten abgebildet. Jede Kante besitzt Informationen über die Art der Beziehung, also ob es sich um eine Vererbungs- oder Assoziationsbeziehung handelt und den Namen der Beziehung. Des Weiteren wird für jede Kante ein Start- und Endknoten festgelegt. Mit der Methode `getClone()` wird eine Kopie der Klasse erzeugt. Dies ist notwendig, da eine Klasse in mehreren Beziehungen stehen kann. Würde man in diesem Fall nicht mit einer Kopie arbeiten, wäre die Kennung nicht mehr eindeutig.

```
14 create Cluster toRootCluster(EPackage pkg):
15   setViewType(toClusterViewType("package", "root"))->
16   setLayoutEngineType(toForceDirectedLayout())->
17   cluster.add(pkg.toCluster())->
18   if (isFeatureSelected("refEdge"))
```

```
19    then edge.addAll(pkg.eAllContents.typeSelect(EReference).select(ref|
          ref.eType.name != null).toEdge())->
20  if (isFeatureSelected("inEdge"))
21    then edge.addAll(pkg.eAllContents.typeSelect(EClass).select(cls|!cls
          .eSuperTypes.isEmpty).toEdge());
22
23 create Cluster toCluster(EPackage pkg):
24  setViewType(toClusterViewType("package", pkg.name))->
25  setLayoutEngineType(toForceDirectedLayout())->
26  setProperty({toTypeProperty("package")})->
27  cluster.addAll(pkg.eSubpackages.toCluster())->
28  cluster.addAll(pkg.eContents.typeSelect(EClass).toCluster());
29
30 create Cluster toCluster(EClass cls):
31  setProperty(cls, "Cluster", this)->
32  setViewType(toClusterViewType("class", cls.name))->
33  setLayoutEngineType(toForceDirectedLayout())->
34  setProperty({toTypeProperty("class")})->
35  node.addAll(cls.eContents.typeSelect(EAttribute).toNode())->
36  node.addAll(cls.eContents.typeSelect(EOperation).toNode());
37
38 create Node toNode(EAttribute att):
39  setUID(att)->
40  setID(getUID(att))->
41  setViewType(toNodeViewType(att.name))->
42  setProperty({toTypeProperty("attribute")});
43
44 create Node toNode(EOperation op):
45  setUID(op)->
46  setID(getUID(op))->
47  setViewType(toNodeViewType(op.name))->
48  setProperty({toTypeProperty("operation")});
49
50 create Edge toEdge(EClass start, EClass end, String type, String label)
       :
51  let startClone = start.getClone():
52  let endClone = end.getClone():
53  ((Cluster)getProperty(start, "Cluster")).node.add(startClone.toNode()
       )->
54  ((Cluster)getProperty(end, "Cluster")).node.add(endClone.toNode())->
55  setStartID(getUID(startClone))->
56  setEndID(getUID(endClone))->
57  setViewType(toEdgeViewType(label))->
58  setProperty({toTypeProperty(type)});
```

Listing 6-5: Modelltransformation: Ecore zu Graph

Das Graphenmodell mit allen relevanten Informationen aus dem Strukturmodell wird im nächsten Schritt durch eine Modellmodifikation um zusätzliche Schlüssel-Wert-Paare bezüglich Position und Größe erweitert. Eine externe Anwendung, im konkreten Fall WilmaScope, berechnet auf der Grundlage eines kräftebasierten Layout-Algorithmus die Positionen der Elemente des Graphen. Des Weiteren wird in diesem Zusammenhang auch die Größe der einzelnen Elemente

in Abhängigkeit zu den jeweiligen Kindelementen bestimmt. Die Modellmodifikation wurde durch eine selbst implementierte Ablauf-Komponente realisiert, deren Aufruf Listing 6-6 zeigt.

```
53  <!-- Calculate x,y and z position for extended graph model -->
54  <component class="org.x3d.generator.components.WilmaComponent">
55    <inputFile value="${target.dir}${graph.dtd.file}"/>
56    <outputFile value="${target.dir}${graph.extended.file}"/>
57  </component>
```

Listing 6-6: Ablauf-Komponente zur Modellmodifikation

Im letzten Schritt erfolgt die Abbildung dieses erweiterten Graphenmodells auf ein 3D-Modell in Form einer weiteren Modelltransformation. Gemäß den festgelegten Formen und Farben für die verschiedenen Strukturelemente und den berechneten Positionen und Größen werden die dreidimensionalen geometrischen Primitive parametrisiert und erzeugt. Das Rendern des 3D-Modells obliegt dem Browser, in den der generierte Szenegraph geladen wird. Listing 6-7 enthält den Aufruf dieser letzten Modelltransformation.

```
66  <!-- Load xwg and x3d metamodel and transform extended graph model
67     to x3d model  -->
68  <component class="oaw.xtend.XtendComponent">
69    <metaModel idRef="xwg" />
70    <metaModel id="x3d" class="org.openarchitectureware.xsd.XSDMetaModel
          ">
71      <schemaFile value="${metamodel.uri}${metamodel.x3d}" />
72    </metaModel>
73    <invoke value="${extensions.dir}graph2x3d::graph2x3d(
          extendedGraphModel)" />
74    <outputSlot value="x3dModel" />
75  </component>
```

Listing 6-7: Ablauf-Komponente zur Modelltransformation: Graph zu X3D

Ein Ausschnitt dieser Modelltransformation ist in Listing 6-8 enthalten. Die Methode `toScene()` erzeugt den Wurzelknoten des Szenegraphen. Die Informationen aus dem Graphenmodell werden hier extrahiert und auf `Transform`-Knoten abgebildet. Weiterhin wird die Farbe des Hintergrunds gesetzt. Die Methode `toTransform()` ist für Graphen, Knoten und Kanten überladen. Im Listing ist diese nur für Graphen dargestellt, da die durchzuführenden Schritte für Knoten und Kanten sehr ähnlich sind. Zunächst werden die Positions- und Größeninformationen aus dem Graphenmodell bezogen und auf den `Transform`-Knoten übertragen. Danach wird die Art der geometrischen Primitive gesetzt. Für Graphen steht die Wahl zwischen Kugel und Quader und bei Knoten kommen Kegel und Zylinder hinzu. Schließlich werden Farbe und Bezeichner festgelegt, wobei der Bezeichner in einen `Billboard`-Knoten eingebettet ist. Auch hier werden alle Werte für Farben und Formen dynamisch über das Plugin *X3D Generator Eclipse Integration* aus den Einstellungen zur Konfiguration des Generators bezogen.

```
17 create SceneType toScene(WilmaGraph graph):
18   transform.addAll(graph.cluster.eAllContents.typeSelect(Cluster).
         toTransform())->
```

```
19  transform.addAll(graph.cluster.eAllContents.typeSelect(Node).
       toTransform())->
20  transform.addAll(graph.cluster.eAllContents.typeSelect(Edge).
       toTransform())->
21  background.add(toBackground());
22
23 create TransformType toTransform(Cluster cluster):
24  let mat = new MaterialType:
25  let app = new AppearanceType:
26  let shapes = new ShapeType:
27  let shapeLabel = new ShapeType:
28  let billboard = new BillboardType:
29  let transformLabel = new TransformType:
30  let text = new TextType:
31  setScale(cluster.viewType.property.select(prop|prop.key=="Radius").
       value.first().toString().toScale())->
32  setTranslation(cluster.property.select(prop|prop.key=="Position").
       value.first().toString())->
33  switch (getProperty(cluster.property.select(prop|prop.key=="Type").
       value.first().toString() + ".shape")){
34      case 'Sphere': shapes.setSphere(new SphereType)
35      case 'Box': shapes.setBox(new BoxType)
36      default : shapes
37  }->
38  mat.setDiffuseColor(getProperty(cluster.property.select(prop|prop.key
       =="Type").value.first().toString() + ".colour"))->
39  app.material.add(mat)->
40  shapes.setAppearance((AppearanceType)app.clone())->
41  shapes.appearance.material.setTransparency(0.8)->
42  shape.add(shapes)->
43  text.setFontStyle(let this = new FontStyleType: setJustify("MIDDLE"))
       ->
44  shapeLabel.setText(text.setString(cluster.viewType.property.select(
       prop|prop.key=="Label").value.first().toString()))->
45  shapeLabel.setAppearance((AppearanceType)app.clone())->
46  billboard.shape.add(shapeLabel)->
47  transformLabel.setTranslation("0 1 0")->
48  transformLabel.setScale("0.2 0.2 0.2")->
49  transformLabel.billboard.add(billboard)->
50  transform.add(transformLabel);
```

Listing 6-8: Modelltransformation: Graph zu X3D

Optional ist es möglich das generierte 3D-Modell vom X3D-Format in ein VRML-Format zu überführen. Das ist beispielsweise notwendig, um das 3D-Modell im Virtual-Reality-Labor des Instituts für Wirtschaftsinformatik an der Universität Leipzig betrachten zu können. Zu diesem Zweck wurde eine Ablauf-Komponente implementiert die eine XSL-Transformation anstößt. Das XSLT-Stylesheet wurde aus dem Autorenwerkzeug *X3D-Edit* bezogen. Listing 6-9 enthält den Aufruf der Transformation.

```
84  <!-- Transform x3d model into vrml model -->
85  <feature isSelected="vrml">
```

```
86  <component class="org.x3d.generator.components.
        XSLTransformerComponent">
87    <inputFile value="${target.dir}${x3d.file}"/>
88    <transformFile value="${metamodel.uri}${x3d2vrml.xslt}"/>
89    <outputFile value="${target.dir}${wrl.file}" />
90  </component>
91  </feature>
```

Listing 6-9: Ablauf-Komponente zur Umwandlung von X3D in VRML

6.4 Beispiel

Anhand eines Beispiels soll die Funktionsweise des Prototyps noch besser verdeutlicht werden. Aus Gründen der Übersichtlichkeit wird ein Beispiel mit einem geringerem Umfang gewählt. Im Anschluss wird gezeigt, dass auch wesentlich umfangreichere Modelle dreidimensional visualisiert werden können. In Abbildung 6-5 ist das Klassendiagramm zu sehen, das dem Beispiel zu Grunde liegt.

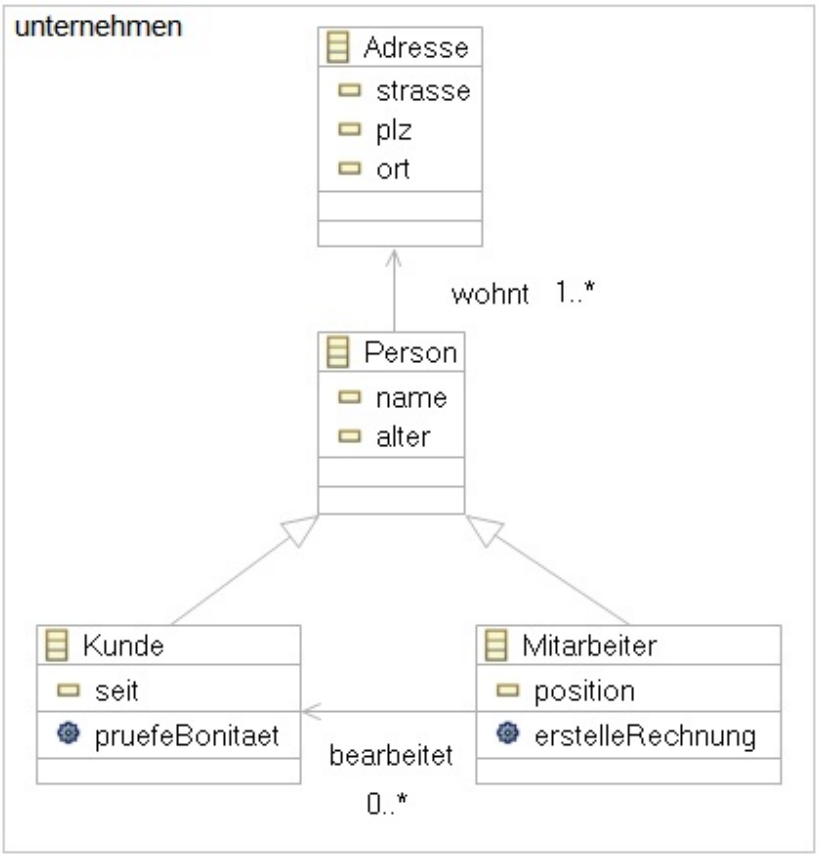


Abbildung 6-5: Klassendiagramm des Strukturmodells

In dem Paket `unternehmen` gibt es vier verschiedene Klassen, die mit Attributen und Methoden näher beschrieben sind und teilweise in Beziehung zueinander stehen. Beispielsweise besitzt die Klasse `Kunde` ein Attribut namens `seit` und eine Methode namens `pruefeBonitaet()`.

Die Klassen `Mitarbeiter` und `Kunde` erben von der Klasse `Person`. Zu einer `Person` gehört mindestens eine `Adresse`. Diese Beziehung wird über eine Assoziation namens `wohnt` verdeutlicht. Weiterhin steht ein `Mitarbeiter` über die Assoziation `bearbeitet` mit einem `Kunden` in Beziehung.
Der Dialog zur Konfiguration des Visualisierungsprozesses ist in Eclipse über den Menüeintrag *Window* über *Preferences* unter *3D Generator Configuration* erreichbar. Der Start des Visualisierungsprozesses erfolgt über ein Kontextmenü, wie es in Abbildung 6-6 zu sehen ist. Der Kontextmenüeintrag *3D Generator* wird nur sichtbar, wenn die Datei die Endung `.ecore` besitzt. Die Statusmeldungen des Generators werden auf der Konsole ausgegeben.

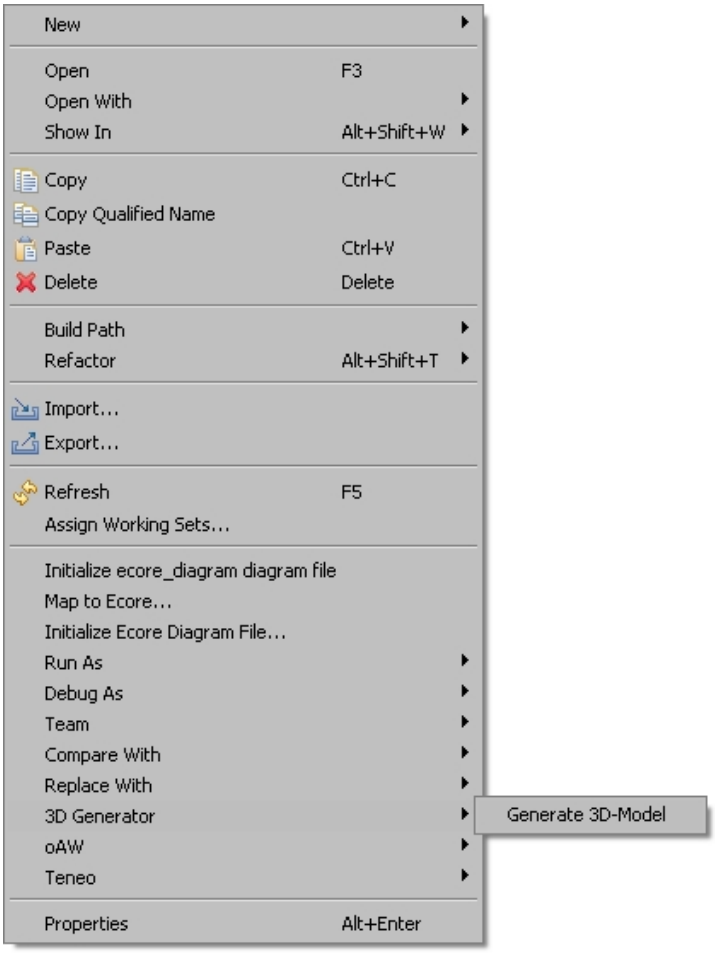


Abbildung 6-6: Kontextmenüeintrag zur Generierung des 3D-Modells

Wird das Beispiel aus Abbildung 6-5 durch den Generator verarbeitet entsteht ein 3D-Modell. Abbildung 6-7 gibt einen Überblick und Abbildung 6-8 bietet eine Detailansicht dieses Modells. Im Beispiel werden Pakete als braune Würfel, Klassen als blaue Kugeln, Methoden als grüne Zylinder, Attribute als rote Kegel und Beziehungen als blaue Linien angezeigt. Die Farben der Pakete und Klassen unterliegen einem Transparenzeffekt und wirken deshalb etwas schwächer. Es sind sowohl Vererbungs- als auch Assoziationsbeziehungen ausgewählt.

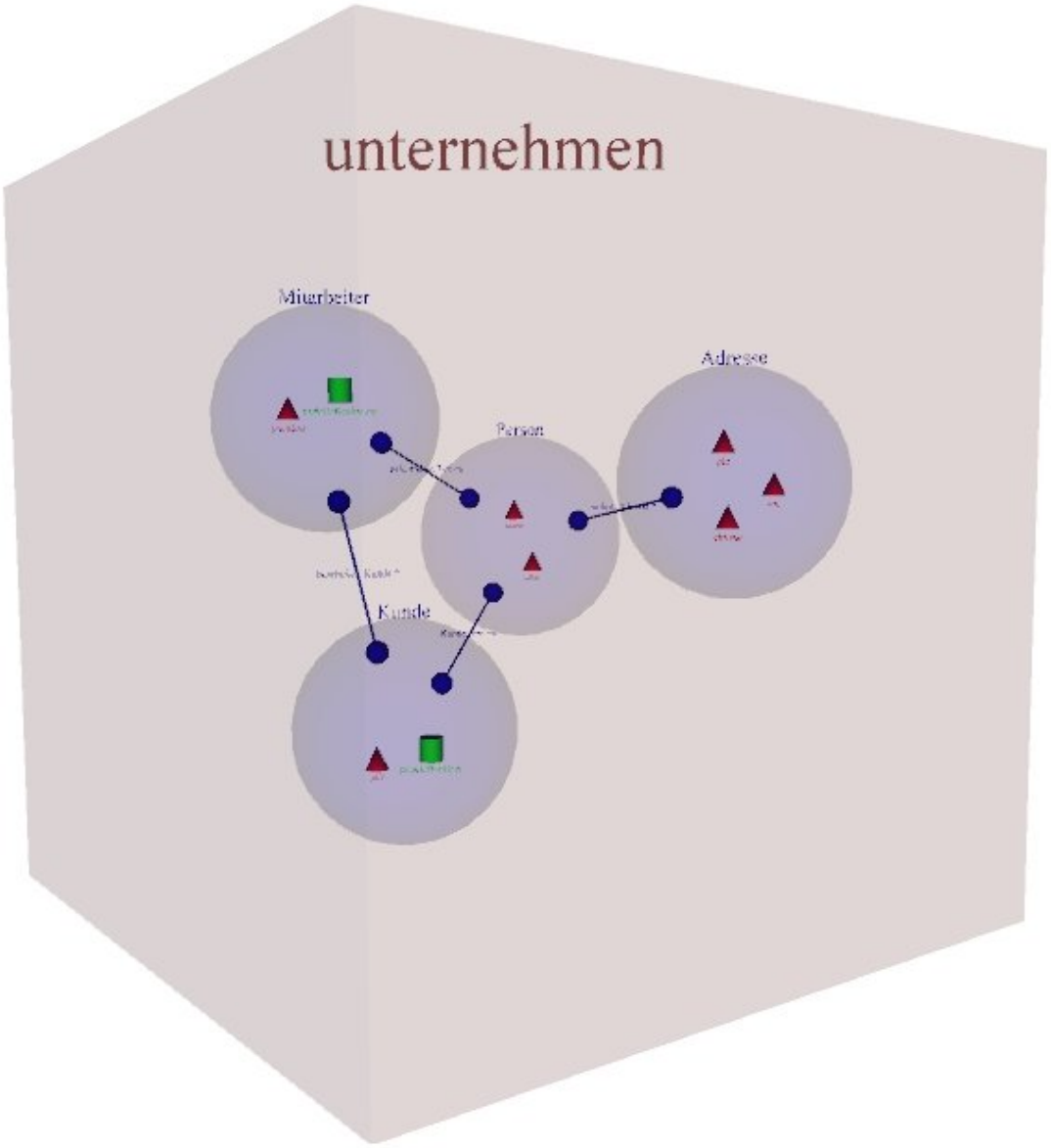


Abbildung 6-7: Beispiel: Überblick über das 3D-Modell

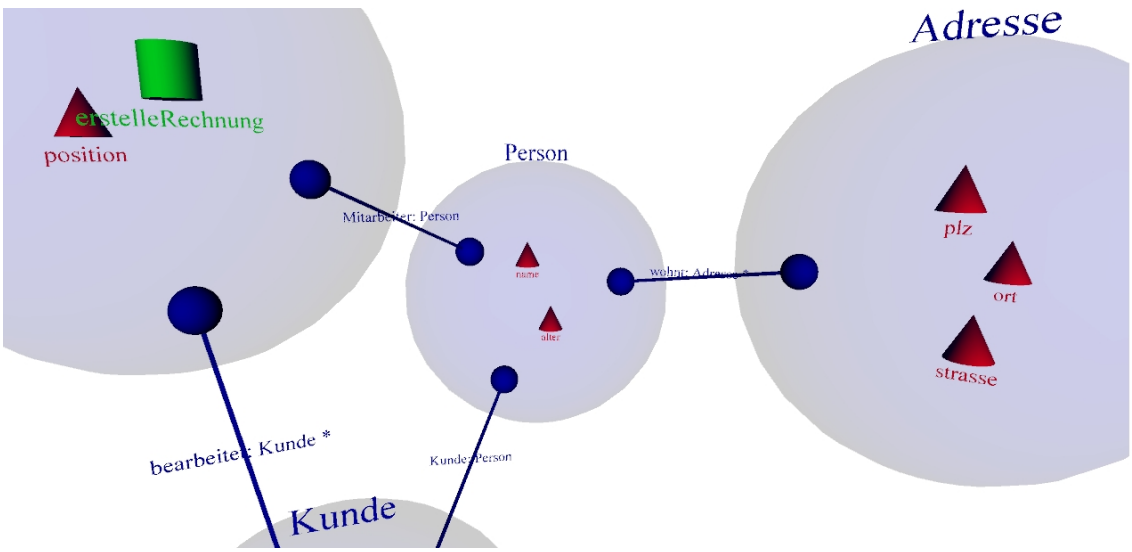


Abbildung 6-8: Beispiel: Detailansicht des 3D-Modells

Im Zeitraum der Anfertigung dieser Arbeit ergab sich die Möglichkeit, ein Strukturmodell aus der Praxis mit mehreren hundert Klassen zu visualisieren. Das Modell bildet die Struktur des Softwaremodellierungswerkzeuges Innovator von MID[41] ab. Ein Ausschnitt aus dem erzeugten 3D-Modell ist in Abbildung 6-9 zu sehen.

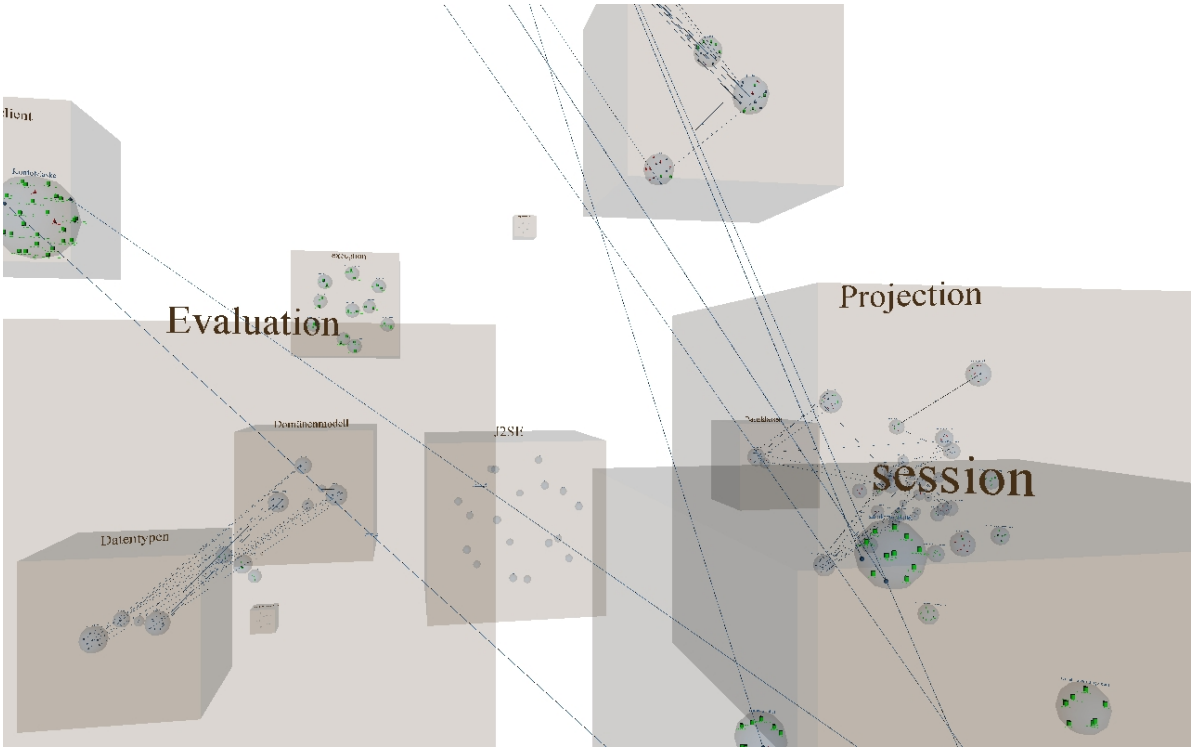


Abbildung 6-9: Ausschnitt aus dem Innovator-Modell

[41] http://www.mid.de

6.5 Evaluation

Gemäß der in Abschnitt 2.1.4 vorgestellten und um die Dimension Automatisierungsgrad erweiterten Taxonomie von Maletic et al. [2002] folgt eine Charakterisierung und Bewertung des Prototyps.

Aufgabe

Die mit dem Prototyp generierten 3D-Modelle zielen darauf ab, das Verständnis über Softwaresysteme durch Visualisierung struktureller Aspekte zu fördern. Das Werkzeug ist darauf ausgelegt, sich nahtlos in den Entwicklungsprozess zu integrieren. Insbesondere sollen die dreidimensionalen Visualisierungen eine zusätzliche und alternative Sicht auf das Softwaresystem sowohl im Kontext der Konzeption und Entwicklung neuer Softwaresysteme als auch bei der Wartung, Erweiterung und Wiederverwendung existierender Softwaresysteme bieten.

Zielgruppe

Der Prototyp dient in erster Linie als Nachweis der Realisierbarkeit des in Kapitel 5 erörterten Konzeptes und besitzt damit den Charakter eines Forschungsprototyps. Demnach setzt sich die derzeitige Zielgruppe aus Professoren, wissenschaftlichen Mitarbeitern und Studenten zusammen. Nach einer weiteren Überarbeitung bestünde die Möglichkeit das Werkzeug für den produktiven Einsatz in der Industrie zu nutzen, um damit die oben angeführten Aufgaben zu unterstützen.

Ausrichtung

Den Gegenstand der Visualisierung bildet die Struktur von Softwaresystemen, das heißt, die unter 6.1.1 identifizierten Elemente Paket, Klasse, Methode und Attribut sowie die Beziehungen zwischen Klassen werden visualisiert. Bei den Beziehungen wird zwischen Vererbung und Assoziationen zwischen Klassen unterschieden. Den Ausgangspunkt bilden Ecore-basierte Modelle, welche die genannten Strukturinformationen enthalten. Infolgedessen unterliegt der Prototyp gewissen Einschränkungen. So fließen zum Beispiel keine Informationen über Metriken, LOC oder ähnliches in den Visualisierungsprozess ein. Die Visualisierung von Verhalten und Evolution von Softwaresystemen deckt der Prototyp nicht ab. Aus Sicht der Skalierbarkeit sei an dieser Stelle auf den vorangegangenen Abschnitt verwiesen. Es wurde gezeigt, dass auch große Strukturmodelle aus der Praxis mit mehreren hundert Klassen visualisiert werden können.

Darstellung

Als Visualisierungs- oder Darstellungstechniken kommen eine abstrakte Metapher in Verbindung mit geschachtelten Visualisierungen und einem kraftgerichteten Layout-Algorithmus zum Einsatz. Zur Gewährleistung expressiver und effektiver Bilder hat der Benutzer die Möglichkeit, den Visualisierungsprozess mittels einer dialogbasierten DSL zu konfigurieren. Dies entspricht dem *visual* oder *computational steering*. Im konkreten Fall kann er die Abbildung der Strukturelemente (Paket, Klasse, Methode und Attribut) auf geometrische Primitive (Quader, Kugel, Zylinder, Kegel) definieren und diese mit einer bestimmten Farbe belegen. Es kann zwischen der Visualisierung von Vererbungs- und Assoziationsbeziehungen oder beiden gewählt werden. Die gewählte Metapher bietet genügend Möglichkeiten, um alle relevanten Aspekte abzubilden

und ist durch die DSL sinnvoll eingeschränkt. Hinsichtlich der Konsistenz der Metapher besteht noch Verbesserungsbedarf bei der Darstellung der Beziehungen, denn diese werden im Moment nur durch deren Beschriftung unterschieden. Nach jedem Visualisierungsprozess desselben Strukturmodells sind die Elemente des 3D-Modells anders angeordnet. Dieser Umstand stellt in Hinblick auf das mentale Modell des Benutzers einen Nachteil dar. In Tabelle 6-2 sind die von Shneiderman aufgestellten Kriterien für eine Visualisierung aufgelistet und in Bezug zu X3D beziehungsweise den Prototyp bewertet.

Kriterium	**Unterstützung**
Vermittlung eines Überblicks	+
Bereitstellung einer Zoom-Funktion	+
Filter-Funktion	+
Details auf Nachfrage	+
Anzeige von Beziehung zwischen den Objekten	+
Aufzeichnung und Abspielung getätigter Aktionen	+ / −
Extraktion von und Anfragen auf Objekte	−

Tabelle 6-2: Bewertung des Prototyps nach den Kriterien von Shneiderman

Es wird ersichtlich, dass fast alle Kriterien erfüllt werden. Die Möglichkeit zum Aufzeichnen und Abspielen getätigter Aktionen hängt vom angebotenen Funktionsumfang des eingesetzten Mediums ab. Lediglich das Extrahieren von Anfragen auf Objekte wird nicht unterstützt.

Medium

Die 3D-Modelle liegen entweder im X3D- oder VRML-Format vor. Damit können sie an einem konventionellen PC-Monitor innerhalb eines X3D-Browsers und im Virtual-Reality-Labor des Institutes für Wirtschaftsinformatik an der Universität Leipzig betrachtet und interaktiv exploriert werden. Außerdem konnten die Modelle nach einer Vorverarbeitung in 3D Studio Max auf autostereoskopischen Monitoren dargestellt werden.

Automatisierungsgrad

Der Visualisierungsprozess ist derart gestaltet, dass ausgehend von einer Anforderungsspezifikation durch einen Benutzer ein 3D-Modell nach Bedarf automatisch erzeugt werden kann. Manuelle Eingriffe sind nicht notwendig. In diesem Sinne kann von Vollautomation gesprochen werden. Der Generierungsprozess ist durch Modelltransformationen in der Sprache Xtend realisiert. Insgesamt gibt es zwei Modelltransformationen, wobei jede mit ungefähr 100 LOC beschrieben ist. In Hinblick auf die Effizienz des Visualisierungsprozesses entspricht das einem angemessenen Verhältnis von Aufwand zu Nutzen.

6.6 Probleme

An einigen Stellen weist der Prototyp noch wenige Schwächen technischer Natur auf. Diese werden nachfolgend dokumentiert.

Im Moment muss dem erzeugten XML-basierten Graphenmodell in einem Zwischenschritt der Pfad zur DTD des Graphenformats XWG angegeben werden. Dies ist Voraussetzung dafür, dass WilmaScope das Graphenmodell fehlerfrei modifiziert. Ideal wäre es, wenn das Hinzufügen des Pfades zur DTD während des Schreibens durch die Ablauf-Komponente von openArchitectureWare erfolgen würde. Das Werkzeug unterstützt dies aber nicht, da es nur auf XML-Schema ausgelegt ist.
Es ist im Prinzip nicht möglich, die Layout-Algorithmen von WilmaScope aufzurufen, ohne dass die Oberfläche gleichzeitig gestartet wird. Bei Aufruf der Layout-Algorithmen zur Positions- und Größenermittlung der Graphenelemente aus einer Ablauf-Komponente heraus, musste sich deswegen eines Tricks bedient werden. Damit die Algorithmen trotzdem funktionieren, wurde ein Offscreen-Puffer initialisiert. Auf diese Weise konnte dieses Problem behoben werden.
Das dynamische Laden des Ecore-basierten Strukturmodells verläuft nicht in jedem Fall fehlerfrei. Beim erstmaligen Laden des Modells gibt es keine Fehler. Danach kommt es aus noch ungeklärter Ursache in unregelmäßigen Abständen zu einem Abbruch des Generierungsprozesses. Dieser Fehler konnte in einigen Fällen durch ein automatisches Säubern des Projektes behoben werden. In jedem Fall half ein Neustart von Eclipse.
Ein letztes bekanntes Problem bezieht sich auf die Darstellung des 3D-Modells in dem Virtual-Reality-Labor des Instituts für Wirtschaftsinformatik an der Universität Leipzig. Hierbei werden Kanten und Beschriftungen nicht angezeigt. Erst wenn das 3D-Modell in VRML in 3D Studio Max importiert und wieder als VRML exportiert wird, werden Kanten und Beschriftungen sichtbar. Eigenartigerweise wird das generierte 3D-Modell ohne die Vorverarbeitung von 3D Studio Max sowohl im X3D- als auch im VRML-Format in allen getesteten X3D-Browsern lückenlos und erwartungsgemäß angezeigt. Es wird vermutet, dass die Software zur Steuerung des Virtual-Reality-Labors eine veraltete VRML-Version unterstützt.

7 Fazit und Ausblick

In dieser Arbeit wurden gemäß dem konstruktionsorientierten Paradigma drei Artefakte entwickelt:

- Modell: Das Konzept einschließlich der konzeptionellen und technischen Beschreibung zur vollautomatisierten Erzeugung dreidimensionaler Modelle zur Visualisierung von Softwaresystemen
- Methode: Der Leitfaden zur Entwicklung eines solchen Generators
- Instanz: Die prototypische Implementierung eines Generators zur Visualisierung der Struktur von Softwaresystemen

Es konnte gezeigt werden, dass unter Ausnutzung der Synergien zwischen dem modellgetriebenen und dem generativen Paradigma ein Konzept erstellt werden konnte, das es ermöglicht, dreidimensionale Visualisierungen von Softwaresystemen automatisch zu generieren. Im Mittelpunkt steht ein Generator, der ausgehend von einer Anforderungsspezifikation vollautomatisiert 3D-Modelle erzeugt. Die Modelle können dabei die Struktur, das Verhalten oder die Evolution von Softwaresystemen visualisieren. Im Zuge einer Technikprojektion wurden Techniken identifiziert, mit denen der Generator realisiert werden kann. Dabei spielen mehrstufige Modelltransformationen in Verbindungen mit dem Werkzeug openArchitectureWare eine zentrale Rolle.
Das konzeptionelle Gerüst des Generators wurde mit der Terminologie des generativen Domänenmodells beschrieben. Auf diese Weise lässt sich der etablierte Prozess der Domänenentwicklung auf den Erstellungsprozess eines solchen Generators anwenden. Die Ergebnisse dieses Prozesses stellen wiederverwendbare Einheiten dar, die die Grundlage für den automatischen Visualisierungsprozess bilden.
Zur Validierung des entwickelten Konzeptes wurde ein Prototyp implementiert, der auf die Visualisierung der Struktur von Softwaresystemen abzielt. Dieser erzeugt aus Ecore-basierten Modellen nach Benutzeranforderungen ein 3D-Modell. Die Synthese des Modells erfolgt über mehrstufige Modelltransformationen und -modifikationen. Die Schwierigkeit der Anordnung der Elemente des 3D-Modells im Raum wurde durch die Zwischenschaltung eines Graphenmodells gelöst. Dadurch ist es möglich, auf bestehende Layout-Algorithmen zurückzugreifen, ohne diese selbst implementieren zu müssen. Der Prototyp lässt sich als Plugin in Eclipse integrieren. Die mit dem Werkzeug generierten Visualisierungen basieren auf X3D - einem freien und standardisierten Format für dreidimensionale Grafiken. Durch den Einsatz von X3D ist eine Anwendungs- und Plattformunabhängigkeit und damit auch Portabilität der Visualisierung gegeben. Die Abwärtskompatibilität von X3D erlaubt es, die generierten 3D-Modelle in VRML umzuwandeln. Das Virtual-Reality-Labor am Institut für Wirtschaftsinformatik akzeptiert dieses Format. Damit lassen sich die Modelle sowohl in der virtuellen Umgebung mit vorheriger Konvertierung als auch an einem herkömmlichen PC mit einem entsprechenden Browser ohne vorherige Konvertierung interaktiv explorieren und erfahren. Die Funktion zur Umwandlung von X3D in VRML ist in dem Prototyp implementiert. Schließlich erfolgte die Evaluation des

Prototyps unter Verwendung etablierter Methoden aus dem Bereich der Softwarevisualisierung. Sowohl bei dem Konzept als auch bei dem Prototyp wurde darauf Wert gelegt, die Entwicklungsprozesse als iterative Suchprozesse darzustellen und die getroffenen Entscheidungen klar nachvollziehbar zu begründen.
Neben den drei angestrebten und erreichten Zielen haben sich noch weitere positive Effekte ergeben, die sich wie folgt zusammenfassen lassen:

- Entwicklungsprozess: Einfache Integration in die Entwicklungsumgebung Eclipse
- Portabilität: Anwendungs- und Plattformunabhängigkeit durch standardisiertes Format X3D
- Automatisierungsgrad: Vollständige Generierung der 3D-Modelle
- Wiederverwendung: Layout-Algorithmen, Komponenten in Form kompletter X3D-Dateien, parametrisierbarer Prototypen oder integrierter Knotentypen
- Flexibilität: Visualisierungsprozess auf Basis komponentenbasierter *Workflows*
- Erweiterbarkeit: Ausnutzung des Plugin-Konzeptes von Eclipse
- Lesbarkeit und Effizienz: Verständlichere Syntax und wesentlich weniger LOC als XSLT
- Typsicherheit: Generierung von validem und wohlgeformtem XML

Trotz der erreichten Ziele besteht noch Verbesserungspotential für den Prototyp. Dabei geht es in erster Linie um die Beseitigung der in Abschnitten 6.6 und 6.5 angeführten Probleme.
In der aktuellen Version des Prototyps werden Informationen über Pakete, Klassen, Methoden und Attribute einschließlich der Beziehungen zwischen Klassen aus Ecore-basierten Modellen extrahiert und visualisiert. Es wäre möglich, weitere Informationen aus dem Angebot von Ecore zu verarbeiten. Als Beispiele seien hier Datentypen, Parameter für Methoden oder Annotationen angeführt. Zudem können alternative Layout-Algorithmen bis hin zu anderen Visualisierungstechniken angewendet und auf Tauglichkeit überprüft werden. Die hier eingesetzte abstrakte Metapher stellt nur eine Möglichkeit dar. So lassen sich die geometrischen Primitive auch auf ganz andere Weise miteinander kombinieren.
Die Suche nach geeigneten Werkzeugen, die dreidimensionale Layout-Algorithmen und gleichzeitig XML-basierte Graphenformate unterstützen, stellte eine Herausforderung dar. Schließlich wurde zwar ein passendes Format und ein Werkzeug gefunden, aber diese Lösung ist - wie bereits dargestellt - suboptimal. Wünschenswert wäre hier eine Art erweiterbare Bibliothek für dreidimensionale Layout-Algorithmen mit Unterstützung gängiger Formate wie GXL oder GraphML.
Der Prototyp beschränkt sich auf die Visualisierung der Struktur von Software. Eine zusätzliche Betrachtung der Visualisierung des Verhaltens oder der Evolution von Software hätte den Rahmen dieser Arbeit gesprengt. In Form weiterer Untersuchungen wäre zu erforschen, inwieweit sich das hier entwickelte Konzept auch auf die anderen beiden Bereiche der Softwarevisualisierung übertragen läßt. X3D bietet neben den zahlreichen geometrischen Primitive auch ein Ereignismodell wie in Abschnitt 4.2.6 beschrieben. Gerade in Hinblick auf die Visualisierung dynamischer und historisierter Informationen könnte dieses eine interessante Rolle spielen.

Eine sehr anspruchsvolle und zugleich kreative Aufgabe stellt die Entwicklung geeigneter Mappings oder die Identifikation nutzbarer Metaphern dar. In Abschnitt 2.1.6.3 wurden bereits einige natürliche Metaphern wie zum Beispiel die Stadt-, Solarsystem- oder Baumring-Metapher angeführt. Diese Auflistung ist bei weitem nicht vollständig und bedarf weiterer Recherchen. In Verbindung mit diesem Sachverhalt wäre zu prüfen, ob es in einem angemessenen Aufwand möglich ist, diese komplexeren Strukturen aufzubrechen, und mit X3D beispielsweise in Form kombinierbarer X3D-Prototypen umzusetzen.
Bei der Implementierung des Prototyps wurde auf bestehende Ecore-basierte Metamodelle zurückgegriffen. Es wäre natürlich auch möglich, eigene Metamodelle zu entwerfen, die auf die Visualisierung spezialisiert sind. Einen möglichen Ausgangspunkt hierfür bietet MOOSE[42]. MOOSE ist eine Kollaborationsplattform für Forschungsaktivitäten mit Fokus auf Softwareanalyse und Informationsvisualisierung. Spezifische Metamodelle für Struktur, Verhalten und Evolution von Softwaresystemen sind über diese Plattform frei zugänglich. In diesem Zusammenhang wäre es interessant zu untersuchen, wie sich relevante Informationen gegebenenfalls automatisiert aus Softwaresystemen ableiten und in eine strukturierte Form bringen lassen.
In dieser Arbeit wurde der direkte Nutzen der dritten Dimension im Rahmen der Softwarevisualisierung nicht näher untersucht. Vielmehr wurde davon ausgegangen, dass dieser existiert und die technische Sicht zur Erstellung von 3D-Modellen beschrieben. In weiteren Arbeiten könnte nun mit Blick auf und Wissen über die Realisierbarkeit der Nutzen der dritten Dimension erörtert werden. Diese *Bottom-up*-Vorgehensweise birgt den Vorteil, dass konkrete Techniken existieren, um 3D-Modelle zu erzeugen, die zum Testen der aufzustellenden Hypothesen dienen.

Zum Schluss dieser Arbeit wird die in der Einleitung angeführte Analogie von Brooks noch einmal aufgegriffen. Er bezeichnet die grundlegenden Probleme der Softwareentwicklung als Werwölfe. Eines davon betrifft die Komplexität und ein anderes die Unsichtbarkeit von Software. Brooks [1987] behauptet, dass Software unsichtbar und nicht visualisierbar sei. Bezogen auf das Gesamtsystem mag er damit immer noch Recht haben. Bis dato ist es nicht möglich, ein komplettes Softwaresystem mit allen dazugehörigen Artefakten zu visualisieren. Es konnte jedoch gezeigt werden, dass sich wesentliche Aspekte von Softwaresystemen durchaus visualisieren lassen. Damit werden Teile des vorerst unsichtbaren und komplexen Gebildes namens Software sichtbar und können auf einer gegenständlichen Ebene verstanden sowie begreifbar gemacht werden. In diesem Sinne handelt es sich bei dieser Arbeit zwar nicht um eine silberne Kugel, aber sie stellt einen Schritt in die Richtung der Komplexitätsbeherrschung von Software durch dreidimensionale Softwarevisualisierung dar.

[42] http://moose.unibe.ch

Literaturverzeichnis

Ahmed, A., Dwyer, T., Forster, M., Fu, X., Ho, J., Hong, S.H., Koschützki, D., Murray, C., Nikolov, N., Taib, R., Tarassov, A., Xu, K., GEOMI: GEOmetry for Maximum Insight, 2006.

Anslow, C., Marshall, S., Noble, J., Biddle, R., X3D Software Visualization, Proceedings of the New Zealand Computer Science Students Research Conference, University of Waikato, Hamilton, New Zealand, 2007, URL http://www.nzdl.org/gsdl/collect/nzcsrsc0/index/assoc/HASH010b.dir/doc.pdf.

Anslow, C., Noble, J., Marshall, S., Biddle, R., Evaluating Extensible 3D Graphics For Use in Software Visualization, Symposium on Visual Languages and Human-Centric Computing (VLHCC), 2008.

Auber, D., Tulip, P. Mutzel, M. Jünger, S. Leipert (Hrsg.), 9th Symp. Graph Drawing, *Lecture Notes in Computer Science*, Bnd. 2265, Springer-Verlag, 2001, S. 335–337.

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

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

Battista, G.D., Eades, P., Tamassia, R., Tollis, I.G., Graph Drawing: Algorithms for the Visualization of Graphs, Prentice-Hall, 1999, ISBN 0-13-301615-3.

Boccuzzo, S., Gall, H.C., CocoViz: Towards Cognitive Software Visualizations, Proceedings of IEEE International Workshop on Visualizing Software for Understanding and Analysis (VisSoft 2007), 8, IEEE Computer Society, 2007.

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

Brandes, U., Eiglsperger, M., Herman, I., Himsolt, M., Marshall, M., GraphML Progress Report Structural Layer Proposal, 2002.

Brooks, F.P., No Silver Bullet: Essence and Accidents of Software Engineering, IEEE Computer, 20 (1987) 4, S. 10–19.

Brutzman, D., Daly, L., X3D: Extensible 3D Graphics for Web Authors, 1. Aufl., Elsevier, 2007, ISBN 978-0-12-088500-8.

Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J., Eclipse Modeling Framework, Pearson Education, 2004, ISBN 0-13-142542-0.

Bull, R.I., Integrating dynamic views using model driven development, Centre for Advanced Studies on Collaborative Research, 2006, S. 219–232.

Card, S.K., Mackinlay, J.D., Shneiderman, B. (Hrsg.), Readings in information visualization: using vision to think, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999, ISBN 1-55860-533-9.

Charters, S.M., Knight, C., Thomas, N., Munro, M., Visualisation for informed decision making; from code to components, SEKE 02: Proceedings of the 14th international conference on Software engineering and knowledge engineering, ACM, New York, NY, USA, 2002, ISBN 1-58113-556-4, S. 765–772.

Churcher, N., Keown, L., Irwin, W., Virtual Worlds for Software Visualisation, A. Quigley (Hrsg.), SoftVis 99: Software Visualisation Workshop, University of Technology, 1999, S. 9–16.

Csardi, G., The igraph Package, 2008, URL http://cran.r-project.org/web/packages/igraph/igraph.pdf.

Czarnecki, K., Overview of Generative Software Development, 2005.

Czarnecki, K., Eisenecker, U.W., Generative Programming Methods, Tools and Applications, Addison-Wesley, 2000, ISBN 0-201-30977-7.

Czarnecki, K., Helsen, S., Feature-based survey of model transformation approaches, IBM Systems Journal, 45 (2006) 3, S. 621–646.

Daum, B., Java-Entwicklung mit Eclipse 3.3 - Anwendungen, Plugins und Rich Clients, 5. Aufl., dpunkt.verlag, 2008, ISBN 978-3-89864-504-1.

Diehl, S., Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software, Springer, 2007, ISBN 978-3-540-46504-1.

Dugerdil, P., Alam, S., Execution Trace Visualization in a 3D Space, 2008, URL http://www.inf.unisi.ch/projects/evospaces/publications/dugerdil-trace3dspace.pdf.

Dwyer, T., Extending the WilmaScope 3D graph visualisation system: software demonstration, APVis 05: proceedings of the 2005 Asia-Pacific symposium on Information visualisation, Australian Computer Society, Inc., Darlinghurst, Australia, 2005, ISBN 1-920-68227-9, S. 39–45.

Dyck, B., Hanlon, S., Wismath, S., Gluskap User's Manual, 2004a, URL http://www.cs.uleth.ca/~vpak/gluskap/docs/manual.pdf.

Dyck, B., Joevenazzo, J., Nickle, E., Wilsdon, J., Wismath, S., GLuskap: Visualization and Manipulation of Graph Drawings in 3-Dimensions, 2004b, URL http://www.springerlink.com/content/nrfau9j5q6utv4bd.

Eades, P.A., A heuristic for graph drawing, Congressus Numerantium, Bnd. 42, 1984, S. 149–160.

Efftinge, S., Friese, P., Haase, A., Hübner, D., Kadura, C., Kolb, B., Köhnlein, J., Moroff, D., Thoms, K., Völter, M., Schönbach, P., Eysholdt, M., openArchitectureWare User Guide, Version 4.3, 2008, URL http://www.eclipse.org/gmt/oaw/doc/4.3/openArchitectureWare-4.3-Reference.pdf.

Feijs, L., Jong, R.D., 3D visualization of software architectures, Commun. ACM, 41 (1998) 12, S. 73–78, ISSN 0001-0782.

Fowler, M., Inversion of Control Containers and the Dependency Injection pattern, 2004, URL http://martinfowler.com/articles/injection.html.

Frankel, D.S., Model Driven Architecture - Applying MDA to Enterprise Computing, Wiley Publishing, Inc., 2003, ISBN 0-471-31920-1.

Fruchterman, T.M.J., Reingold, E.M., Graph drawing by force-directed placement, Softw. Pract. Exper., 21 (1991) 11, S. 1129–1164, ISSN 0038-0644.

Gansner, E.R., North, S.C., An open graph visualization system and its applications to software engineering, Softw. Pract. Exper., 30 (2000) 11, S. 1203–1233, ISSN 0038-0644.

Gentleman, R., Whalen, E., Huber, W., Falcon, S., The graph Package, 2008, URL http://cran.r-project.org/web/packages/graph/graph.pdf.

Gogolla, M., Radfelder, O., Richters, M., Towards three-dimensional representation and animation of uml diagrams, Proc. 2 nd Int. Conf. Unified Modeling Language, IEEE Computer Society Press, 1999, S. 489–502.

Gracanin, D., Matkovic, K., Eltoweissy, M., Software Visualization, 2005, URL http://www.cg.tuwien.ac.at/research/publications/2005/gracanin-2005-soft/gracanin-2005-soft-PDF.pdf.

Graham, H., Yang, H.Y., Berrigan, R., A Solar System Metaphor for 3D Visualisation of Object Oriented Software Metrics, 2004, URL citeseer.ist.psu.edu/723578.html.

Gruhn, V., Pieper, D., Röttgers, C., MDA - Effektives Software-Engineering mit UML 2 und Eclipse, Springer, Berlin, Heidelberg, 2006, ISBN 978-3-540-28744-5.

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

Hasebrook, J., Multimedia-Psychologie : eine neue Perspektive menschlicher Kommunikation, Spektrum Akademischer Verlag, Heidelberg, Berlin, Oxford, 1995, ISBN 3-86025-287-9.

Hevner, A.R., March, S.T., Park, J., Ram, S., Design Science in Information Systems Research, MIS Quarterly, 28 (2004) 1, S. 75–105.

Holt, R.C., Schürr, A., Sim, S.E., Winter, A., GXL: A graph-based standard exchange format for reengineering, Science of Computer Programming, 60 (2006) 2, S. 149–170, URL http://www.sciencedirect.com/science/article/B6V17-4HKCYNK-1/2/2c18ce2a27eebb769c531596c1fd317c.

Johnson, C., Parker, S.G., Hansen, C., Kindlmann, G.L., Livnat, Y., Interactive Simulation and Visualization, Computer, 32 (1999) 12, S. 59–65, ISSN 0018-9162.

Kamada, T., Kawai, S., An algorithm for drawing general undirected graphs, Inf. Process. Lett., 31 (1989) 1, S. 7–15, ISSN 0020-0190.

Kent, S., Model Driven Engineering, Integrated Formal Methods, (2002), S. 286–298.

Kleppe, A.G., Warmer, J., Bast, W., MDA Explained: The Model Driven Architecture: Practice and Promise, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003, ISBN 032119442X.

Knight, C., Munro, M., Comprehension with[in] Virtual Environment Visualisations, IWPC, 1999, S. 4–11.

Knight, C., Munro, M., Virtual but Visible Software, IV, 2000, S. 198–205.

Koschke, R., Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey, Journal of Software Maintenance, 15 (2003) 2, S. 87–109.

Kotzyba, M., Low, T., Matthes, K., Vergleich von Toolkits zur Visualisierung von und Interaktion mit Graphen, User Interface Engineering, 2008, S. 100–118.

Kuhn, T.S., The Structure of Scientific Revolution, 3. Aufl., University of Chicago Press, 1962, ISBN 0-226-45808-3.

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

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

Maletic, J.I., Leigh, J., Marcus, A., Dunlap, G., Visualizing Object Oriented Software in Virtual Reality, 2001.

Maletic, J.I., Marcus, A., Collard, M.L., A Task Oriented View of Software Visualization, VISSOFT 02: Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis, IEEE Computer Society, Washington, DC, USA, 2002, S. 32–40.

Marcus, A., Comorski, D., Sergeyev, A., Supporting the Evolution of a Software Visualization Tool Through Usability Studies, IWPC 05: Proceedings of the 13th International Workshop on Program Comprehension, IEEE Computer Society, Washington, DC, USA, 2005, ISBN 0-7695-2254-8, S. 307–316.

Mcintosh, P., Hamilton, M., van Schyndel, R., X3D-UML: enabling advanced UML visualisation through X3D, Web3D 05: Proceedings of the tenth international conference on 3D Web technology, ACM, New York, NY, USA, 2005, ISBN 1-59593-012-4, S. 135–142.

Mcintosh, P., Hamilton, M., van Schyndel, R., X3D-UML: 3D UML State Machine Diagrams, 2008.

Meyers Online Lexikon, Paradigma, 2008, URL http://lexikon.meyers.de/.

Mulder, J.D., van Wijk, J.J., van Liere, R., A survey of computational steering environments, Future Gener. Comput. Syst., 15 (1999) 1, S. 119–129, ISSN 0167-739X.

Müller, J., Technikprojektion zur generativen Programmierung mit openArchitectureWare, Diplomarbeit, Universität Leipzig Wirtschaftswissenschaftliche Fakultät, 2008.

Myers, B.A., Taxonomies of Visual Programming and Program Visualization, Journal of Visual Languages and Computing, 1 (1990) 1, S. 97–123.

Object Management Group, MDA Guide Version 1.0.1, 2003, URL http://www.omg.org/docs/omg/03-06-01.pdf.

Object Management Group, Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Final Adopted Specification ptc/07-07-07, 2007, URL http://www.omg.org/docs/ptc/07-07-07.pdf.

Phong, B.T., Illumination for computer generated pictures, Commun. ACM, 18 (1975) 6, S. 311–317, ISSN 0001-0782.

Price, B.A., Baecker, R., Small, I.S., A Principled Taxonomy of Software Visualization, J. Vis. Lang. Comput., 4 (1993) 3, S. 211–266, URL http://mcs.open.ac.uk/bp5/papers/1993-JVLC/.

Reingold, E.M., Tilford, J.S., Tidier Drawings of Trees, IEEE Transactions on Software Engineering, 7 (1981) 2, S. 223–228, ISSN 0098-5589.

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

Rekimoto, J., Green, M., The information cube: Using transparency in 3D information visualization, In Proceedings of the Third Annual Workshop on Information Technologies & Systems (WITS'93), 1993, S. 125–132.

Robertson, G.G., Mackinlay, J.D., Card, S.K., Cone Trees: animated 3D visualizations of hierarchical information, CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems, ACM, New York, NY, USA, 1991, ISBN 0-89791-383-3, S. 189–194.

Roman, G.C., Cox, K.C., A Taxonomy of Program Visualization Systems, Computer, 26 (1993) 12, S. 11–24, ISSN 0018-9162.

dos Santos, S., Brodlie, K., Gaining understanding of multivariate and multidimensional data through visualization, Computers & Graphics, 28 (2004) 3, S. 311–325, URL http://www.sciencedirect.com/science/article/B6TYG-4CGGC80-2/2/ec4cf64cd11e332d27293c6cb0a8a9a3.

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

Shneiderman, B., The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations, Proceedings of the 1996 IEEE Symposium on Visual Languages, IEEE Computer Society, 1996, S. 336–343.

Stahl, T., Völter, M., Efftinge, S., Haase, A., Modellgetriebene Softwareentwicklung - Techniken, Engineering, Management, 2. Aufl., dpunkt.verlag, 2007, ISBN 978-3898644488.

Stasko, J.T., Patterson, C., Understanding and characterizing program visualization systems, Technical report git-gvu-91/17, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, 1991, URL citeseer.ist.psu.edu/stasko91understanding.html.

Strahringer, S., Ein sprachbasierter Metamodellbegriff und seine Verallgemeinerung durch das Konzept des Metaisierungsprinzips, Modellierung, 1998, URL http://SunSITE.Informatik.RWTH-Aachen.DE/Publications/CEUR-WS/Vol-9/Strahringer.ps.

Sugiyama, K., Tagawa, S., Toda, M., Methods for Visual Understanding of Hierarchical System Structures, IEEE Trans. Systems, Man and Cybernetics, 11 (1981) 2, S. 109–125, ISSN 0018-9472.

Therón, R., Hierarchical-Temporal Data Visualization Using a Tree-Ring Metaphor, 2006.

Web3D-Konsortium, Virtual Reality Modelling Language (VRML97): Spezifikation, 1997, URL http://www.web3d.org/x3d/specifications/vrml/.

Web3D-Konsortium, Extensible 3D (X3D): Spezifikation, 2008, URL http://www.web3d.org/x3d/specifications/x3d.

Wettel, R., Lanza, M., Visualizing Software Systems as Cities, Visualizing Software for Understanding and Analysis, VISSOFT 2007, 4th IEEE International Workshop on, 2007, S. 92–99, URL http://www.inf.unisi.ch/projects/evospaces/publications/Wettel07b.pdf.

Anhang A - Ecore-Modell

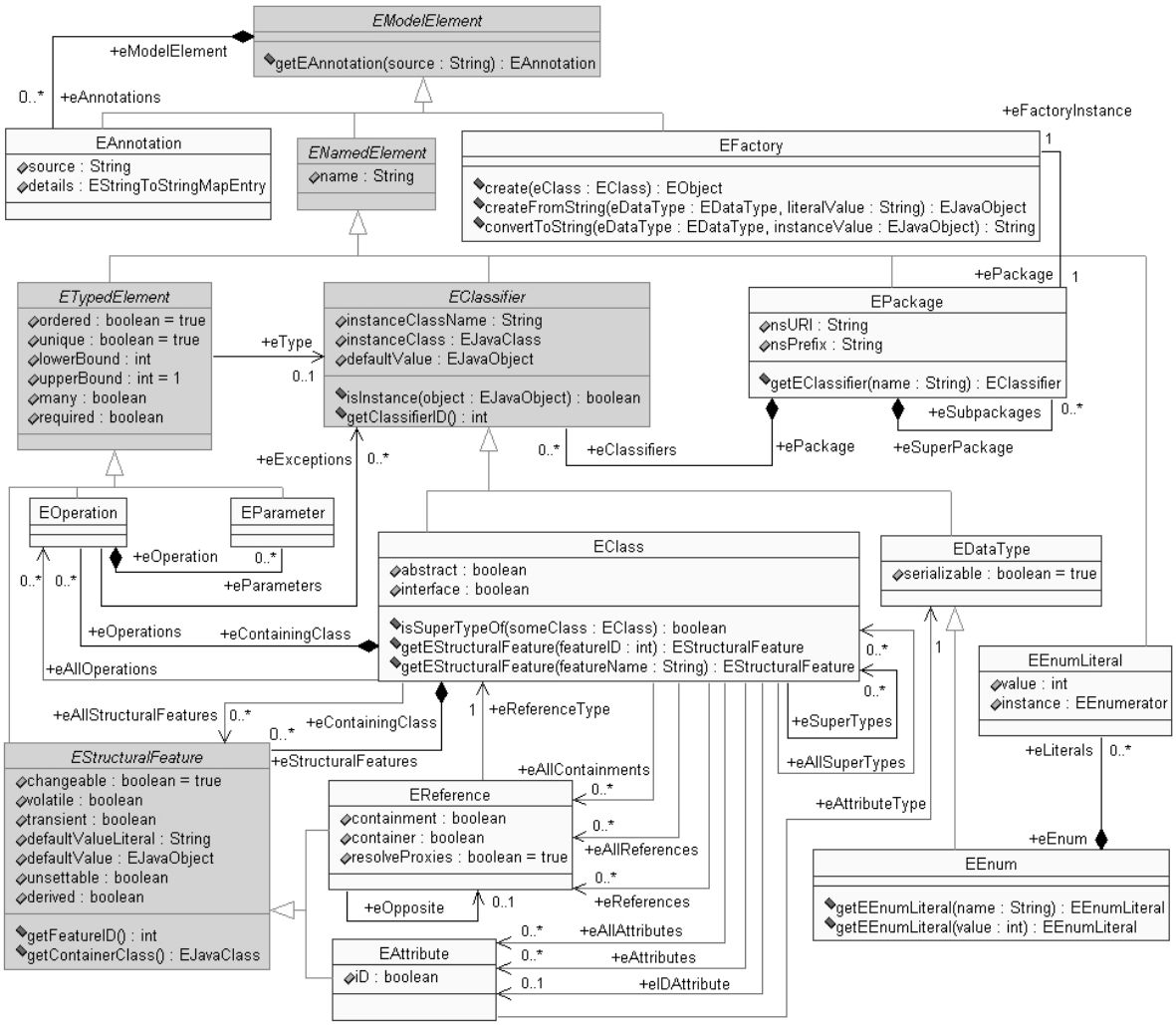


Abbildung A-1: Klassendiagramm von Ecore mit Attributen und Methoden

Anhang B - Spezifikation der Knotentypen des Prototyps

```
Transform : X3DGroupingNode {
  MFNode   [in]   addChildren         [X3DChildNode]
  MFNode   [in]   removeChildren      [X3DChildNode]
  SFVec3f  [in,out] center     0 0 0  (-8,8)
  MFNode   [in,out] children   []     [X3DChildNode]
  SFNode   [in,out] metadata   NULL   [X3DMetadataObject]
  SFRotation [in,out] rotation 0 0 1 0 [-1,1] or (-8,8)
  SFVec3f  [in,out] scale      1 1 1  (-8, 8)
  SFRotation [in,out] scaleOrientation 0 0 1 0 [-1,1] or (-8,8)
  SFVec3f  [in,out] translation 0 0 0 (-8,8)
  SFVec3f  []     bboxCenter   0 0 0  (-8,8)
  SFVec3f  []     bboxSize     -1 -1 -1 [0,8) or -1 -1 -1
}
```

Listing B-1: Transform

```
Billboard : X3DGroupingNode {
  MFNode [in]  addChildren       [X3DChildNode]
  MFNode [in]  removeChildren    [X3DChildNode]
  SFVec3f [in,out] axisOfRotation 0 1 0 (-8,8)
  MFNode [in,out] children []    [X3DChildNode]
  SFNode [in,out] metadata NULL  [X3DMetadataObject]
  SFVec3f []   bboxCenter 0 0 0  (-8,8)
  SFVec3f []   bboxSize   -1 -1 -1 [0,8) or -1 -1 -1
}
```

Listing B-2: Billboard

```
Shape : X3DShapeNode {
  SFNode [in,out] appearance NULL [X3DAppearanceNode]
  SFNode [in,out] geometry NULL [X3DGeometryNode]
  SFNode [in,out] metadata NULL [X3DMetadataObject]
  SFVec3f []   bboxCenter 0 0 0 (-8,8)
  SFVec3f []   bboxSize -1 -1 -1 [0,8) or -1 -1 -1
}
```

Listing B-3: Shape

```
Appearance : X3DAppearanceNode {
  SFNode [in,out] fillProperties NULL [FillProperties]
  SFNode [in,out] lineProperties NULL [LineProperties]
  SFNode [in,out] material NULL [X3DMaterialNode]
  SFNode [in,out] metadata NULL [X3DMetadataObject]
```

```
  MFNode [in,out] shaders  []  [X3DShaderNode]
  SFNode [in,out] texture  NULL [X3DTextureNode]
  SFNode [in,out] textureTransform NULL [X3DTextureTransformNode]
}
```

Listing B-4: Appearance

```
Material : X3DMaterialNode {
  SFFloat [in,out] ambientIntensity 0.2 [0,1]
  SFColor [in,out] diffuseColor 0.8 0.8 0.8 [0,1]
  SFColor [in,out] emissiveColor 0 0 0 [0,1]
  SFNode [in,out] metadata  NULL     [X3DMetadataObject]
  SFFloat [in,out] shininess 0.2     [0,1]
  SFColor [in,out] specularColor 0 0 0 [0,1]
  SFFloat [in,out] transparency 0    [0,1]
}
```

Listing B-5: Material

```
Box : X3DGeometryNode {
  SFNode [in,out] metadata NULL [X3DMetadataObject]
  SFVec3f []   size   2 2 2 (0,8)
  SFBool []    solid  TRUE
}
```

Listing B-6: Box

```
Sphere : X3DGeometryNode {
  SFNode [in,out] metadata NULL [X3DMetadataObject]
  SFFloat []   radius 1   (0,8)
  SFBool []    solid  TRUE
}
```

Listing B-7: Sphere

```
Cone : X3DGeometryNode {
  SFNode [in,out] metadata NULL [X3DMetadataObject]
  SFBool []    bottom    TRUE
  SFFloat []   bottomRadius 1 (0,8)
  SFFloat []   height    2   (0,8)
  SFBool []    side      TRUE
  SFBool []    solid     TRUE
}
```

Listing B-8: Cone

```
Cylinder : X3DGeometryNode {
  SFNode [in,out] metadata NULL [X3DMetadataObject]
  SFBool []    bottom TRUE
```

```
  SFFloat []   height 2   (0,8)
  SFFloat []   radius 1   (0,8)
  SFBool []    side   TRUE
  SFBool []    solid  TRUE
  SFBool []    top    TRUE
}
```

Listing B-9: Cylinder

```
Text : X3DGeometryNode {
  SFNode [in,out] fontStyle NULL [X3FontStyleNode]
  MFFloat [in,out] length [] [0,8)
  SFFloat [in,out] maxExtent 0.0 [0,8)
  SFNode [in,out] metadata NULL [X3DMetadataObject]
  MFString [in,out] string []
  MFVec2f [out] lineBounds
  SFVec3f [out] origin
  SFVec2f [out] textBounds
  SFBool []     solid   FALSE
}
```

Listing B-10: Text

```
FontStyle : X3DFontStyleNode {
  SFNode [in,out] metadata NULL [X3DMetadataObject]
  MFString []   family   "SERIF"
  SFBool []     horizontal TRUE
  MFString []   justify  "BEGIN" ["BEGIN","END","FIRST","MIDDLE",""]
  SFString []   language ""
  SFBool []     leftToRight TRUE
  SFFloat []    size     1.0   (0,8)
  SFFloat []    spacing  1.0   [0,8)
  SFString []   style    "PLAIN" ["PLAIN"|"BOLD"|"ITALIC"|"BOLDITALIC"|"
     "]
  SFBool []     topToBottom TRUE
}
```

Listing B-11: FontStyle

```
Background : X3DBackgroundNode {
  SFBool [in]   set_bind
  MFFloat [in,out] groundAngle [] [0,p/2]
  MFColor [in,out] groundColor [] [0,1]
  MFString [in,out] backUrl [] [URI]
  MFString [in,out] bottomUrl [] [URI]
  MFString [in,out] frontUrl [] [URI]
  MFString [in,out] leftUrl [] [URI]
  SFNode [in,out] metadata NULL [X3DMetadataObject]
  MFString [in,out] rightUrl [] [URI]
  MFString [in,out] topUrl [] [URI]
```

```
  MFFloat [in,out] skyAngle [] [0,p]
  MFColor [in,out] skyColor 0 0 0 [0,1]
  SFFloat [in,out] transparency 0 [0,1]
  SFTime [out]  bindTime
  SFBool [out]  isBound
}
```

Listing B-12: Background

Die Forschungsberichte des Instituts für Wirtschaftsinformatik (IWi) der Universität Leipzig erscheinen in unregelmäßiger Reihenfolge.

Ein Heft kostet 15 Euro, Erscheinungsort ist immer Leipzig.

Bisher in dieser Reihe veröffentlichte Forschungsberichte:

Heft 1: Hrach, C.; Alt, R.: Einsatz von Business Intelligence-Technologien in Call Centern, 2008.
Heft 2: Schmelich, V.; Alt, R.: Functional Analysis of Open Source ERP Systems - An Exploratory Analysis, 2008.
Heft 3: Alt, R.; Eisenecker, U.; Franczyk, B.: 9. Interuniversitäres Doktorandenseminar Wirtschaftsinformatik der Universitäten Halle-Wittenberg, Jena und Leipzig, 2008.

UNIVERSITÄT LEIPZIG

Impressum: Prof. Dr. Rainer Alt,
Prof. Dr. Ulrich Eisenecker,
Prof. Dr. Bogdan Franczyk

Grimmaische Straße 12
D-04109 Leipzig
www.iwi.uni-leipzig.de

ISSN : 1865-3189