Universität Leipzig
Wirtschaftswissenschaftliche Fakultät
Institut für Wirtschaftsinformatik
Professur für Wirtschaftsinformatik, insb. Softwareentwicklung für Wirtschaft und Verwaltung

Diplomarbeit

Freie wissenschaftliche Arbeit
zur Erlangung des akademischen Grades Diplom-Wirtschaftsinformatiker
an der Wirtschaftswissenschaftlichen Fakultät der Universität Leipzig

Statische Codemetriken als Bestandteil dreidimensionaler Softwarevisualisierungen

Betreuender Hochschullehrer: Prof. Dr. U.W. Eisenecker
Bearbeiter: Jan Schilbach

Matr.-Nr.: 9517190
10. Semester

Eingereicht am: 23.02.2010

Abstract

Statische Codemetriken sind wichtige Indikatoren für die Qualität eines Softwaresystems. Sie beleuchten dabei unterschiedliche Aspekte eines Softwaresystems. Deshalb ist es notwendig, mehrere Codemetriken zu nutzen, um die Qualität eines Softwaresystems in seiner Gesamtheit bewerten zu können. Wünschenswert wäre zudem eine Darstellung, die die Struktur des Gesamtsystems und die Bewertung einzelner Elemente eines Softwaresystems in einer Darstellung kombiniert. Die Arbeit untersucht deshalb, welche Metaphern geeignet sind, um eine solche Darstellung zu ermöglichen. Ein zweites Ziel der Arbeit war es, eine solche Visualisierung automatisch erzeugen zu können. Dafür wurde ein Generator entwickelt, der diese Anforderung erfüllt. Zur Konzeption dieses Generators kamen Techniken aus der generativen Softwareentwicklung zum Einsatz. Bei der Umsetzung des Generators wurde auf Techniken aus der modellgetriebenen Softwareentwicklung zurückgegriffen, vor allem auf Techniken aus dem openArchitectureWare-Framework. Der Generator kann in Eclipse eingebunden werden und ist in der Lage, aus einem Java-Projekt die Struktur und die Metrikwerte automatisch zu extrahieren. Diese Werte werden daraufhin in ein dreidimensionales Modell überführt, das auf dem offenen Extensible 3D Standard basiert. Der Generator ermöglichte zudem die Evaluierung zweier unterschiedlicher Metaphern, die im Rahmen der Arbeit durchgeführt wurde.

Schlüsselwörter

Softwarevisualisierung, Softwaremetrie, Metrikvisualisierung, Visualisierungsmetapher, Extensible 3D, openArchitectureWare, generative Softwareentwicklung, modellgetriebene Softwareentwicklung

Inhaltsverzeichnis

Abbildungsverzeichnis IV

Tabellenverzeichnis V

Verzeichnis der Listings VI

Abkürzungsverzeichnis VII

1 Einleitung 1

1.1 Problemstellung und Motivation . . . 1
1.2 Zielstellung . . . 1
1.3 Aufbau der Arbeit . . . 2

2 Metriken 4

2.1 Definition . . . 4
2.2 Skalen . . . 4
2.3 Qualität von Metriken . . . 5
2.4 Softwaremetriken . . . 5
2.4.1 Softwarequalität . . . 6
2.4.2 Goal-Question-Metric . . . 7
2.5 Statische Codemetriken . . . 7
2.5.1 Imperative Metriken . . . 8
2.5.2 Objektorientierte Metriken . . . 10
2.5.3 Visualisierungen von Metriken . . . 12

3 Softwarevisualisierung 15

3.1 Definition . . . 15
3.2 Einsatzbereiche und Kategorisierung . . . 15
3.3 Vorgang der Visualisierung . . . 17
3.4 Metaphern . . . 18
3.4.1 Definition und Kategorisierung . . . 18
3.4.2 Ausgewählte dreidimensionale Metaphern . . . 19
3.5 Virtuelle Umgebungen . . . 27
3.6 Extensible 3D als Basis . . . 28
3.6.1 X3D-Architektur . . . 29
3.6.2 X3D-Spezifikation und Profilkonzept . . . 30
3.6.3 X3D-Browser . . . 31

4 Modellgetriebene Softwareentwicklung 33

4.1 Definition . . . 33
4.2 Domänenarchitektur . . . 34
4.2.1 Metamodelle . . . 35
4.2.2 Metametamodelle . . . 35
4.2.3 Domänenspezifische Sprachen . . . 36
4.2.4 Modelltransformationen . . . 37

5 Generative Programmierung 40

5.1 Definition . . . 40
5.2 Generatives Domänenmodell . . . 41
5.2.1 Problemraum . . . 41
5.2.2 Konfigurationswissen . . . 42
5.2.3 Lösungsraum . . . 42
5.3 Technikprojektion . . . 42
5.4 Prozessmodell . . . 42
5.4.1 Domänenentwicklung . . . 43
5.4.2 Anwendungsentwicklung . . . 43
5.5 Verhältnis zur modellgetriebenenen Softwareentwicklung . . . 44
5.6 Adaption an die Softwarevisualisierung . . . 45

6 Prototypentwicklung 47

6.1 Aufbau des Kapitels . . . 47
6.2 Ziele des Prototyps . . . 47
6.3 Verwendete Werkzeuge . . . 48
6.3.1 Eclipse . . . 48
6.3.2 openArchitectureWare . . . 51
6.4 Erweiterung der Visualisierungspipeline . . . 54
6.4.1 Modifikation des Visualisierungsalgorithmus . . . 55
6.4.2 Extraktion von Strukturinformationen . . . 56
6.4.3 Metrikextraktion und Merging mit Strukturinformationen . . . 57
6.5 Metamodelle . . . 60
6.5.1 Abstrakte Java-Syntax . . . 60
6.5.2 XML-Schema . . . 60
6.5.3 Ecore . . . 64
6.6 Evaluierung von Metaphern . . . 65
6.6.1 Schachtelungsmetapher . . . 65
6.6.2 Abstrakte Weltraummetapher . . . 67
6.7 Entwicklung der domänenspezifischen Sprachen . . . 69
6.7.1 Textuelle DSL . . . 70
6.7.2 Dialogbasierte DSL . . . 71
6.8 Erstellung des Plugins . . . 73

7 Fazit und Ausblick **75**

Literaturverzeichnis **79**

Abbildungsverzeichnis

2.1 Streudiagramme 13
2.2 Kiviatdiagramm 14
2.3 Dreidimensionale *treemap* 14

3.1 Dreidimensionale Bäume 20
3.2 Schachtelungsmetapher 21
3.3 UML in 3D 22
3.4 Hemisphärenmetapher 23
3.5 Architekturvisualisierung mittels Stadtmetapher 24
3.6 Sonnensystemmetapher 25
3.7 Imsovision 26
3.8 X3D-Architektur 29

4.1 Hierarchie der Metamodellierung 35
4.2 Modelltransformationen 38

5.1 Generatives Domänenmodell 41
5.2 Prozesse der generativen Entwicklung 43
5.3 Generatives Domänenmodell zur Verbindung von Struktur und Metrikvisualisierung 45
5.4 Prozessmodell zur Softwarevisualisierung in 3D 46

6.1 Visualisierungspipeline von Müller (2009) 55
6.2 Erweiterte Visualisierungspipeline 55
6.3 Visualisierung mit der Schachtelungsmetapher 67
6.4 Visualisierung mit der abstrakten Weltraummetapher 68
6.5 Dialogbasierte DSL 72

Tabellenverzeichnis

3.1 Bewertungen der Möglichkeiten zur Metrikvisualisierung. +: gut nutzbar; ∘: eingeschränkt nutzbar; −: kaum nutzbar 27

6.1 Verwendete *Java Model*-Schnittstellen 49
6.2 Vom *Metrics*-Plugin bestimmbare Metriken 50
6.3 Verwendete Workflowkomponenten 53
6.4 Verwendete X3D-Elemente 64
6.5 Variablen der textuellen DSL 71

Verzeichnis der Listings

6.1 Aufruf einer Komponente 52
6.2 XTend-Modelltransformation zum Zuordnen der gewählten Metriken zum jeweiligen Paket 54
6.3 Strukturextraktion mit Java 57
6.4 Auslesen einer Eigenschaft 70
6.5 Auszug aus einer Konfigurationsdatei 70
6.6 Erweitern des Kontextmenüs 73
6.7 Aufruf eines Workflow mittels Java 74

Abkürzungsverzeichnis

DIT .... Depth of Inheritance Tree
DSL .... Domain specific Language
DTD .... Document Type Definition
EMF .... Eclipse Modelling Framework
EMF .... Eclipse Modelling Framework
EMOF .... Essential Meta-Object Facility
FODA .... Feature Oriented Domain Analysis
GQM .... Goal Question Metrik
ISO .... International Standardization Organisation
JavaML .... Java Markup Language
LCOM .... Lack of Cohesion of Methods
LOC .... Lines of Code
MDA .... Model Driven Architecture
MDSD .... Model Driven Software Development
MOF .... Meta Object Facility
NOA .... Number of Attributes
NOF .... Number of Fields
NOM .... Number of Methods
NORM .... Number of Overridden Methods
oAW .... openArchitectureWare
OMG .... Object Modelling Group
PIM .... Plattform Independent Model
PSM .... Platform Specific Model
QVT .... Query View Transformation
UML .... Unified Modelling Language
VRML .... Virtal Reality Markup Language
VRML .... Virtual Reality Markup Language
WMC .... Weighted Methods per Class
X3D .... Extensible 3D
XMI .... XML Metadata Interchange
XML .... Extensible Markup Language
XSL .... Extensible Stylesheet Language

1 Einleitung

1.1 Problemstellung und Motivation

Größere Softwaresysteme werden mittlerweile immer häufiger im Rahmen eines modellgetriebenen Softwareentwicklungsprozesses entwickelt, da dieser einen deutlich höheren Automatisierungsgrad beim Erstellen solcher Systeme mit sich bringt. Diese höhere Automatisierung führt zu einer höheren Produktivität, einer höheren Codequalität durch die Vermeidung von Flüchtigkeitsfehlern und zu einer leichten Änderbarkeit des Softwaresystems.

Einige der Modelle, die während eines modellgetriebenen Entwicklungsprozesses erstellt und verwendet werden, lassen sich sehr gut im Rahmen der Softwarevisualisierung einsetzen. Des Weiteren fallen in einem Softwareentwicklungsprozess auch weitere Informationen, wie zum Beispiel Codemetriken oder Statistiken, über das Produkt an, bei denen es wünschenswert wäre, diese auf geeignete Weise visualisieren zu können. Visualisierungen dieser Informationen können dabei helfen die Komplexität der Software selbst, als auch die Komplexität des Softwareentwicklungsprozesses besser beherrschbar zu machen.

Im Bereich der modellgetriebenen Softwareentwicklung (engl. model driven software development, MDSD) sind bereits einige grafische Modelle vorhanden, die zur Darstellung verschiedener Sichten auf Software verwendet werden. Zumeist sind diese Darstellungen Diagramme der Unified Modelling Language oder Diagramme, die auf Ecore, dem Metametamodell des Eclipse Modelling Frameworks, aufbauen. Diese Diagrammtypen bieten leider nur wenig Möglichkeiten zur Anreicherung mit Zusatzinformationen. Aus diesem Grund werden Zusatzinformationen oft von den Strukturinformationen entkoppelt dargestellt. Es wäre aber wünschenswert, wenn diese Informationen direkten Einfluss auf die Visualisierungen des Softwaresystems an sich nehmen würden. Die vorliegende Arbeit untersucht daher, wie ein Softwaremodell mit Informationen so angereichert werden kann, dass Interessensbeteiligte für sie wichtige beziehungsweise kritische Elemente und Strukturen eines Softwaresystems in geeigneter Weise überblicken können.

Im Rahmen dieser Arbeit sollen statische Codemetriken exemplarisch für eine Reihe von Zusatzinformationen stehen, die bei einer solchen Visualisierung dargestellt werden können. Der vorwiegende Grund für die Verwendung von statischen Codemetriken ist, dass sie durch geeignete Werkzeuge relativ einfach automatisch erzeugt werden können. Dadurch soll es möglich werden, den gesamten Prozess der Softwarevisualisierung und die Anreicherung mit Zusatzinformationen vollautomatisch ablaufen zu lassen.

1.2 Zielstellung

Die Arbeit soll aufzeigen, wie statische Codemetriken aufgebaut sein müssen, damit sie für die Zwecke der Visualisierung gut geeignet sind. Zudem sollen die unterschiedlichen bereits vorhandenen Visualisierungen im Rahmen der Darstellung von Codemetriken und Software-

systemen dargestellt und diskutiert werden. Besonderes Augenmerk fällt hierbei den dreidimensionalen Visualisierungen zu, da auf diese Weise einige Probleme mit zweidimensionalen Darstellungen gelöst werden können und der Nutzer wertvolle Interaktionsmöglichkeiten an die Hand bekommt, mit denen er eine solche Visualisierung besser untersuchen und begreifen kann. Durch die räumliche Darstellung soll es zudem möglich werden, mehr Informationen so darzustellen, dass keine Überforderung des Nutzers entsteht, da der Mensch es gewöhnt ist, sich in einem dreidimensionalen Raum zu bewegen und zu orientieren. Außerdem bietet die dreidimensionale Darstellung völlig neue Freiheiten bei der Entwicklung geeigneter Darstellungsmetaphern, die im zweidimensionalen Raum so nicht vorhanden sind.

Um die Anreicherungsmöglichkeiten einer solchen Visualisierung darzustellen und zu testen, soll ein Prototyp erstellt werden, auf dessen Entwicklung im Rahmen der Arbeit näher eingegangen wird. Die Grundlage des Prototyps soll dabei der von Müller (2009) beschriebene Generator zur automatischen Visualisierung eines Softwaresystems bilden. Im Rahmen der Entwicklung des Prototyps werden zwei unterschiedliche Metaphern näher betrachtet, die für die Zwecke der Arbeit jeweils spezifische Vor- und Nachteile bieten, die Schachtelmetapher und die Weltraummetapher. Die Schachtelmetapher wurde von Müller (2009) bereits im Rahmen einer vollautomatischen Visualisierung genutzt. Der Einsatz einer Weltraummetapher für die Zwecke der Metrikvisualisierung wurde von Graham et al. (2004) im Rahmen des Tools „MetricVisualizer3D“ gezeigt. Der Prototyp soll diesen Ansatz weiterentwickeln. Zum einen werden mehrere, vom Nutzer wählbare Metriken gleichzeitig in einer Darstellung angezeigt und zum anderen läuft die Erzeugung der Visualisierung vollautomatisch ab. Weiterhin sollen auch die Möglichkeiten der Konfiguration einer solchen Visualisierung dargestellt und implementiert werden, wofür zwei unterschiedliche domänenspezifische Sprachen zum Einsatz kommen. Die Entwicklung der Visualisierungspipeline, die hierfür zum Einsatz kommt, soll ebenfalls aufgezeigt werden.

1.3 Aufbau der Arbeit

Der Hauptteil der Arbeit gliedert sich in fünf Kapitel. In den Kapiteln 2–5 werden zunächst die relevanten theoretischen Grundlagen gelegt werden, die für die Arbeit relevant sind.

Zuerst wird in Kapitel 2 auf Codemetriken eingegangen. Dazu gehört zunächst eine Begriffsklärung. Des Weiteren werden in diesem Kapitel aber auch ausgesuchte Metriken und einige Visualisierungsformen vorgestellt, die aktuell zum Einsatz kommen. Hierbei wird auch auf deren Vor- und Nachteile eingegangen. Daran schließt sich Kapitel 3 an, welches sich mit dem aktuellen Stand und den Möglichkeiten der Softwarevisualisierung beschäftigt. Auch hier werden die grundlegenden Begrifflichkeiten erläutert sowie Vor- und Nachteile von ausgesuchten Visualisierungsformen geklärt.

In den Kapiteln 4 und 5 werden die zwei zentralen Paradigmen für die Entwicklung des Prototyps erläutert. Im Einzelnen sind dies das modellgetriebene Paradigma und das generative Paradigma. Das modellgetriebene Paradigma hat sich mittlerweile als ein wichtiges Paradigma der industriellen Softwareentwicklung heraus kristallisiert. Im Rahmen der

vom Generator durchgeführten Visualisierung kommt es zu einer Reihe von Modelltransformationen und Modell-Weaving-Schritten, die vor allem auf die Entwicklungen im Bereich der modellgetriebenen Softwareentwicklung zurückgehen. Das generative Paradigma liefert durch die Definitionen zu domänenspezifischen Sprachen und Domänenarchitekturen wichtige Grundlagen für diese Arbeit.

Kapitel 6 befasst sich im Anschluss daran mit der Entwicklung eines Generators, der es ermöglichen soll, diese Möglichkeiten prototypisch aufzugreifen und umzusetzen. Zu Beginn dieses Kapitels werden zunächst die einzelnen verwendeten Werkzeuge vorgestellt. Daran schließen sich die Überlegungen zur Definition eines Ausgangsformats und zur Entwicklung von zwei unterschiedlichen domänenspezifischen Sprachen zur Konfiguration der Visualisierung an. Darauf aufbauend folgen dann Überlegungen zu unterschiedlichen Metaphern und zum Aufbau einer Visualisierung, um die Vollautomatisierung der Visualisierung zu erreichen. Am Ende des Kapitels folgt dann die Beschreibung, wie das gewünschte Eclipse-Plugin erstellt und aufgebaut wurde.

Im Schlussteil der Arbeit werden dann die unterschiedlichen gewonnenen Erkenntnisse eingeordnet und diskutiert sowie ein Fazit gezogen. In diesem Kapitel werden auch unterschiedliche Möglichkeiten zur Weiterentwicklung aufgezeigt und noch vorhandene Probleme dargestellt.

2 Metriken

2.1 Definition

Metriken sind mathematisch definiert als Funktionen, die ein System aus einer Menge von Systemen auf einen Wert aus dem möglichen Ergebnisraum abbilden (vgl. Böhme und Freiling, 2008). Zum Beispiel könnte eine Metrik die Luftfeuchtigkeit aus der Menge der Lufteigenschaften auf einen Wert aus dem Ergebnisraum „trocken“ und „feucht“ abbilden. Häufig basiert diese Zuordnung auf einem bestimmten Messverfahren. Es ist allerdings auch möglich, andere Einordnungsmöglichkeiten zu finden, die zum Teil auch subjektiv sein können.

2.2 Skalen

Die Aufteilung des Ergebnisraums wird durch Skalen definiert. Hierbei gibt es unterschiedliche Skalentypen (vgl. Böhme und Freiling, 2008; Liggesmeyer, 2009):

- Nominalskala
- Ordinalskala
- Intervallskala
- Rationalskala
- Absolutskala

Die Nominalskala ist die am einfachsten aufgebaute Skalenform. Das oben genannte Beispiel mit der Luftfeuchtigkeit basiert auf einer solchen Nominalskala. Hierbei sind die Werte untereinander zwar abgegrenzt, jedoch lässt sich keine Ordnung der Werte vornehmen. Deswegen kann bei einem Wert aus einer Nominalskala auch nicht darauf geschlossen werden, ob dieser größer oder kleiner als ein anderer Wert der Skala ist.

Bei der Ordinalskala kommen geordnete Werte zum Einsatz. Ein einfaches Beispiel für eine Ordinalskala ist das klassische Notensystem in der Schule. Hierbei ist definiert, in welcher Rangfolge die einzelnen Werte angeordnet sind. Bei dieser Form der Skala ist es nicht erforderlich, dass die Abstände zwischen den einzelnen Werten gleich oder bedeutsam sind, dies ist erst bei einer Intervallskala der Fall.

Die Intervallskala ist somit eine strengere Form der Ordinalskala und wird beispielsweise bei der Messung der Temperatur genutzt. Mit ihr ist es möglich, die Abstände von Werten miteinander zu vergleichen. Während bei der Notenskala in der Schule nicht davon ausgegangen werden kann, dass der Unterschied von der Note 4 zur Note 6 der gleiche ist wie von der Note 1 zur Note 3, kann bei der Intervallskala davon augegangen werden, dass der Temperaturunterschied zwischen 10°C und 20°C der gleiche ist wie zwischen 20°C und 30°C.

Die Rationalskala ist eine Spezialisierung der Intervallskala. Zusätzlich zu den Eigenschaften der Intervallskala besitzt die Verhältnisskala einen natürlich gegebenen Nullpunkt. Dies ist zum Beispiel beim Messen der Stromstärke oder bei Längenangaben der Fall.

Die Absolutskala stellt wiederum eine Spezialisierung der Rationalskala dar, die sich zusätzlich dadurch auszeichnet, dass auch ihre Maßeinheit natürlich gegeben ist und nicht frei gewählt werden kann. Es handelt sich hierbei meist um Zählungen, zum Beispiel von Personen, Gegenständen o. ä. (vgl. Liggesmeyer, 2009).

Die Intervall- und die Rationsalskala werden zum Teil auch unter dem Begriff Kardinalskala eingeordnet.

2.3 Qualität von Metriken

Liggesmeyer (2009) fordert für Metriken, dass sie sechs Voraussetzungen erfüllen müssen, da sie ansonsten nur sehr bedingt zum Einsatz kommen können oder sollten. Als erste Voraussetzung nennt Liggesmeyer (2009) ein möglichst einfaches Ergebnis der Messung, um eine klare und eindeutige Verständlichkeit zu ermöglichen. Dies dient dazu, schnell zu erkennen, ob ein Ergebnis für die jeweiligen Bedürfnisse gut oder schlecht ist. Die zweite Forderung ist die nach Validität. Eine Metrik soll das abbilden, was gemessen werden soll. Stabilität bildet die dritte Forderung. Dadurch soll sichergestellt werden, dass sich die Messwerte nicht aufgrund unbedeutender Einflüsse stark ändern. Eine weitere von Liggesmeyer (2009) geforderte Eigenschaft ist die Möglichkeit zur rechtzeitigen Messung, da man nur dann etwas messen muss, wenn man noch die Möglichkeiten besitzt, steuernd einzugreifen. Die letzten beiden Forderungen sind die nach der Analysierbarkeit und der Reproduzierbarkeit der Metrik. Es ist offensichtlich, dass eine Metrik, die vom Zufall oder dem Messverfahren beeinflusst wird, keine wirklich aussagekräftigen Werte liefern kann. Für die Analysierbarkeit der Metriken ist es von großer Bedeutung, dass sie auf Kardinalskalen oder Absolutskalen arbeiten, damit sie einer statistischen Auswertung zugänglich sind. Auch für die Zwecke dieser Arbeit ist es wichtig, dass die verwendeten Metriken auf einer der beiden genannten Skalentypen arbeiten. Andernsfalls würden die beeinflussten Parameter die dargestellten Metriken nicht korrekt widerspiegeln, weil die Messwerte im Rahmen des Prototyps auf Prozentwerte abgebildet werden.

2.4 Softwaremetriken

Softwaremetriken sind spezielle Metriken, die eine Bewertung von Software im Ganzen und von einzelnen Aspekten ermöglichen, indem sie ausgewählte Eigenschaften von Software quantifizieren (vgl. Liggesmeyer, 2009). Da Software mittlerweile in fast allen Bereichen des Lebens Einzug gehalten hat, auch in kritische, wird ihre Messung um so wichtiger, weil sie hilft, das Design von Software beherrschbarer zu machen und kritische Bereiche von Software hervorzuheben.

Softwaremetriken haben als einheitliches Grobziel, die Güte von Software messen zu können. Eine solche Messung ist Voraussetzung um die Wirkung von vorgenommenen Maßnahmen beurteilen zu können, da nur so eine gewünschte Verbesserung objektiv erkennbar ist. Es ergeben sich hierbei unterschiedliche Perspektiven, wie diese Qualität aussieht. Die

unterschiedlichen Aspekte der Softwarequalität sollen im folgenden Absatz näher beleuchtet werden.

2.4.1 Softwarequalität

Hoffmann (2008) nennt zur Betrachtung von Softwarequalität acht unterschiedliche Aspekte:

- Korrektheit
- Zuverlässigkeit
- Laufzeitverhalten
- Benutzbarkeit
- Änderbarkeit
- Übertragbarkeit
- Testbarkeit
- Transparenz

Bei Ghezzi et al. (2002) sind zusätzlich die zwei weiteren Qualitäten Produktivität und Termintreue definiert, die für den Softwareentwicklungsprozess von Bedeutung sind. Zudem sind bei Ghezzi et al. (2002) die einzelnen Qualitäten noch stärker differenziert. Beispielsweise wird die Robustheit eines Softwaresystems von den Autoren von der Zuverlässigkeit getrennt betrachtet. Robustheit stellt bei Ghezzi et al. (2002) die Fähigkeit des Systems dar, von einem instabilen Zustand wieder in einen stabilen Zustand zurückzukehren, also auch bei Fehleingaben oder Hardwaredefekten nicht abzustürzen. Zuverlässigkeit hingegen ist definiert als die Wahrscheinlichkeit, mit der ein Programm über einen definierten Zeitraum erwartungsgemäß funktioniert (vgl. Musa, 1987; Ghezzi et al., 2002).

Die einzelnen Personengruppen, die an der Entwicklung und dem Betrieb von Software beteiligt sind, haben aber unterschiedliche Interessenlagen. Dies führt dazu, dass für die einzelnen Beteiligten jeweils unterschiedliche Softwarequalitäten von Bedeutung sind. Für den Anwender stellt sich die Frage, ob die Software korrekt arbeitet, wie schnell er mit dem System arbeiten kann, wie oft er aufgrund von Fehlern seine Arbeit unterbrechen muss usw. Eine andere Möglichkeit ist, die Softwarequalität aus der Sicht der Wartung eines Softwaresystems zu betrachten. Die hier auftretenden Fragen sind zum Beispiel, wie komplex die einzelnen Funktionen sind, wie gut Teile wiederverwendet werden können oder wo sich Code befindet, der in bestimmten Situationen nicht funktioniert. Oft können und sollen beide Perspektiven auf die Softwarequalität nicht voneinander getrennt betrachtet werden, da sie sich gegenseitig beeinflussen. Zum Beispiel kann man davon ausgehen, dass ein Entwickler in einem Softwaresystem, dessen Bestandteile gut wartbar sind, einen Fehler schneller beheben kann als in einem System, dessen Bestandteile so komplex sind, dass dieser Fehler sehr schlecht erkannt werden kann. Es gibt allerdings auch Softwarequalitäten, die sich gegenseitig behindern. Zum Beispiel muss ein hoher Grad der Zuverlässigkeit beziehungsweise Robustheit meist mit Einschränkungen in der Performanz erkauft werden.

2.4.2 Goal-Question-Metric

Der Goal-Question-Metric-Ansatz (GQM) stellt eine systematische Vorgehensweise dar, um die Definition oder Spezifizierung von geeigneten Metriken zu ermöglichen. Ursprünglich wurde dieser Ansatz von Basili und Weiss (1984) für den Bereich der Softwareentwicklung entwickelt, mittlerweile findet er aber auch in anderen Bereichen Verwendung.

Da es sich im Laufe der Zeit gezeigt hat, dass die Ableitung von möglichen Verwendungszwecken aus den erstellten Metriken meist nicht zielführend ist, stellt der GQM-Ansatz die Definition der Messziele an den Beginn. Die Entwicklung beziehungsweise Auswahl von Metriken wird hier also top-down vorgenommen (vgl. Koziolek, 2008).

Van Solingen und Berghout (1999) teilen das Vorgehen hierbei in vier verschiedene Phasen auf:

- Planungsphase
- Definitionsphase
- Datensammlungsphase
- Interpretationsphase

Die Planungsphase dient zur Planung der einzelnen notwendigen Tätigkeiten. Daran schließt sich die Definitionsphase an, in der zuerst bestimmt wird, was das Ziel der Messung ist (Goal). Aus diesem Ziel werden Fragen abgeleitet, die gestellt werden sollen, um dieses Ziel zu erreichen (Question) und passend zu diesen Fragen werden daraufhin Metriken entwickelt oder gesucht, deren Werte in der Lage sind, die Fragen zu beantworten und somit das Ziel zu erreichen.

In der Datensammlungsphase werden die Daten der Metriken erhoben und in der Interpretationsphase ausgewertet, sodass die gestellten Fragen beantwortet und die gestellten Ziele erreicht werden können.

2.5 Statische Codemetriken

Statische Codemetriken sind Metriken, die sich mit der Analyse des Quelltextes eines Programms befassen. Infolgedessen können sie mögliche kritische Punkte im Quelltext, aber nicht unbedingt im Laufzeitverhalten eines Programms quantifizieren und so für den Nutzer sichtbar machen. Durch die Möglichkeiten, den Quellcode automatisch parsen zu können, werden statische Codemetriken mittlerweile meist vollständig automatisiert erzeugt. Dies ist ein Punkt, der sie für die Verwendung im Rahmen dieser Arbeit so interessant macht. Auf diese Weise wird die Visualisierung eines mit Codemetriken angereicherten Softwaresystems möglich, ohne dass der Generierungsprozess noch manuell durchzuführende Schritte beeinhaltet. Aktuell ist die wohl verbreitetste Form der Darstellung von Codemetriken die Tabellenform, welche im Grunde denkbar ungeeignet ist, da hier sowohl Korrelationen zwischen mehreren Metriken, als auch die Struktur des Softwaresystems ausgeblendet werden. Zusätzlich sind durch die Vielzahl der vorhandenen Metriken die Tabellen schnell sehr

unübersichtlich und für den Betrachter ist das Finden von kritischen Stellen des Softwaresystems nur mit erheblichem Aufwand möglich. Drei alternative Darstellungsformen für Metriken, die bei Hoffmann (2008) aufgeführt sind, werden in Abschnitt 2.5.3 erläutert. Diese Darstellungsformen können zum Teil von Tools erzeugt werden, die die Metriken automatisiert ermitteln. In den folgenden beiden Abschnitten 2.5.1 und 2.5.2 werden unterschiedliche Metriken betrachtet und kategorisiert.

2.5.1 Imperative Metriken

Die imperativen Metriken stellen die ersten Metriken dar, die im Rahmen der Softwareentwicklung genutzt wurden, da das imperative Paradigma das älteste Paradigma der Softwareentwicklung darstellt. Bei einem vollständig imperativ implementierten Programm beziehen sich die Metriken immer auf das Gesamtprogramm. Dies ist für die Zwecke dieser Arbeit aber nicht zielführend, weil im Rahmen dieser Arbeit die kritischen Punkte eines objektorientierten Softwaresystems dreidimensional hervorgehoben werden sollen und keine explizite Bewertung des Gesamtsystems durchgeführt werden soll. Da im Bereich der objektorientierten Softwareentwicklung die einzelnen Methoden durchaus imperativ entwickelt werden, wurden die nachfolgend genannten Metriken im Laufe der Zeit auch für objektorientierte Programme angepasst, sodass sie ebenfalls für die Zwecke dieser Arbeit genutzt werden können.

2.5.1.1 Kategorisierung

Codemetriken können in unterschiedliche Kategorien eingeteilt werden. Laird und Brennan (2006) nehmen eine Einteilung in Metriken vor, die zum einen die Größe und zum anderen die Komplexität von Software messen. Diese Kategorisierung ist recht grob und nicht sehr trennscharf, da zum Beispiel auch Größenmetriken, wie die Anzahl der Codezeilen einen Anhaltspunkt für die Komplexität eines Softwaresystems liefern können. Es gibt jedoch für die Bewertung der Komplexität deutlich geeignetere Maße, weswegen die Einteilung in Größen- und Komplexitätsmaße hier beibehalten werden soll.

2.5.1.2 Ausgesuchte imperative Metriken

Die wohl bekannteste Codemetrik stellt die *Lines-of-Code*-Metrik (LOC) dar. Sie ist wahrscheinlich die am einfachsten zu erzeugende Metrik, da nur die Anzahl der Codezeilen aller Quellcodedateien aufsummiert wird. Sie ist auf alle textuell dargestellten Programmiersprachen anwendbar (vgl. Hoffmann, 2008).

Die LOC-Metrik spiegelt die Größe des Programms wider. Sie wird jedoch auch als Indikator für die Komplexität eines Computerprogramms gesehen, da man davon ausgeht, dass größere Programme auch eine größere Komplexität aufweisen.

Die Metrik ist sehr leicht erzeugbar. Als häufigste Kritik ist zu beachten, dass es bei unterschiedlichen Programmierstilen zu stark auseinander driftenden Werten kommen kann,

etwa wenn ein Java-Programmierer sein Programm ausführlich kommentiert und geschweifte Klammern immer auf eine neue Zeile setzt und ein anderer nicht. Um diese Probleme zu lösen, wurden im Laufe der Zeit unterschiedliche Lösungen vorgeschlagen. Vorgeschlagene Lösungen sind zum Beispiel das Zählen von unkommentierten Zeilen, das Zählen der Anweisungen anstatt Zeilen oder die Nutzung eines automatischen Quellcodeformatierers um eine einheitliche Form für die Messung zu gewährleisten. Grundsätzlich sind natürlich alle Erhebungsformen möglich, es sollte nur darauf geachtet werden, dass die Erhebungsform konstant ist, also immer gleich gemessen wird (vgl. Laird und Brennan, 2006).

Eine weitere Gruppe von imperativen Metriken stellen die Halstead-Metriken dar. Sie wurden von Maurice Howard Halstead in den 70er Jahren entwickelt und dienen der Messung von unterschiedlichen Aspekten imperativer Systeme (vgl. Halstead, 1977). Die Halstead-Maße basieren auf dem Zählen lexikalischer Elemente, wie Operatoren und Operanden, im Quelltext. Halstead konnte mit den so erzeugten Werten theoretische Berechnungen zur Komplexität und Größe eines Softwaresystems durchführen, deren Validität in späteren empirischen Untersuchungen nachgewiesen werden konnte (Liggesmeyer, 2009). Die grundlegenden Maße der Halstead-Metriken sind die Anzahl unterschiedlicher Operatoren (η_1) und unterschiedlicher Operanden (η_2) sowie die Gesamtzahl der Vorkommen aller Operatoren (N_1) und Operanden (N_2). Aus diesen grundlegenden Maßzahlen lassen sich nun eine Reihe abgeleiteter Maße berechnen (Hoffmann, 2008):

$$\eta = \eta_1 + \eta_2 , \tag{2.1}$$

$$N = N_1 + N_2 , \tag{2.2}$$

$$V = N \times \log_2 \eta . \tag{2.3}$$

η stellt die Größe des Vokabulars dar, sie dient der groben Einschätzung der Komplexität eines Programms. N stellt die Länge des Programms dar. Gegenüber der *Lines of Code*-Metrik hat sie den Vorteil, dass die so gebildete Kenngröße völlig unabhängig von der Formatierung der Quelltexte arbeitet und somit als Größenmetrik besser geeignet ist. V ist die Größe für das Volumen des Programms. Die so gewonnene Größe geht davon aus, dass für alle Operatoren und Operanden eine gleich große Zahl von Bits notwendig ist und gibt die Größe der Implementierung in Bits an. Für eine ausführliche Darstellung der Halstead-Maße und ihrer Berechnung sei an dieser Stelle auf Hoffmann (2008) verwiesen.

Eine weitere sehr bekannte Kenngröße für die Komplexität stellt die zyklomatische Zahl dar, die Thomas J. McCabe 1976 zur Messung von Komplexität vorschlug. Die Berechnung dieser Maßzahl beruht auf der Annahme, dass ein imperatives Computerprogramm für einen Menschen umso komplexer ist, je mehr Möglichkeiten es gibt, den Kontrollfluss eines Programms zu durchlaufen. McCabe (1976) definiert die zyklomatische Zahl als

$$v = e - n + 2p , \tag{2.4}$$

wobei e die Anzahl der Kanten und n die Anzahl der Knoten im gesamten Graphen darstellt. p ist in dieser Gleichung die Anzahl der einzelnen Kontrollflussgraphen, d. h. Funktionen be-

ziehungsweise Prozeduren. Ein Beispiel für die Berechnung der zyklomatischen Zahl einer Funktion findet sich in Hoffmann (2008).

McCabe (1976) empfiehlt den Versuch, einen Wert zu erreichen, der unterhalb von 10 liegt, da ein Programm oder eine Methode mit einem solchen Wert für einen normalen Menschen noch als gut verständlich angesehen werden kann. Dieser Wert ist nicht als in Stein gemeißelt zu betrachten, sondern dient lediglich als Richtwert. Wird der Wert von 10 allerdings in Methoden notorisch überschritten, sollte geprüft werden, ob es wirklich keine Refaktorisierungsmöglichkeiten gibt.

Die McCabe-Metrik liefert einen guten Anhaltspunkt für die Komplexität einer Methode. Es gibt aber auch Kritik an der McCabe-Metrik, da zum Beispiel durch `switch`-Anweisungen sehr schnell hohe Werte erreicht werden, während die subjektive Komplexität hingegen noch gering sein kann.

2.5.2 Objektorientierte Metriken

Im Bereich der objektorientierten Programmierung reichen die klassischen imperativen Metriken zur Beurteilung der Qualität eines Gesamtsystems nicht mehr aus. Durch die Einführung der neuen Konzepte, Kapselung und Vererbung, wird zwar Komplexität beherrschbar gemacht, es kann jedoch nicht davon ausgegangen werden, dass nur aufgrund der Anwendung dieser Konzepte die Komplexität großer Softwaresysteme tatsächlich auch beherrscht wird. Es ist deswegen im Bereich der objektorientierten Programmierung nötig und wünschenswert, Metriken einzusetzen, die die Aspekte eines Softwaresystems beleuchten, die durch Kapselung und Vererbung entstehen.

2.5.2.1 Kategorisierung

Hoffmann (2008) unterteilt Codemetriken in zwei große Kategorien. Zum einen sind dies die Komponentenmetriken, die jeweils für eine Komponente des Softwaresystems gelten. Zum anderen sind dies die Strukturmetriken, die für das objektorientierte Softwaresystem als Ganzes bestimmt werden und Aufschluss darüber geben sollen, ob die aktuell gewählte Strukturierung des Systems in Klassen sinnvoll ist. Bei den Strukturmetriken führt Hoffmann (2008) die Fan-In- und Fan-Out-Metriken als wichtige Begriffe an, wobei sie bei eigenständiger Betrachtung dieser Werte auch in den Bereich der Komponentenmetriken eingeordnet werden können. Nichtsdestotrotz leisten die von diesen beiden Metriken abgeleiteten Werte für das Gesamtsystem einen wichtigen Beitrag für die Bewertung der Struktur von Software.

Für die Zwecke der vorliegenden Arbeit sind vor allem die Komponentenmetriken von Bedeutung, da durch ihre Visualisierung einzelne Elemente zum besseren Erkennen von Problemstellen aus dem System hervorgehoben werden können. Die Komponentenmetriken können in drei unterschiedliche Kategorien eingeteilt werden (vgl. Hoffmann, 2008). Zunächst sind dies Umfangsmetriken, die in etwa mit den klassischen imperativen Größenmetriken vergleichbar sind und einen Hinweis auf die Größe eines Softwaresystems unter Berücksichtigung der objektorientierten Konzepte geben sollen. Die zweite Kategorie stellen die Vererbungmetriken dar, die sich an einer Bewertung der Güte und Komplexität der Verer-

bung versuchen. Die letzte der drei Kategorien stellen die Kohäsionsmetriken dar. Grundsätzlich sollte in der Programmierung versucht werden, eine möglichst geringe Kopplung, aber eine möglichst hohe Kohäsion der einzelnen Module zu erreichen. Dies gilt auch für die objektorientiere Programmierung. Aufgabe der Kohäsionsmetriken ist es nun, den Grad dieser Kohäsion zu messen. Dadurch soll der Nutzer in die Lage versetzt werden, die Güte des Aufbaus eines Softwaresystems bewerten zu können. Im folgenden Abschnitt sollen ausgesuchte Vertreter dieser Kategorien näher betrachtet werden. Für den im Rahmen dieser Arbeit entwickelten Prototyp kommt zur Generierung der Metriken für die einzelnen Komponenten das Tool „Metrics" für Eclipse in Version 1.3.6 zum Einsatz. Informationen zu den Fähigkeiten des Tools und zu den erzeugten Metriken finden sich unter Sauer (2009).

2.5.2.2 Ausgesuchte Komponentenmetriken

Gebräuchliche Umfangsmetriken sind im Bereich von Java unter anderem die NOF-, NOM- und WMC-Metrik (vgl. Hoffmann, 2008). NOF steht hierbei für *Number of Fields*. Die Metrik gibt also für eine Klasse die Anzahl der Attribute an, die eine Klasse definiert, weswegen diese Metrik zum Teil auch als NOA-Metrik abgekürzt wird. NOM ist die Abkürzung für *Number of Methods*, also die Anzahl der Methoden einer Klasse. Die Metrik *Weighted Methods per Class* (WMC) gibt für jede Klasse die Summe der Einzelkomplexitäten ihrer Methoden an. Die Komplexität der einzelnen Methoden wird dabei normalerweise mit Hilfe der McCabe-Metrik berechnet.

Beispiele für Vererbungsmetriken sind die *Number of Overridden Methods* (NORM) Metrik und die *Depth of Inheritance Tree* (DIT) Metrik. Die NORM-Metrik zählt die in einer Klasse überschriebenen Methoden. Hierbei gibt es unterschiedliche Zählweisen, sodass aufgerufene abstrakte Methoden oder Methoden, die die überschriebene Methode (in Java durch das Schlüsselwort `super`) aufrufen, entweder mitgezählt werden oder nicht. Es ist zu beachten, dass beim Vergleichen von unterschiedlichen Werten für diese Metrik die gleiche Zählweise zum Einsatz kommt. Die DIT-Metrik gibt in Java für eine Klasse den Abstand zur Klasse `Object` an, zeigt also, wie tief innerhalb einer Vererbungshierarchie sich die jeweilige Klasse befindet.

Ein Beispiel für Kohäsionsmetriken ist die LCOM-Metrik (*Lack of Cohesion of Methods*), die ermöglichen soll, den Mangel an Methodenkohäsion zu bewerten. Es gibt eine Variante dieser Metrik, die auf einer festen Skala arbeitet, LCOM* genannt. Die Berechnungsvorschrift für LCOM* nach der Henderson-Sellers-Methode ist wie folgt (vgl. Sauer, 2009):

$$\text{LCOM*} = \frac{\left(\left(\sum_{i=1}^{n} m(A_i)\right) \Big/ a\right) - m}{m - 1}. \tag{2.5}$$

Hierbei steht n für die Anzahl der Attribute und $m(A_i)$ für die Anzahl der Methoden, die auf das Attribut i zugreifen. Es wird also die durchschnittliche Anzahl an Methoden, die

auf die Objektattribute zugreifen, gebildet. Von dieser wird dann die Anzahl an Methoden m der Klasse abgezogen und das Ergebnis wird durch $m-1$ geteilt. Somit bilden sich Werte zwischen 0 und 1, wobei ein hoher Wert einen Mangel an Kohäsion bedeuten soll. Die Aussagekraft ist hingegen stark umstritten, zum einen, weil empirische Untersuchungen keine Signifikanz der Metrik gezeigt haben (vgl. Hoffmann, 2008), zum anderen, da zum Beispiel in Java das konsequente Nutzen von Get- und Set-Methoden von der Metrik bestraft wird (vgl. Sauer, 2009). Auch Mäkelä und Leppänen (2006) bemerken, dass ein guter Wert zwar ein gutes Design der Klasse anzeigt, aber ein schlechter Wert nicht unbedingt ein schlechtes Klassendesign anzeigen muss. Einige Designentscheidungen, wie zum Beispiel ein Design, das auf eine Verwendung durch Vererbung zugeschnitten ist führt zu schlechten LCOM-Werten. Ein solches Design muss aber nicht unbedingt schlecht sein.

Es gibt auch Metriken, deren Ziel es ist, die Kopplung und nicht die Kohäsion einer Komponente zu bestimmen. Beispiele für solche Metriken sind *Afferent Coupling* (dt. hinführende Kopplung) und *Efferent Coupling* (dt. wegführende Kopplung). *Afferent Coupling* gibt die Anzahl der Klassen außerhalb des Paketes an, die von Klassen innerhalb des Paketes abhängen. *Efferent Coupling* gibt die Anzahl der Klassen innerhalb eines Paketes an, die von Klassen außerhalb des Paketes abhängen.

2.5.3 Visualisierungen von Metriken

Bei der Darstellung von Metriken kommen unterschiedliche Visualisierungsformen zum Einsatz, die jeweils unterschiedliche Einsatzzwecke sowie spezifische Vor- und Nachteile besitzen. Dass die standardmäßige Darstellung von Metriken in Tabellenform meist nicht zielführend ist, wurde bereits in Abschnitt 2.5 beschrieben. Um ein paar der Probleme dieser Darstellungsform zu beheben, kommen zum Teil auch farbliche Hervorhebungen sowie Faltungsmechanismen zum Einsatz. Diese Hilfsmittel können einige Probleme der Tabellenform mildern, aber die grundsätzlichen Nachteile nicht beheben. Im Laufe der Zeit wurden verschiedene Visualisierungsformen, vor allem aus dem Bereich der Statistik, für die Softwareentwicklung adaptiert. Dabei ist zu beachten, dass diese Visualisierungsformen nicht speziell für den Bereich der Softwareentwicklung entwickelt wurden, weshalb sie nur in einem engen Rahmen für den Bereich der Visualisierung von Metriken geeignet sind. Drei dieser Visualisierungsformen werden im folgenden Abschnitt näher beleuchtet. Es wird auf die Möglichkeiten, aber auch die Unzulänglichkeiten dieser Diagrammarten eingegangen.

2.5.3.1 Ausgewählte Visualisierungsformen

Das Paretodiagramm ist im Grunde ein einfaches Balkendiagramm mit nach Größe geordneten Balken. Zusätzlich wird noch eine Kurve eingezeichnet, die die akkumulierten Balkengrößen angibt. Der Name des Paretodiagramms geht auf Vilfredo Pareto, einen italienischen Ökonom, Soziologen und Ingenieur zurück, der entdeckte, dass der Reichtum in Italien ungleichmäßig verteilt ist und dass 80% des Geldes sich im Besitz von 20% der Bevölkerung befinden. Später wurde diese Beobachtung auch in anderen Bereichen gemacht und man for-

mulierte aus dieser Beobachtung die sogenannte 80/20-Regel (vgl. Hoffmann, 2008). Der Vorteil des Paretodiagramms im Bereich der Softwareentwicklung besteht in der Möglichkeit, schnell zu erkennen, welche Elemente einen hohen Anteil an der gemessenen Größe aufweisen und deshalb genauer betrachtet werden sollten. Die Nachteile liegen darin, dass nur eine Größe elementübergreifend vergleichbar und die Struktur eines Softwaresystems nicht mehr erkennbar ist.

Beim Streudiagramm oder *scatterplot* werden die verschiedenen Elemente eines Systems in einem Koordinatensystem eingetragen, dessen Achsen jeweils durch unterschiedliche Metriken belegt sind (Abb. 2.1). Auf diese Weise ist es möglich, Korrelationen zwischen einzelnen Metriken zu untersuchen und zu bewerten. Streudiagramme werden häufig in einem zweidimensionalen Koordinatensystem verwendet. Es ist auch möglich, ein dreidimensionales Koordinatensystem für die Eintragungen zu verwenden, wodurch die Korrelationen von drei unterschiedlichen Metriken betrachtet werden können (vgl. Hoffmann, 2008). Bei Streudiagrammen gehen aber auch einige Informationen verloren, die für die Betrachtung eines Softwaresystems von großer Wichtigkeit sind. Als Erstes ist hierbei der Verlust der Information zu nennen, welches Element eines Softwaresystems wie bewertet wurde, da die Bezeichner der Elemente meist nicht in das Diagramm aufgenommen werden. Die zweite wichtige Information, die im Streudiagramm verloren geht, ist die Struktur des Softwaresystems.

Kiviat-Diagramme beziehungsweise Netzdiagramme stellen n Messwerte für ein einzelnes Element parallel dar (Abb. 2.2). Sie entstehen, indem von einem Ursprungspunkt aus n Achsen, auf denen jeweils ein Messwert abgetragen wird, gleichmäßig verteilt eingezeichnet werden. Wenn für alle Messwerte gilt, dass jeweils entweder größere oder kleinere Werte besser sind, ist es zudem möglich, die einzelnen Punkte zu verbinden, um eine Fläche zu bilden, die dann entweder maximal oder minimal sein sollte (vgl. Hoffmann, 2008). Werden Kiviat-Diagramme übereinander gelegt, ist es möglich, intuitiv zu bewerten, welches dargestellte Element besser als das jeweils andere ist. Zudem dienen Kiviat-Diagramme, bei entsprechender Einzeichnung von Grenzwerten, dazu, schnell erkennen zu können, welche Grenzwerte überschritten werden und wie stark die Überschreitung ist. Wegen ihrer intuitiven Verständlichkeit wird diese Diagrammform vielfältig auch im nichtwissenschaftlichen Bereich eingesetzt, zum Beispiel im Rahmen von Fußballsimulationen am Computer, um unterschiedliche Mannschaften schnell miteinander vergleichen zu können. Der große Nachteil

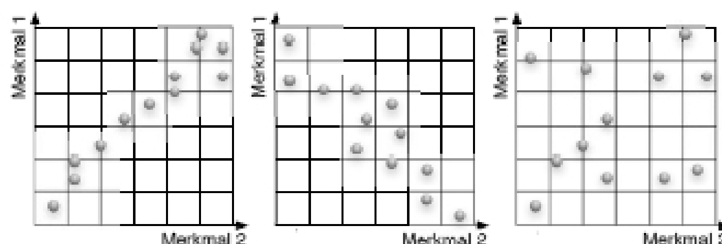

Abbildung 2.1: Streudiagramme mit unterschiedlichen Korrelationen der einzelnen Metriken (Quelle: Hoffmann, 2008, S. 269))

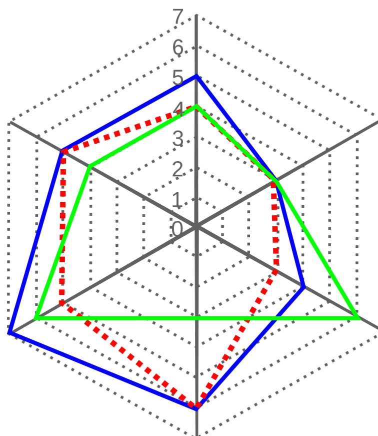

Abbildung 2.2: Kiviatdiagramm (Quelle: Wikipedia, 2009))

für die Zwecke dieser Arbeit besteht darin, dass Kiviat-Diagramme entweder zur Bewertung einzelner Elemente oder des Gesamtsystems genutzt werden können. Bei der Betrachtung von Einzelelementen geht dabei die Struktur des Gesamtsystems verloren und bei der Bewertung des Gesamtssystems geht die Information verloren, welche Elemente den stärksten Beitrag zur Bewertung liefern.

Liggesmeyer et al. (2009) zeigen eine weitere Form der Metrikvisualisierung in Form einer dreidimensionalen *treemap* (Abb. 2.3). Hierbei können unterschiedliche Daten zur Codequalität unterschiedlichen Eigenschaften der Quader, wie zum Beispiel Position, Größe, Höhe und Farbe, zugewiesen werden. Zudem stehen bei dieser Form der dreidimensionalen Visualisierung noch unterschiedliche Interaktionsmöglichkeiten zur Verfügung. Die großen Vorteile liegen hier in der Konfigurierbarkeit der Darstellung und der Möglichkeit, mehrere Metriken gemeinsam betrachten zu können. Der Nachteil der Darstellung besteht im Verlust der Struktur des Softwaresystems.

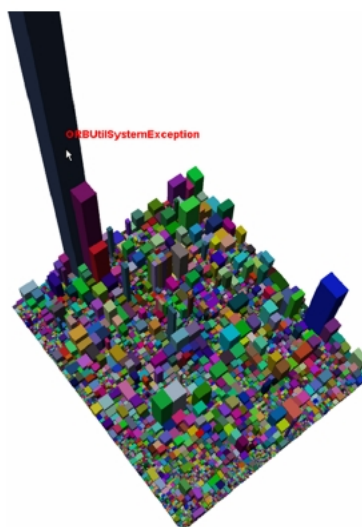

Abbildung 2.3: Dreidimensionale *treemap* (Quelle: Liggesmeyer et al., 2009, S. 6))

3 Softwarevisualisierung

3.1 Definition

Der Mensch ist nicht sehr gut darin, große Zahlenberge zu begreifen und zu verstehen. Die allgemeine Visualisierung versucht deshalb, Daten grafisch aufzubereiten, sodass sie von menschlichen Nutzern besser interpretiert und erfasst werden können.

Die allgemeine Visualisierung zerfällt zum einen in die Datenvisualisierung und zum anderen in die Informationsvisualisierung. Diese beiden sich zum Teil überlappenden Formen unterscheiden sich dahingehend, dass die Datenvisualisierung versucht, Daten, die aus der Messung physischer Vorgänge und Zustände entstehen, darzustellen. Die Informationsvisualisierung beschäftigt sich hingegen mit der Visualisierung von Informationen, also von abstrakten Daten.

Software ist abstrakt. Sie hat dewegen den Nachteil, dass man sie nicht anfassen, befühlen und vor allem nicht ansehen kann. Die Softwarevisualisierung versucht nun bei der Unsichtbarkeit Abhilfe zu schaffen. Durch die große Komplexität von Software hat sich noch keine ganzheitliche Visualisierung von Software herausgebildet. Vielmehr versuchen die unterschiedlichen Visualisierungsformen, unterschiedliche Aspekte von Software darzustellen. So soll erreicht werden, dass die einzelnen Aspekte von den Nutzern einer Visualisierung besser erkannt und begriffen werden können.

Im Laufe der Zeit haben sich unterschiedliche Definitionen der Softwarevisualisierung herausgebildet. Die vorliegende Arbeit baut auf der Definition von Knight und Munro (1999) auf:

> *„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.“*

Da die vorliegende Arbeit vor allem auf die Visualisierung von bereits vorhandenen Softwaresystemen abzielt, ist die Beschränkung auf existierende Software hierbei zielführend. Die Ziele der Visualisierung, die in der Definition genannt sind, sind ebenfalls deckungsgleich mit den Zielen dieser Arbeit. Knight und Munro (1999) gehen in ihrem Artikel vor allem auf die Visualisierung von Programmcode ein. Im Rahmen dieser Arbeit sollte der Begriff „Software“ allerdings weiter gesehen werden, sodass nicht nur der eigentliche Quellcode die Software darstellt, sondern auch weitere Artefakte, wie zum Beispiel die Daten, die aus der Erstellung von Codemetriken entstehen.

3.2 Einsatzbereiche und Kategorisierung

Softwarevisualisierungen kommen in den unterschiedlichsten Bereichen zum Einsatz. Zu den visualisierten Aspekten von Software gehören dabei nicht nur die Struktur und ande-

re Bereiche des Quellcodes, sondern auch die Datenflüsse und das Verhalten eines Softwaresystems (vgl. Gračanin et al., 2005). Dies sind dabei die Bereiche eines Softwaresystems, die von der spezifischen Definition von Knight und Munro (1999) abgedeckt werden. Gračanin et al. (2005) nennen allerdings auch weitere Bereiche der Softwarevisualisierung, die ebenfalls eine große Bedeutung besitzen, zum Beispiel die Animation von Algorithmen, die Fehlerdiagnose, die Anforderungsanalyse oder die Darstellung der Fortschritte während der Softwareentwicklung.

Softwarevisualisierung lässt sich nach unterschiedlichsten Kriteren und Attributen kategorisieren. Die wohl bekanntesten sind die Taxonomien von Roman und Cox (1993), Price et al. (1993) und Maletic et al. (2002), die das Thema aus unterschiedlichen Blickwinkeln analysieren. Dabei legen die unterschiedlichen Taxonomien unterschiedliche Fokusse. Die Taxonomie von Roman und Cox (1993) legt vor allem Wert auf die Visualisierung an sich. Somit werden einige Aspekte rund um die Verwendung der Visualisierung ausgeblendet. Bei Roman und Cox (1993) kommen diese Aspekte vor, aber auch hier liegt der Schwerpunkt auf der Art und Weise, wie die Visualisierung aufgebaut ist und nicht auf der Aufgabe, die mit einer Visualisierung verfolgt wird (vgl. Maletic et al., 2002; Gračanin et al., 2005). Maletic et al. (2002) stellen in ihrer Betrachtung der Dimensionen und Einordnungsmöglichkeiten von Softwarevisualisierung die bei Roman und Cox (1993) und Price et al. (1993) vernachlässigten Aspekte mehr heraus. In ihrer Taxonomie geht es vor allem um die folgenden fünf Fragen:

- Wer benötigt die Visualisierung? (Nutzergruppe)
- Warum benötigt er die Visualisierung? (Aufgabe)
- Was will der Benutzer der Visualisierung dargestellt haben? (Quelle)
- Wie wird es dargestellt? (Repräsentation)
- Wo soll die Visualisierung dargestellt werden? (Medium)

Die aufgeführten Fragen lassen sich nicht oder nur sehr stark begrenzt voneinander unabhängig betrachten. Für die Eignung einer Softwarevisualisierung sollten alle Anforderungen, die sich aus ihnen ergeben, beachtet werden. Besondere Bedeutung kommt vor allem der Aufgabe und der Quelle einer Visualisierung zu. Eine Visualisierung, die dem Nutzer die Informationen aus der falschen Quelle darstellt oder die die Information für die falsche Aufgabe aufbereitet, kann als sinnlos erachtet werden. Es wird aber auch klar, dass diese Dimensionen nicht unabhängig von der Nutzergruppe einer Visualisierung gesehen werden können. Hierbei ist zum einen der Kenntnisstand des Nutzers zu beachten und zum anderen die unterschiedlichen Präferenzen bei der Wahl der Abstraktion der Darstellung von zum Beispiel Entwicklern und Softwarearchitekten.

Die Dimensionen Repräsentation und Medium, auf die die beiden letzten Fragen von Maletic et al. (2002) zielen, sind eher technisch geprägt. Auch sie besitzen verschiedene Wechselwirkungen, einerseits untereinander, aber auch zu den anderen Dimensionen. Die

Repräsentation einer Visualisierung definiert, wie die darzustellenden Informationen angezeigt werden sollen. Hierbei kommen Metaphern zum Einsatz, die in Abschnitt 3.4 näher beleuchtet werden sollen. Dabei ist zu beachten, dass die unterschiedlichen einsetzbaren Medien unterschiedliche Freiheitsgrade bezüglich der Metaphern aufweisen. Wird eine Softwarevisualisierung zum Beispiel ausschließlich in schwarz-weiß auf Papier gedruckt, entfallen die Möglichkeiten des Einsatzes von Farbe oder Tiefe als Informationsträger.

3.3 Vorgang der Visualisierung

Bei jeder Visualisierung sind einige Aufgaben immer vorhanden. Hierzu gehören neben der eigentlichen Darstellung der Visualisierung vor allem drei verschiedene Vorgänge: die Aufbereitung der Rohdaten, die Abbildung dieser abgeleiteten Daten auf ein abstraktes Visualisierungsobjekt (Mapping) und das Rendern der Daten zu einem darstellbaren Bild (vgl. Haber und McNabb, 1990).

Die Aufbereitung der Rohdaten, die aus unterschiedlichen Quellen stammen können, erfolgt, um die nachfolgenden Schritte zu vereinfachen oder gar erst zu ermöglichen. Allerdings argumentieren dos Santos und Brodlie (2004), dass das von Haber und McNabb (1990) aufgestellte Vorgehensmodell nicht unbedingt geeignet ist, um multivariate und multidimensionale Daten zu visualisieren. Sie schlagen deshalb ein erweitertes Vorgehensmodell vor, um diese Probleme zu lösen. Dazu ersetzen beziehungsweise erweitern sie den Schritt der Datenaufbereitung durch die beiden aufeinander folgenden Schritte Datenanalyse und Filterung. Die Datenanalyse stellt bei dos Santos und Brodlie (2004) einen Interpolationsschritt dar, der automatisch ausgeführt werden kann. Die Filterung hingegen ist ein Schritt, der nur durch menschliche Interaktion mit der Visualisierung möglich ist. Der Nutzer der Visualisierung muss hierbei wählen, welche Daten er in welcher Kombination visualisiert haben möchte. Dazu kann er auch mehrere Filter zur Selektion der gewünschten Datenmenge hintereinander kombinieren (vgl. dos Santos und Brodlie, 2004).

Beim Mapping geht es vor allem um die Schaffung eines abstrakten Visualisierungsobjektes der zu visualisierenden Daten (vgl. Haber und McNabb, 1990; dos Santos und Brodlie, 2004). Abstrakte Visualisierungsobjekte stellen Attributfelder bereit, die unterschiedliche Darstellungsparameter, wie zum Beispiel Größe, Ausrichtung, Farbe, Transparenz usw., abbilden. Um die aufbereiteten Rohdaten beziehungsweise die gefilterten, zu visualisierenden Daten auf ein abstraktes Visualisierungsobjekt abzubilden, müssen von dem Ersteller einer Visualisierung Transferfunktionen bereitgestellt werden, die diese Aufgabe übernehmen.

Die so erzeugten abstrakten Objekte werden dann während des Renderns in ein Bild überführt, das dargestellt werden kann. Der Begriff des Bildes ist dabei im Rahmen der dreidimensionalen stereoskopischen Darstellung etwas weiter zu verstehen.

Es sollte unbedingt darauf geachtet werden, dass gleiche zu visualisierende Sachverhalte oder Elemente immer auf eine konsistente Weise dargestellt werden. Zum Beispiel werden im Rahmen eines UML-Klassendiagramms Klassen immer als Rechtecke mit einer Zeichenkette (dem Klassennamen) dargestellt, niemals als Kreise oder Dreiecke; die Recht-

ecke an sich können aber durchaus in gewissen Grenzen unterschiedlich abgebildet werden. Auch die anderen Elemente eines UML-Klassendiagramms, wie zum Beispiel Assoziationen, Generalisierungs-, Kompositions- und Aggregationsbeziehungen werden ebenfalls konsistent dargestellt. Neben diesem Punkt sollte aber auch darauf geachtet werden, dass Elemente des Diagramms ebenfalls eindeutig Elementen des abzubildenden Systems zuzuordnen sind. Zum Beispiel sind Pfeile mit einer nicht ausgefüllten dreieckigen Spitze in UML-Klassendiagrammen immer Generalisierungsbeziehungen und niemals Kompositionsbeziehungen (vgl. Maletic et al., 2002).

Bei der Positionierung im dreidimensionalen Raum kommen unterschiedliche Algorithmen zum Einsatz. Meist sind es Abwandlungen der klassischen Algorithmen, die zum zweidimensionalen Graphzeichnen eingesetzt werden. Eine sehr viel versprechende Gruppe von Algorithmen, die im Rahmen der dreidimensionalen Visualisierung häufig eingesetzt werden, sind dabei die kraftgerichteten Algorithmen. Hierbei wird ein System aus Objekten hergestellt, zwischen denen Kräfte herrschen, die entweder dafür sorgen, dass sich die einzelnen Objekte anziehen oder abstoßen. Nun wird ein Gleichgewicht gesucht, bei dem die Summe aller Kräfte null ist. Kraftgerichtete Algorithmen bestehen zumeist aus zwei Teilen (vgl. Battista et al., 1999). Im ersten Teil wird das System mit den herrschenden Kräften definiert und im zweiten Teil wird ein Algorithmus beschrieben, der die Bestimmung des gesuchten Gleichgewichts ermöglicht. Battista et al. (1999) nennt vor allem zwei Gründe für die Beliebtheit der kraftgerichteten Algorithmen. Zum einen die physikalische Analogie, die für ein einfaches Verständnis des Algorithmus sorgt und zum anderen die oft einfache Programmierbarkeit. Zusätzlich zu diesen beiden Gründen könnte genannt werden, dass sie sich relativ problemlos auf dreidimensionale Graphen übertragen lassen, wie einige Ansätze zur dreidimensionalen Visualisierung zeigen (z. B. Dwyer, 2001; Müller, 2009). Ein Nachteil dieser Algorithmen ist, dass die Ergebnisse von einer Ausführung des Algorithmus zur nächsten differieren. Dies kann in einigen Fällen vernachlässigt werden. In anderen wiederum, in denen es gewünscht ist, Softwaresysteme automatisch im Rahmen der Softwareentwicklung zu visualisieren, entstehen so bei jedem Lauf des Algorithmus unterschiedliche Darstellungen, wodurch sich der Nutzer der Visualisierung immer wieder neu orientieren muss (vgl. Panas et al., 2007).

3.4 Metaphern

3.4.1 Definition und Kategorisierung

Eine Metapher ist laut Lakoff (2004) eine rhetorische Figur, die dazu dienen soll, eine reale Sache oder einen Sachverhalt so abzubilden, dass dem Nutzer der Metapher ein Verständnis beziehungsweise das Erfahren des realen Objekts oder Vorgangs möglich ist. Softwarevisualisierung ist zu großen Teilen von Metaphern abhängig, da Software durch ihren abstrakten Charakter nicht berührbar oder sichtbar ist. Für das Sichtbarmachen einer unsichtbaren Sache ist es deshalb erforderlich, Bilder zu definieren, die die gedanklichen Konstrukte, die sich

in der Software verbergen, darstellen können. Im Laufe der Zeit haben sich im Bereich der Softwarevisualisierung unterschiedliche Metaphern herausgebildet. Diese bieten eine große Spannbreite der Darstellungen. Dies liegt zum Großteil wahrscheinlich daran, dass Software so komplex ist, dass unterschiedliche Elemente und unterschiedliche Abstraktionen für die unterschiedlichen Einsatzbereiche einer Softwarevisualisierung erforderlich wurden. Metaphern können in diesem Bereich vor allem nach zwei Dimensionen kategorisiert werden:

- Echtweltmetapher vs. abstrakte Metapher
- zweidimensionale Metapher vs. dreidimensionale Metapher

Eine Echtweltmetapher ist eine Metapher, die die einzelnen darzustellenden Elemente einem in der realen Welt vorkommendem Bild zuweist. Als Beispiele sind hier die Stadtmetapher, die Raummetapher oder die Sonnensystemmetapher zu nennen. Abstrakte Metaphern bilden hingegen die darzustellenden Entitäten auf frei gewählte visuelle Elemente, wie zum Beispiel Würfel oder Zylinder ab und definieren auch die Verbindungen zwischen diesen Entitäten als ausgewählte Darstellungselemente (vgl. Maletic et al., 2001). Echtweltmetaphern haben sowohl Vorteile als auch Nachteile. So nennen Maletic et al. (2001) die Nutzung von vordefiniertem Wissen als Vorteil und natürliche Beschränkungen einer solchen Metapher als Nachteil. Zum Beispiel können bei der Stadtmetapher sehr gut Gebäude als Entitäten verwendet werden, da der normale Nutzer auch davon ausgehen wird, dass Gebäude Entitäten sind. Es gibt innerhalb der Stadtmetapher allerdings wenige Möglichkeiten, die Gebäude gleichzeitig in mehrere Hierarchien einzubinden, da auch innerhalb einer realen Stadt die Häuser nicht in mehrere Hierarchien eingebunden sind.

Metaphern können sowohl zwei- als auch dreidimensional sein. Meist beschränken sich zweidimensionale Metaphern für die Softwarevisualisierung auf abstrakte Elemente. Dies ist mutmaßlich darauf zurückzuführen, dass der Mensch in einer dreidimensionalen Welt lebt und deshalb wenige passende zweidimensionale Metaphern zur Verfügung stehen. Stattdessen werden zumeist im Bereich der zweidimensionalen Visualisierung verschiedene Arten von Graphen, wie zum Beispiel Netze oder Bäume eingesetzt. Ein Beispiel für eine zweidimensionale Echtweltmetapher wäre die Schreibtischmetapher, die bei den meisten Betriebssystemen mit grafischer Benutzungsoberfläche Verwendung findet.

3.4.2 Ausgewählte dreidimensionale Metaphern

Dieser Abschnitt soll sich mit unterschiedlichen Umsetzungen der Metaphern innerhalb der dreidimensionalen Softwarevisualisierung beschäftigen. Die gezeigten Beispiele erheben dabei keinen Anspruch auf Vollständigkeit, vielmehr sollen sie einen kleinen Einblick in die Möglichkeiten der dreidimensionalen Visualisierung geben. Zudem sollen die unterschiedlichen Parameter der Visualisierung dahingehend beleuchtet werden, ob und inwieweit die Metapher geeignet ist, die Struktur eines Softwaresystems zusammen mit den Metriken der einzelnen Elemente darzustellen.

3.4.2.1 Dreidimensionale Bäume

Die Darstellung von hierarchischen Strukturen in drei Dimensionen durch *cone trees*, also die Abbildung der Information mittels konisch angeordneter Baumstrukturen, wird von Robertson et al. (1991) vorgeschlagen. Diese Darstellung nutzt den Raum der zur Darstellung von hierarchisch aufgebauten Informationen besser als die zweidimensionalen Darstellungen geeignet ist, weil die Tiefe hier zusätzlich Elemente aufnehmen kann, ohne dass der Nutzer hierbei die Elemente aus seinem Blickfeld verliert. Robertson et al. (1991) nutzen diese Technik, um zum Beispiel Dateistrukturen innerhalb eines Unix- oder Windows-Systems darzustellen. Die hierbei gewählte Metapher, die in Abb. 3.1 gezeigt wird, kann aber auch gut auf die Klassen- und Paket-Hierarchie eines objektorientierten Softwaresystems übertragen werden. Die Metapher bietet einen guten Überblick über die dargestellten Elemente. Um Metrikwerte in der Metapher zu verankern bieten sich unterschiedliche Parameter an. Als vorrangig können dabei die Farbe, die Form, die Rotation der Objekte und in gewissen Grenzen die Größe genutzt werden. Die Metapher nutzt die Positionierung in allen drei Dimensionen, um die Hierarchie des zu visualisierenden Systems darzustellen. In der Folge ist die Positionierung als Parameter für eine Darstellung der Metriken quasi verloren. Als eine weitere Limitierung der Metapher führen Robertson et al. (1991) die Uneffektivität bei tiefen Hierarchien (mehr als 10 Ebenen) beziehungsweise bei Hierarchien mit sehr vielen Elementen (mehr als 1000 Elemente) an.

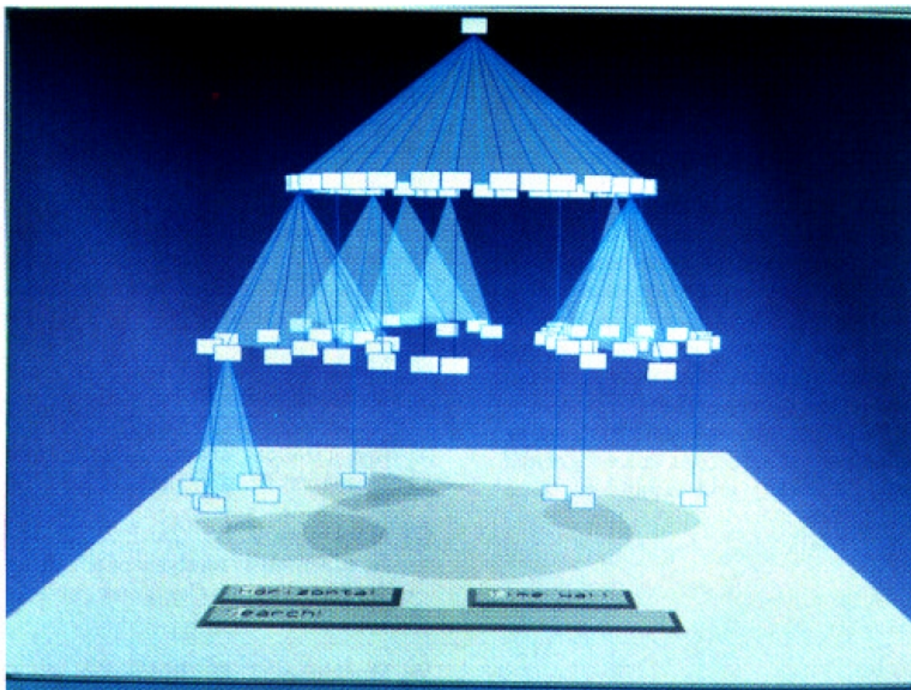

Abbildung 3.1: Dreidimensionale Bäume (Quelle: Robertson et al., 1991)

3.4.2.2 Schachtelungsmetapher

Diese Metapher wurde von Rekimoto und Green (1993) vorgeschlagen, um die Beschränkungen der Metapher von Robertson et al. (1991) aufzulösen. Rekimoto und Green (1993) schlagen dabei vor, ineinander geschachtelte transparente Quader zu verwenden, die ihre jeweiligen Unterelemente enthalten, um die hierarchischen Strukturen abzubilden (Abb. 3.2). Diese Metapher wird sehr leicht verstanden, da das Konzept der Schachtel als Behälter allgemein bekannt ist. Hierbei ist die Nutzung bestimmter Transparenzgrade wichtig und nicht nur die Darstellung der Rahmen der Behälter, da dies dazu führen würde, dass die einzelnen Zugehörigkeiten vom Nutzer nicht mehr überblickt werden können (vgl. Rekimoto und Green, 1993). Diese Metapher besitzt verschiedene Freiheitsgrade, die für die Darstellung von Metriken genutzt werden können, allerdings sind diese durch die Beschränkungen der Metapher meist beschnitten. Zu den Freiheitsgraden gehört die Größe, die Farbe und die Rotation. Von den genannten Parametern ist aber nur die Rotation ebenenübergreifend vergleichbar. Die Größe wird immer durch die Größe des Elternelements beschränkt. Die Farbe ist durch den Einsatz von Transparenz in tieferen Ebenen zum Teil verschieden. Auch für die Positionierung der Elemente in den einzelnen Würfeln herrschen Beschränkungen. Die größte Einschränkung ist, dass sich Objekte nicht überschneiden sollten. Wenn die Ausrichtung in x-, y- und z-Achse durch Metriken bestimmt wird, lässt sich nicht garantieren, dass sich keine Objekte überschneiden. Der Autor der vorliegenden Arbeit nimmt deshalb an, dass die Ausrichtung in einer der drei Richtungen nicht durch eine Metrik belegt werden sollte. Die Ausrichtung entlang dieser Achse kann dann dazu genutzt werden, die Objekte kollisionsfrei anzuordnen.

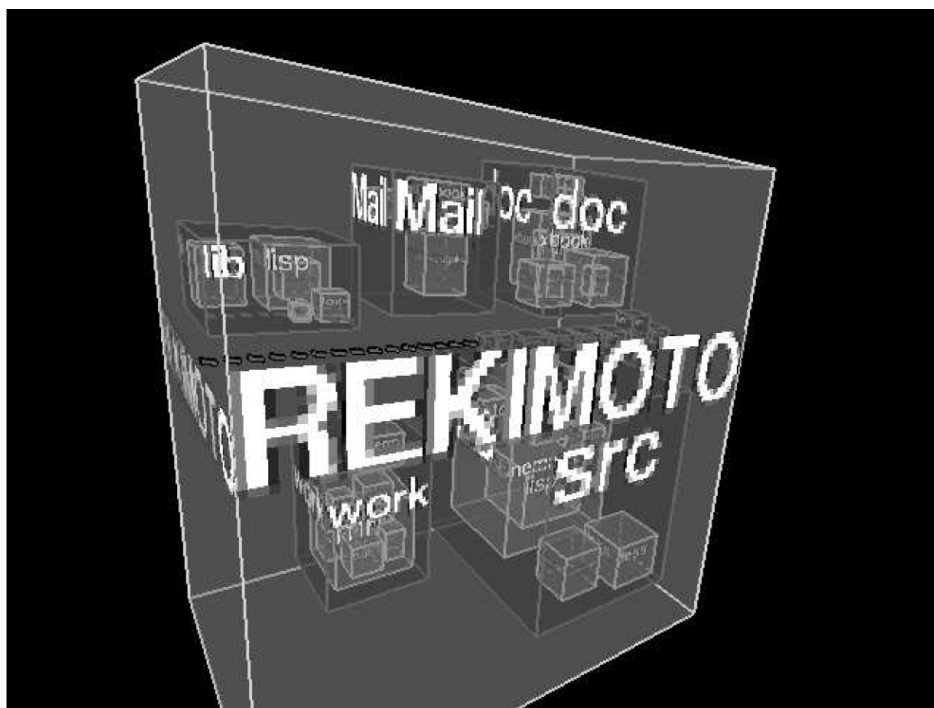

Abbildung 3.2: Schachtelungsmetapher (Quelle: Rekimoto und Green, 1993)

3.4.2.3 UML in 3D

Die Unified Modelling Language (UML) ist wohl die bekannteste Modellierungssprache für objektorientierte Systeme. Im Laufe der Zeit wurden viele Werkzeuge entwickelt, die UML aus Code erzeugen, aus UML Code erzeugen oder mit denen man UML-Diagramme darstellen und bearbeiten kann. Ein großer Vorteil der UML liegt in ihrer weiten Verbreitung. Wahrscheinlich jeder, der objektorientiert entwickelt, hat schon einmal ein UML-Diagramm gesehen oder ist zumindest in der Lage, ein einfaches UML-Diagramm zu verstehen. Meist werden mit der UML nur die verschiedenen Diagrammarten verbunden, die verschiedene Sichten auf ein Softwaresystem ermöglichen, indem sie unterschiedliche Aspekte in unterschiedlichen Abstraktionsebenen beleuchten. Die in der UML-Spezifikation aufgeführten Diagramme haben in dieser Form allerdings einige Nachteile. Durch ihre Zweidimensionalität können große Systeme nicht mehr ausreichend effektiv dargestellt werden, sie sind unübersichtlich und der Nutzer muss sich entscheiden, entweder nur einen Teilausschnitt zu sehen oder die einzelnen Elemente sehr klein anzeigen zu lassen. Zudem ist es aufgrund der unterschiedlichen vorhandenen Assoziationsarten der UML-Elemente möglich, dass es im zweidimensionalen Raum keine Möglichkeiten mehr gibt, diese ohne Überlappungen darzustellen, wodurch die Darstellung sehr unübersichtlich wird. Aus diesen Gründen haben sich im Laufe der Zeit dreidimensionale Darstellungen mit dem Ziel entwickelt, UML-Diagramme darzustellen. Ein Beispiel für diese Entwicklungen ist die Arbeit von McIntosh et al. (2005), die aus JavaML, einer vom Compiler erzeugten Auszeichnungssprache, für die Struktur eines Java-Systems durch Extentensible Stylesheet Language Transformations (XSLT) ein dreidimensionales Klassendiagramm erzeugen (Abb. 3.3). Für die Zwecke der Metrikdarstellungen bietet sich in dieser Darstellungsform sowohl die Farbe, als auch die Größe und die Rotation. Auch die Positionierung bietet sich an, wobei zu beachten ist, dass die Positionierung der Klassen bei Darstellungen, die die Assoziationen mit anzeigen, eventuell beschränkt sein kann.

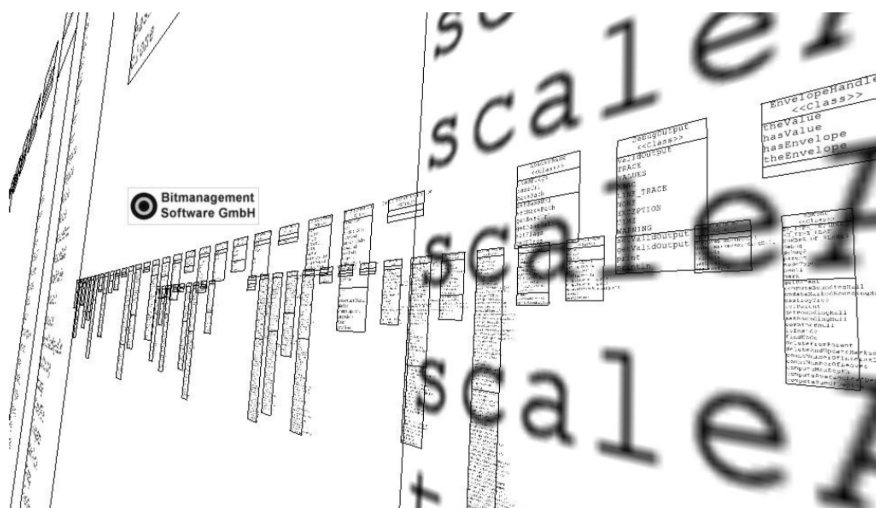

Abbildung 3.3: UML in 3D (Quelle: McIntosh et al., 2005)

3.4.2.4 Hemisphärenmetapher

Die Hemisphärenmetapher, die von Balzer und Deussen (2004) vorgeschlagen wurde, stellt die hierarchisch geprägten Strukturinformation von objektorientiert entwickelter Software dar. Abbildung 3.4 zeigt solch eine Repräsentation. Die Hemisphärenmetapher adaptiert die Grundprinzipien der Schachtelungsmetapher von Rekimoto und Green (1993), indem zum Beispiel die Pakete, die in anderen Paketen beinhaltet sind, in diesen dargestellt werden. Zusätzlich zu dieser Schachtelungseigenschaft bei Paketen, die als Sphären dargestellt werden, wird in der Implementierung von Balzer und Deussen (2004) mit unterschiedlichen Transparenzgraden gearbeitet, je nachdem, wie weit sich der Betrachter von der jeweiligen Sphäre entfernt. Bei einem Abstand, der mehr als fünfmal so groß ist wie der Radius der Halbkugel, wird die jeweilige Sphäre völlig undurchsichtig, um den Nutzer vor zu vielen Details zu bewahren. Klassen werden bei dieser Metapher als Kreise dargestellt, die je nach der Anzahl ihrer Attribute und Felder größer oder kleiner sind. Die Attribute und Methoden werden hingegen als Schachteln am Boden der Hemisphäre dargestellt. Für die Darstellung von Metriken bietet die Metapher verschiedene Möglichkeiten, allerdings sind hierbei ähnliche Restriktionen wie bei der normalen Schachtelungsmetapher zu beachten. Die Größe ist durch die Größe des Elternelements beschränkt, die Farbe wird durch die Transparenzen verfälscht dargestellt und auch die Positionierung muss so gewählt werden, dass die Objekte immer noch ineinander geschachtelt sind. Die Form der Objekte kann durch Metriken bestimmt werden. Hierbei ist aber zu beachten, dass die Form nur für eine kleine Gruppe von Metriken geeignet ist. Ob die Rotation als Informationsträger genutzt werden kann, müsste geprüft werden. Jedenfalls könnte ein Benutzer durch den Boden in dieser Darstellung wahrscheinlich besser als bei schwebenden Schachteln erkennen, ob und wie stark ein Objekt rotiert ist.

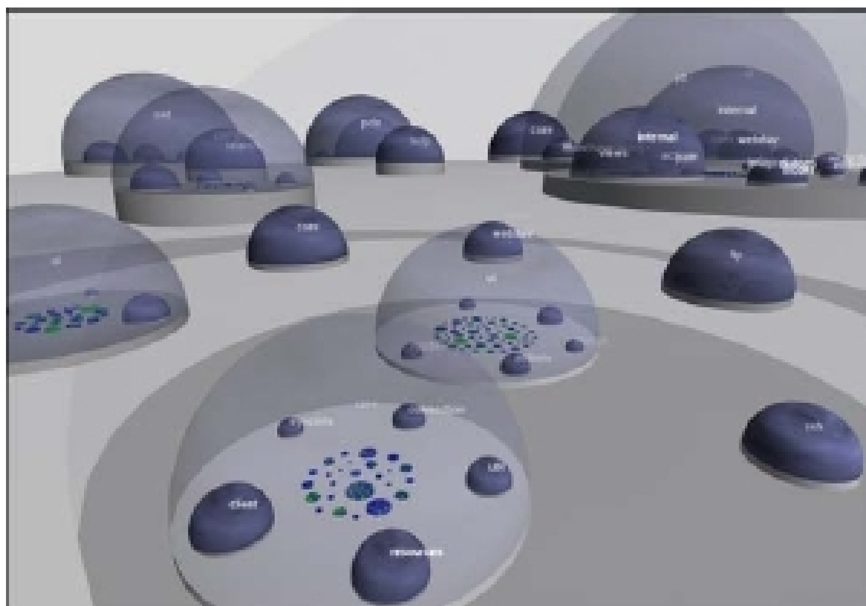

Abbildung 3.4: Hemisphärenmetapher (Quelle: Balzer und Deussen, 2004)

3.4.2.5 Stadtmetapher

Die Stadtmetapher ist eine sogenannte Echtweltmetapher. Sie dient dazu, Strukturinformationen unter Nutzung des bereits vorhandenen Wissens der Nutzer darstellen zu können. Die Metapher wird relativ häufig als geeignet angesehen, um Informationen darzustellen (vgl. z. B. Knight und Munro, 1999; Wettel und Lanza, 2007; Panas et al., 2007). Wie Panas et al. (2007) zeigen, sind damit auch unterschiedliche Abstraktionsgrade dieser Strukturinformationen darstellbar, sodass sowohl Softwarearchitekten als auch Entwickler die selbe Darstellung benutzen können. Dies führt zu einer besseren Kommunikation der unterschiedlichen Interessensbeteiligten. In der von Panas et al. (2007) gewählten Stadtmetapher werden Funktionen als Gebäude abgebildet (Abb. 3.5). Die Texturen der jeweiligen Gebäude spiegeln die Metriken der jeweiligen Funktionen wider. Städte stellen in dieser Metapher die einzelnen Dateien dar. Wassertürme repräsentieren die Header-Dateien und die Landschaften sind das Darstellungselement für die Ordner, in denen die einzelnen Dateien liegen. Für die Darstellung von Metriken bieten sich in dieser Darstellungsform einige Parameter an. Hierzu gehören die Größe, die Farbe, die Form und die Rotation von Objekten. Die Rotation könnte in einer solchen Metapher sogar intuitiv eingesetzt werden, da ein schief stehendes Haus wahrscheinlich recht gut einer Funktion zugeordent werden könnte, in der etwas „schief“ läuft. Die Positionierung der einzelnen Elemente ist in y-Richtung beschränkt, da man die Häuser in der Darstellung nicht vom Boden abheben kann, ohne die Metapher sehr stark zu verfälschen und die Identifikation der einzelnen Elemente durch den Nutzer zu untergraben. Die Größe kann in dieser Metapher auf verschiedene Weise genutzt werden. Der Nutzer der Metapher wird bei einem hohen Gebäude bestimmte Assoziationen haben, zum Beispiel

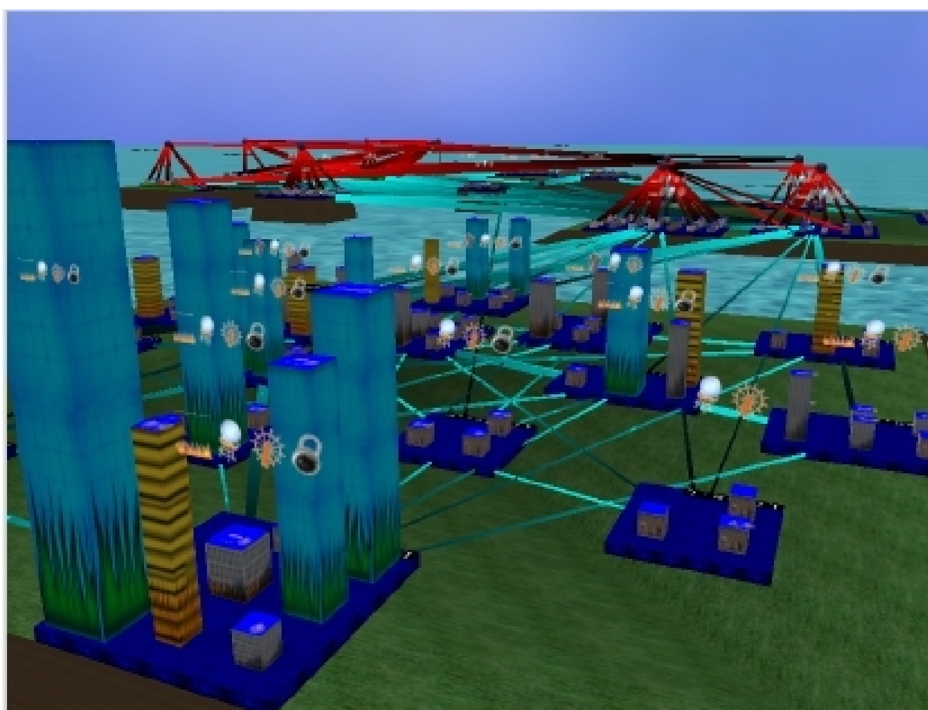

Abbildung 3.5: Architekturvisualisierung mittels Stadtmetapher (Quelle: Panas et al., 2007)

zu einem Bürogebäude oder ähnlichem. Eine große Grundfläche des Gebäudes bei geringer Höhe wird eher mit einem Lagerhaus oder Ähnlichem in Verbindung gebracht werden. Somit ist die Größe in dieser Metapher auf vielfältige Art und Weise nutzbar.

3.4.2.6 Sonnensystemmetapher

Die Sonnensystemmetapher kann ebenfalls wie die Stadtmetapher den Echtweltmetaphern zugeordnet werden, da sie ebenfalls versucht, das vorhandene Wissen über den Aufbau eines Sonnensystems zu nutzen. Es kann aber auch argumentiert werden, dass kein Mensch diese Anordnung je nachweisbar mit eigenen Augen gesehen hat. Da die einzelnen Elemente bei der Metapher zur besseren Übersichtlichkeit meist deutlich näher zusammen dargestellt werden ist eine solche Anordnung der Planeten und Sonnen auch physikalisch kaum möglich, da die Gravitationskräfte die einzelnen Objekte zu stark zusammenziehen würden. Aus diesen Gründen kann die Sonnensystemmetapher durchaus auch als abstrakte Metapher gesehen werden.

In der von Graham et al. (2004) eingesetzten Form stellen die Sonnen Java-Pakete und die Planeten Klassen beziehungsweise Interfaces dar (Abb. 3.6). Die Farbe der Klassen repräsentiert bei Graham et al. (2004) entweder den Unterschied zwischen Schnittstellen und Klassen oder stellt die Kopplung der Klassen dar. Die Kreise um die Klassen stellen die Aufrufhierarchie der Klasse dar. Die Größe der Planeten wird durch die Metrik *Lines of Code* bestimmt. Hierbei zeigt sich, dass durch die Wahl dieser Metapher auch mehrere Metriken gleichzeitig dargestellt und verstanden werden können. Weitere mögliche Parameter für eine

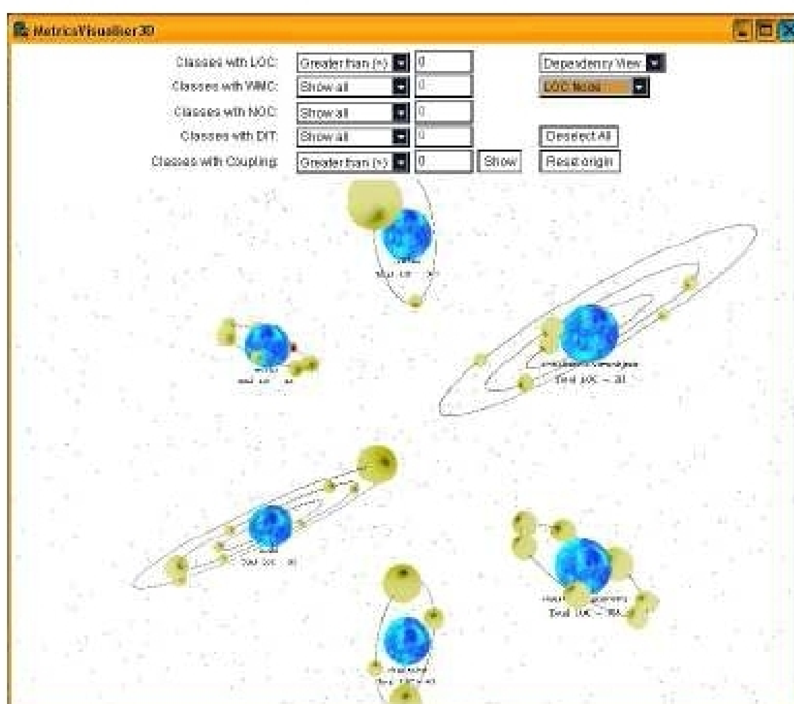

Abbildung 3.6: Visualisierung der *Lines of Code* mit der Sonensystemmetapher (Quelle: Graham et al., 2004)

Darstellung von Metrikinformationen wären die Form der Objekte, ihre Farbe, ihre Rotation und ihre Ausrichtung in *x*-, *y*- und *z*-Achse. Die Ausrichtung ist allerdings eingeschränkt, da die Zugehörigkeiten der Objekte zueinander noch erhalten bleiben müssen.

3.4.2.7 Imsovision-Metapher

Imsovision (Immersive Software Visualization) ist ein System, das von Maletic et al. (2001) vorgestellt wurde. Es dient der Darstellung objektorientierte Software mittels VRML (Virtual Reality Markup Language) in einer virtuellen Umgebung und soll auf diese Weise helfen, Programme besser zu verstehen und untersuchen zu können. Die Metapher, die im Rahmen dieses Systems eingesetzt wird, ist eine abstrakte Metapher, die unterschiedliche grafische Primitiven einsetzt, um ein Verständnis für das System zu ermöglichen (Abb. 3.7). In diesem Rahmen werden auch die Möglichkeiten der virtuellen Umgebungen ausgenutzt, indem zum Beispiel ein Benutzer nur dann die privaten Elemente einer Klasse sehen kann, wenn er die Plattform, die diese Klasse repräsentiert, umdreht. Attribute werden in dieser Metapher durch Kugeln und Methoden durch Säulen abgebildet. Die Farbe der Säule lässt den Nutzer dabei zwischen den unterschiedlichen Methodenarten differenzieren. Die Größe einer Plattform spiegelt die Anzahl von Attributen und Methoden der Klasse wider, die Größe einer Säule repräsentiert die Anzahl der Codezeilen der Methode. Die dabei verwendete Metapher bietet durch ihren abstrakten Charakter sehr viele Freiheiten bezüglich der Darstellung von Metriken, die zum Teil auch schon innerhalb des Systems genutzt werden. Die Positionierung der Elemente in mindestens zwei der drei möglichen Richtungen ist hierbei möglich. Diese Positionierungen werden aktuell dazu genutzt den Zweck der Objekte darzustellen. Maletic et al. (2001) verwenden hierzu eine Stadt als Vergleichsobjekt, um die Positionierung der einzelnen Elemente zu erläutern. So werden abgeleitete Klassen um ihre Basisklasse angeordnet, ähnlich wie bei der Anordnung von Stadtkernen und Vororten. Zusätzlich zu den angeführten

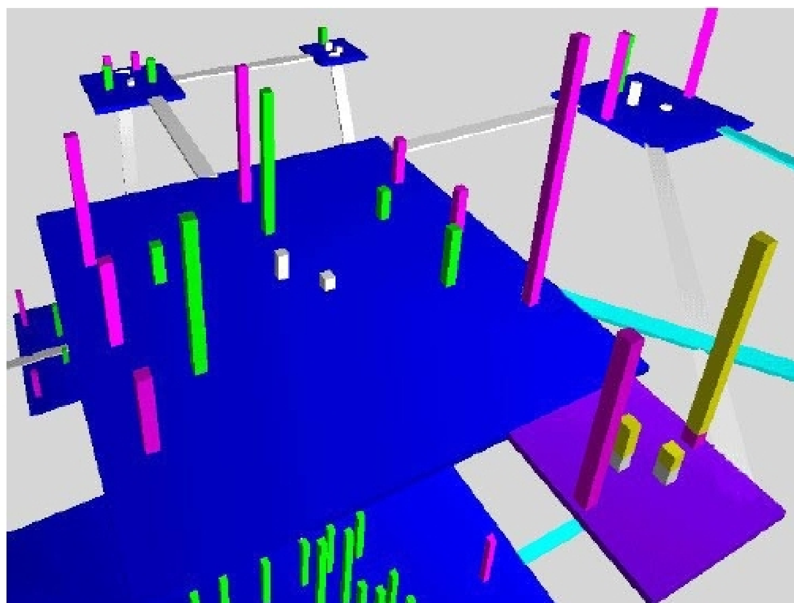

Abbildung 3.7: Architekturvisualisierung mittels Imsovision (Quelle: Maletic et al., 2001)

Metapher	Ausrichtung x	y	z	Größe	Form	Farbe	Rotation
Dreidimensionale Bäume	−	−	−	+	+	+	+
Schachtelungsmetapher	○	○	○	○	○	+	+
UML in 3D	+	+	+	+	−	+	+
Hemisphärenmetapher	○	○	○	○	○	○	○
Stadtmetapher	+	−	+	+	+	+	+
Sonnensystemmetapher	○	○	○	+	+	+	+
Imsovision	+	○	+	+	+	+	+

Tabelle 3.1: Bewertungen der Möglichkeiten zur Metrikvisualisierung. +: gut nutzbar; ○: eingeschränkt nutzbar; −: kaum nutzbar

Parametern könnte auch die Rotation von Objekten in gewissen Rahmen eingesetzt werden. Diese wird aktuell aber noch nicht genutzt (vgl. Maletic et al., 2001).

3.4.2.8 Gegenüberstellung der Metaphern

Um die einzelnen Metaphern und ihre Möglichkeiten besser miteinander vergleichen zu können, vergleicht Tabelle 3.1 die zuvor gezeigten Möglichkeiten zur Darstellung von Metriken. Hierbei kann ein Parameter innerhalb der Methaper gut (+), eingeschränkt (○) oder kaum (−) für die Darstellung von Metriken nutzbar sein. Die Bewertungen hängen dabei nicht von den tatsächlich bereits für die Metrikvisualisierung eingesetzten Parametern ab, sondern ausschließlich von der prinzipiellen Möglichkeit, dies zu tun.

3.5 Virtuelle Umgebungen

Virtuelle Umgebungen ermöglichen es dem Nutzer, Objekte nicht nur dreidimensional zu sehen, sondern lassen ihn vielmehr in eine dreidimensionale Welt eintauchen, mit der er interagieren und in der er sich relativ frei bewegen kann. Dieses Eintauchen in die virtuelle Welt wird als „Immersion“ bezeichnet. Die dreidimensionale Darstellung von Informationen hilft dem Nutzer außerdem dabei, sich Wege besser zu merken und Größen von Objekten besser vergleichen zu können (vgl. Maletic et al., 2001). Dies ist wohl darauf zurückzuführen, dass der Mensch gewohnt ist, sich innerhalb von dreidimensionalen Räumen zu bewegen und sich dort vollzogene Wege zu merken. Virtuelle Umgebungen bieten dem Nutzer verschiedene Möglichkeiten der Navigation und der Interaktion mit seiner Umgebung. Als Beispiel sind hier die Auswahl und das Bewegen von Objekten oder auch das Hineintauchen in Objekte zu nennen. Als ein Merkmal, das allen virtuellen Umgebungen gleichermaßen zu eigen ist, nennen Gračanin et al. (2005) die Möglichkeit, ein Objekt eingehend zu untersuchen, indem der Nutzer sich nah an das Objekt heranbewegt, wobei der Rest der virtuellen Welt trotzdem in der Ferne im Sichtfeld des Nutzers bleibt.

Das Institut für Wirtschaftsinformatik an der Universität Leipzig besitzt ein Virtual Reality System der Firma IC:IDO, mit dem es möglich ist, dreidimensionale Modelle interaktiv

zu erkunden und zu erforschen. Das System besteht aus drei Beamerpaaren, die jeweils von einem Server gesteuert werden und die zusammen eine Powerwall in zwei leicht verschobenen Farbräumen bespielen, sodass es mit Hilfe von Interferenzfilterbrillen möglich ist, eine räumliche Wirkung zu erzeugen. Das Virtual Reality Labor ist zudem mit einem Tracking-System ausgerüstet, welches die Position einer Brille und eines Eingabegeräts bestimmen kann, sodass für den Nutzer ein deutlich höherer Grad der Immersion und eine Interaktion mit den Modellen möglich wird.

3.6 Extensible 3D als Basis

Müller (2009) geht in seiner Arbeit ausgiebig auf die unterschiedlichen Möglichkeiten zur Visualisierung von Softwaresystemen ein, Er evaluiert unter anderem Techniken wie Open GL oder Java 3D und kommt zu dem Schluss, dass für die Visualisierung von Software mittels eines Generators Extensible 3D (X3D) ein sehr geeignetes Format darstellt. Deswegen soll X3D im Rahmen dieser Arbeit vorgestellt und betrachtet werden. Es kommt auch im Prototyp als Zielformat zum Einsatz. Als Gründe für die Wahl von X3D nennt Müller (2009) die grafischen Möglichkeiten, die Erweiterbarkeit und die Entscheidung für XML als Trägerformat. Die Wahl einer XML-basierten Sprache beruht dabei vor allem darauf, dass openArchitectureWare mit XML-Schemata als Metamodell umgehen kann.

X3D kann folgendermaßen definiert werden (Web3D Consortium, 2008):

> *„Extensible 3D (X3D) is a software standard for defining interactive web- and broadcast-based 3D content integrated with multimedia.“*

X3D ist also vor allem entwickelt worden, um dreidimensionale Anwendungen mit Multimedia-Funktionalitäten im Web zu ermöglichen. Um dieses Ziel zu erreichen sollte es also möglich sein, X3D-Datei zwischen unterschiedlichen Endgeräten auszutauschen und für unterschiedliche Anwendungsdomänen einzusetzen.

Als Merkmale von X3D, die für die Zwecke der dreidimensionale Softwarevisualisierung von Bedeutung sind, sind vor allem die folgenden Punkte aufzuführen (Web3D Consortium, 2008):

- 3D-Grafik
- 2D-Grafik
- Einbindung von Multimedia-Dateien in dreidimensionale Szenen
- Benutzer-Interaktion
- Navigation
- Möglichkeiten, die Umgebung mittels Skriptsprachen zu verändern

Hierbei sind auch solche Fähigkeiten von X3D aufgeführt, die im Rahmen der Softwarevisualisierung noch nicht oft eingesetzt werden, zum Beispiel die Einbindung von Multimedia-Dateien. So ist vorstellbar, dass eine bedrohlich klingende Audio-Datei abgespielt wird,

wenn sich der Benutzer in einem Bereich der Software befindet, deren Zustand kritischer ist, oder dass der Benutzer durch bestimmte Geräusche auf bestimmte andere Bereiche mit spezifischen Eigenschaften aufmerksam gemacht wird.

3.6.1 X3D-Architektur

Um die Ziele Plattformunabhängigkeit und die Anwendbarkeit für unterschiedliche Domänen zu erreichen, wurde vom Web3D Consortium bei der Spezifikation darauf geachtet, Daten und Ausführungsumgebung zu trennen. Diese Trennung hat vor allem den Vorteil, dass die Daten zwischen verschiedenen Ausführungsumgebungen ausgetauscht werden können. So ist es zum Beispiel möglich, die X3D-Modelle, die der im Rahmen dieser Arbeit entwickelte Generator erzeugt, in unterschiedlichen X3D-Browsern zu betrachten. Deren Ausprägungen reichen vom Browser-Plugin bis hin zur vollständigen Applikation.

Die Architektur einer X3D-Anwendung besteht aus einer Datenstruktur, dem sogenannten Szenegraphen und einer Laufzeitumgebung (Abb. 3.8). Der Szenegraph ist ein gerichteter azyklischer Graph, der die darzustellenden Objekte und die Verbindungen zwischen diesen Objekten enthält. Die einzelnen Knoten in einem Szenegraphen sind dabei nicht als starre Körper zu betrachten. Vielmehr bietet X3D hier eine Reihe von Möglichkeiten, um deren Zustände und Übergänge zu beschreiben (vgl. Behr et al., 2004). Der Szenegraph bietet zwei verschiedene Hierarchien, nämlich die Transformations- und die Verhaltenshierarchie. In X3D gibt es deshalb auch zwei Arten von Verbindungen. Zum einen die durch die Transformationshierarchie vorgegebenen Verbindungen und die durch die Verhaltenshierarchie beschriebenen Laufzeitverbindungen zwischen Feldern und Ereignisströmen in der beschriebenen dreidimensionalen Welt. Der Szenegraph ermöglicht außerdem geschachtelte Hierarchien. Er enthält Knoten mit Feldern, die die Eigenschaften des Knotens widerspiegeln. Ein solcher Knoten kann auch Unterknoten besitzen, die wiederum durch Felder beschrieben werden und die selbst wieder Unterknoten oder Instanzen von Knoten besitzen

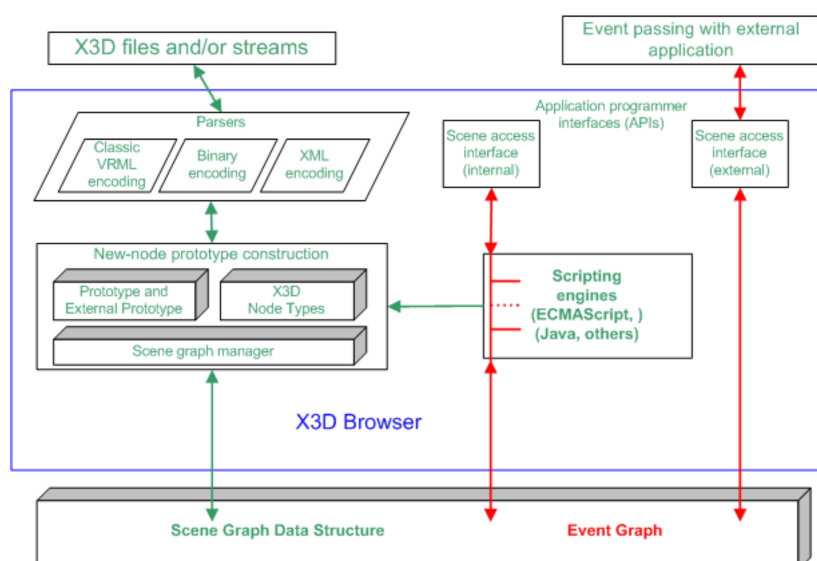

Abbildung 3.8: X3D-Architektur (Quelle: Web3D Consortium, 2008)

können (vgl. Geroimenko und Geroimenko, 2006).

Die Laufzeitumgebung der X3D-Anwendung dient dazu, den Szenegraph so zu verändern und darzustellen, dass die vom Ereignismodell definierten Änderungen umgesetzt werden. Zusätzlich ist die Laufzeitumgebung auch für die Interaktion mit der gewünschten Anwendung zuständig. Sie muss dafür eine Schnittstelle zum Parsen und zum Export der gewünschten Dateiformate und die Möglichkeiten der Beeinflussung des Szenegraphen bereitstellen. Szenegraphen können durch unterschiedliche Programmiersprachen, wie zum Beispiel ECMAScript und Java, beeinflusst werden (vgl. Geroimenko und Geroimenko, 2006).

3.6.2 X3D-Spezifikation und Profilkonzept

X3D hat den Standardisierungsprozess der International Standardization Organisation (ISO) durchlaufen. Der gesamte Standard für X3D ist in drei separate Standarddokumente gegliedert, die sich jeweils mit unterschiedlichen Bereichen von X3D beschäftigen und in ihrer Gesamtheit die X3D-Spezifikation bilden. Dies ist zum einen ISO/IEC 19775. Dieser Standard besteht wiederum aus drei Dokumenten, die sich mit der Beschreibung aller funktionalen Elemente von X3D und dem Zugriffsinterface auf Szenen, die ein X3D-Browser bieten sollte, beschäftigen. Bei der Beschreibung wird die Technologie, mit der diese Funktionalitäten umgesetzt werden, ausgeklammert. Die beiden darauf aufbauenden Standards sind ISO/IEC 19776 und 19777, die jeweils mehrere Unterdokumente haben. ISO/IEC 19776 beschäftigt sich mit den unterschiedlichen Datenformaten, in denen X3D-Modelle repräsentiert werden können. ISO/IEC 19777 spezifiziert Sprachanbindungen für Java und ECMAScript, mit denen das Verhalten von Objekten verändert werden kann (vgl. Müller, 2009; Web3D Consortium, 2008).

Durch Profile soll der Einstieg in die X3D-Technologie zum einen vereinfacht werden, zum anderen sollen auch fortgeschrittene Anwendungen möglich sein. Die Anwendungen, die X3D verarbeiten oder anzeigen, müssen auf diese Weise nicht unbedingt den kompletten Funktionsumfang abbilden, sondern können auch ein Profil mit einem geringeren Funktionsumfang implementieren.

Die X3D-Spezifikation unterscheidet zwischen sieben Profilen (Web3D Consortium, 2008):

- Core
- Interchange
- Interactive
- MPEG-4 Interactive
- Immersive
- Full
- CADInterchange

Die fünf Hauptprofile sind dabei: *Core*, *Interchange*, *Interactive*, *Immersive* und *Full*. Jedes dieser Profile umfasst den vollständigen Funktionsumfang des jeweils zuvor genannten

Profils und erweitert diesen (vgl. Anslow et al., 2006).

Die Profile unterscheiden sich vornehmlich in der Art und Anzahl der unterstützten Komponenten. Komponenten sind eine definierte und benannte Menge von Funktionen, die in ihrer Gesamtheit eine Funktionalität ermöglichen sollen. Dafür definieren die einzelnen Komponenten unterschiedliche Knotentypen und Dienste, die für die jeweilige Funktionalität von Bedeutung sind (vgl. Geroimenko und Geroimenko, 2006).

Das *Core*-Profil ist das Profil mit dem geringsten Funktionsumfang. Es dient der Definition der grundlegenden Dateiformate und Szeneninhalte. Ziel dieses Profils ist es, eine möglichst einfache Implementierungsmöglichkeit zu schaffen, damit eine große Fülle von Werkzeugen entwickelt werden kann.

Darauf aufbauend dient das *Interchange*-Profil dazu, einen Teil des kompletten Funktionsumfangs von X3D auszuwählen, der den einfachen Austausch von Daten zwischen Autorensystemen ermöglichen soll. Zudem sollen hier die Funktionalitäten herausgegriffen werden, die auch Werkzeuge umsetzten können, die mit der benötigten Rechenleistung für komplexe Berechnungen im Bereich von X3D überfordert wären.

Das *Interactive*-Profil erweitert die einfacheren Profile um solche Funktionalitäten, die aufwendigere Darstellungen und Interaktivität mit dem dreidimensionalen Modell ermöglichen.

Das *Immersive*-Profil legt seinen Fokus auf die Darstellungen von Szenen in virtuellen Umgebungen. Dafür werden alle Möglichkeiten benötigt, die X3D im Bereich der Interaktion und Navigation bietet.

Den maximalen Funktionsumfang besitzt das Profil *Full*, es enthält alle im X3D-Standard verfügbaren Fähigkeiten und Komponenten. Es ist damit naturgemäß das Profil, das die weitreichendsten Funktionen ermöglicht, das aber auch dementsprechend schwierig zu implementieren ist.

3.6.3 X3D-Browser

Die Aufgabe eines X3D-Browsers ist es, durch einen Parser die Datei, in der sich der Szenegraph befindet auszulesen und die Szene darzustellen. Dabei sollte auch der Ereignisgraph eingelesen und die dort beschriebenen Ereignisse verabreitet werden. Um X3D-Szenen zu betrachten, gibt es mittlerweile eine recht große Auswahl an X3D-Browsern. Diese unterscheiden sich zum Teil grundlegend. So gibt es für Firefox ein Plugin, das die Betrachtung von in Webseiten eingebetteten X3D-Szenen direkt im Browser ermöglicht. Beispiele für X3D-Browser sind der *instant player* aus dem *instant reality*-Framework (Fraunhofer IDG, 2010), der *Octaga Player* von der Firma *Octaga* (Octaga, 2010) und der *BS Contact* von der Firma *Bitmanagement* (Bitmanagement, 2010). McIntosh et al. (2005) nennen für die Wahl des geeigneten Browsers zwei Kernprobleme:

- nicht funktionierende X3D-Browser
- unvollständige Implementierung des Standards

Diese beiden Probleme kann auch der Autor der vorliegenden Arbeit bestätigen. Je nach Art des Betriebssystems oder in Abhängigkeit von verwendeten Treibern für die Grafikkarte funktionieren einige X3D-Browser entweder gar nicht oder nicht stabil genug. Oft sind auch keine Fehlerausgaben zu erkennen und einige Browser stürzen beim Laden eines Modells reproduzierbar ab, während andere Browser das gleiche Modell problemlos anzeigen.

Die meisten Browser implementieren nicht den gesamten Funktionsumfang des X3D-Standards. Zum einen ist dies vom Web3D Consortium so gewollt, um die Eintrittsbarrieren nicht zu hoch zu legen, andererseits sind so einige Möglichkeiten von X3D schlecht nutzbar. McIntosh et al. (2005) bezeichnet den Bereich der X3D-Browser auch als „minefield“ (dt. „Minenfeld“), weil man beim Erreichen des Ziel sehr stark mit dem Prinzip von Versuch und Irrtum in Berührung kommt. Als einen guten Ausgangspunkt für die Betrachtung von dreidimensionalen Modellen nennen McIntosh et al. (2005) den *BS Contact*. Dieser unterstützt eine Reihe der für die Softwarevisualisierung nützlichen Funktionen des Standards und läuft stabil.

Das Virtual Reality Labor ist in der Lage, Modelle im VRML-Format (Virtal Reality Markup Language) darzustellen. VRML kann als Vorgängerformat von X3D bezeichnet werden. Um die vom erstellten Generator erzeugten X3D-Modelle im Virtual Reality Labor darstellen zu können, kommt eine XSL-Transformation zum Einsatz.

4 Modellgetriebene Softwareentwicklung

4.1 Definition

Modellgetriebene Softwareentwicklung ist vor allem in den letzten Jahren zu einem wichtigen Paradigma im Bereich der professionellen Softwareentwicklung geworden. Als Definition für modellgetriebene Softwareentwicklung soll im Rahmen dieser Arbeit die Definition von Stahl et al. (2007) genutzt werden:

> *„Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen."*

Formale Modelle können hierbei unterschiedliche Ausprägungen haben. Die bekannteste Form von Modellen sind wohl die grafisch repräsentierten Modelle. Modelle können aber auch textueller Natur sein (vgl. Stahl et al., 2007). Als Beispiele sind hier Quellcode oder XML-Modelle zu sehen. Die abstrakte und die konkrete Syntax von Programmiersprachen werden von der jeweiligen Sprachspezifikation definiert. Bei XML-Modellen können abstrakte und konkrete Syntax zum Beispiel durch XML-Schema definiert werden. Als Beispiele für grafische Modelle dienen die verschiedenen Formen der UML-Diagramme. Die Definition fordert dabei auch, dass Modelle formal sein müssen, um sie für die modellgetriebene Entwicklung nutzen zu können. Um diese Forderung einzuhalten, müssen sie verschiedene Eigenschaften besitzen: (vgl. Stahl et al., 2007):

- Das Modell muss zumindest einen Teilbereich der Software abschließend beschreiben können.
- Es muss ein Metamodell vorhanden sein, das die Domäne beschreibt, deren Inhalte von dem Modell spezifiziert werden.
- Das Modell muss einer domänenspezifischen Sprache gehorchen und damit deren spezifische und abstrakte Syntax sowie ihre statische Semantik einhalten.

Die abstrakte Syntax unterscheidet sich von der konkreten Syntax dahingehend, dass die abstrakte Syntax die Elemente der Sprache und ihre Beziehungen beschreibt, während die konkrete Syntax ausdrückt, wie diese innerhalb der Sprache beschrieben werden.

Die statische Semantik dient dazu, den Bereich, den ein Modell beschreiben kann, durch Bedingungen einzuschränken. Hierdurch können nur verwendbare Modelle im Rahmen des modellgetriebenen Entwicklungsprozesses genutzt werden und Fehler können nicht erst zur Laufzeit entstehen (vgl. Stahl et al., 2007).

Die modellgetriebene Vorgehensweise besitzt verschiedene Vorteile (vgl. Pietrek und Trompeter, 2007; Stahl et al., 2007):

- Produktivitätssteigerung

- Aufwandsreduktion
- Qualitätsverbesserung
- Flexibilität
- Portabilität
- Abstraktion
- Wiederverwendung
- einheitliche Architektur

Die Gründe für diese Vorteile sind sehr vielfältig und einige der Vorteile verstärken sich dabei. Diese Vorteile müssen natürlich in gewisser Weise auch erkauft werden, sie stellen sich nicht einfach ein. Die Kosten für diese Vorteile sind die Aus- oder Weiterbildung der Mitarbeiter, die Anpassung des Vorgehens und die Beachtung der Prinzipien der modellgetriebenen Softwareentwicklung. Das modellgetriebene Paradigma ist auch nicht für alle Einsatzgebiete geeignet. Es spielt seine Vorzüge vor allem bei Produkten mit langem Lebenszyklus und bei Produkten mit gleicher Architektur aus. Bei Software mit langem Lebenszyklus kommt der Vorteil durch die Möglichkeit zur effektiveren Wartung zustande. Bei Softwaresystemfamilien spielen die Möglichkeiten zur Wiederverwendung eine große Rolle.

Für die Zwecke dieser Arbeit soll nur ein Teilbereich der modellgetriebenen Softwareentwicklung verwendet werden, da es im Rahmen der Prototypentwicklung (Kap. 6) nicht darum gehen soll, eine lauffähige Software zu erzeugen, sondern eine dreidimensionale Visualisierung. Deswegen werden einige Bereiche der modellgetriebenen Softwareentwicklung in diesem Kapitel nur am Rande erwähnt und nicht näher aufgeführt, zum Beispiel die Modell-zu-Code-Transformationen.

4.2 Domänenarchitektur

Eine Domäne dient in der modellgetriebenen Softwareentwicklung dazu, den Bereich, für den sie entwickelt wird, von der Umwelt abzugrenzen und somit einen Rahmen zu schaffen. Domänen können dabei aus unterschiedlichen Blickwinkeln gesehen werden. Ein JavaEE-Entwickler wird eine Domäne eher mit Begriffen wie „Session Bean“ u. a. beschreiben. Ein Fachexperte, zum Beispiel ein Bibliothekar, der die Entwicklung eines Ausleihsystems unterstützen soll, wird eher in Kategorien wie, „Buch“ oder „Ausleiher“ denken. Diese unterschiedlichen Sichtweisen spiegeln sich in der Unterscheidung von Domänen in technische und fachliche Domänen wider (vgl. Stahl et al., 2007). Die Domänenarchitektur soll eine solche Domäne in ihrer Gesamtheit beschreiben und formalisieren. Dazu gehören sowohl domänenspezifische Begrifflichkeiten und Sprachen, als auch ein Metamodell, die passenden Transformationen und die Plattform, die die technischen Grundlagen bereitstellt.

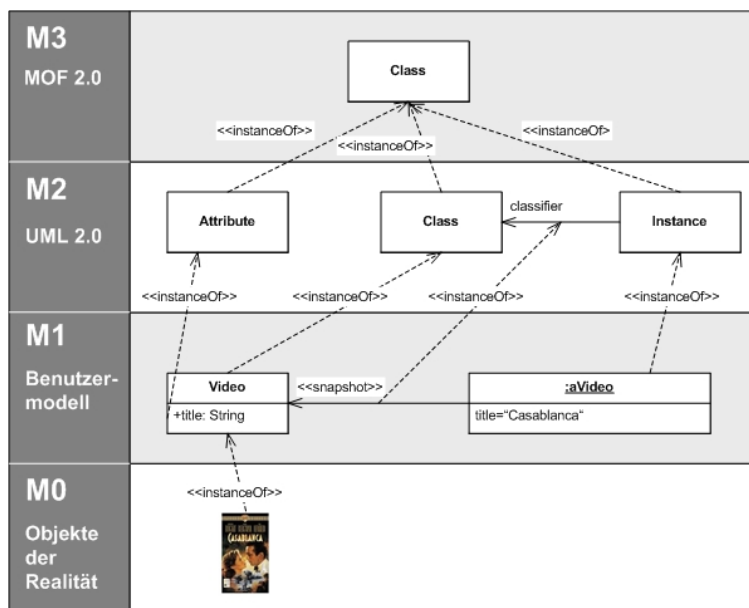

Abbildung 4.1: Hierarchie der Metamodellierung (Quelle: Wikipedia, 2008)

4.2.1 Metamodelle

Metamodelle stehen im Abstraktionsniveau über den Modellen. Sie definieren, wie die Konzepte, die den Modellen zugrunde liegen, zueinander in Beziehung stehen. Des Weiteren definieren sie die abstrakte Syntax und die Semantik von domänenspezifischen Sprachen (vgl. Pietrek und Trompeter, 2007). Pietrek und Trompeter (2007) nennen als Einsatzgebiete von Metamodellen:

- Definition von domänenspezifischen Sprachen
- Modellvalidierung
- Modelltransformation
- Codegenerierung
- Werkzeugintegration

Hier zeigt sich die große Bedeutung von Metamodellen für die Zwecke der modellgetriebenen Softwareetwicklung. Abbildung 4.1 zeigt die von der Object Modelling Group (OMG) definierten Schichten der Metamodellierung. Die M3-Ebene definiert die Metametamodelle, auf die in Abschnitt 4.2.2 näher eingegangen wird.

4.2.2 Metametamodelle

Metametamodelle sind die Metamodelle von Metamodellen. Folglich könnte auch eine M4-Ebene definiert werden, deren Instanzen dann die Metametamodelle wären. Um diesen Kreislauf zu durchbrechen, definieren sich Metametamodelle meist selbst und stellen somit die höchste Abstraktionsschicht dar. Weit verbreitete Metametamodelle sind die Meta Object Facility (MOF) der OMG, die auch in Abb. 4.1 dargestellt wird, und Ecore. MOF ist die

Grundlage der UML. Ecore ist das Metamodell des Eclipse Modelling Frameworf (EMF) und wird auch im Rahmen des openArchitectureware-Frameworks sehr rege genutzt (vgl. Stahl et al., 2007).

4.2.3 Domänenspezifische Sprachen

Der Begriff der domänenspezifischen Sprache (engl. *domain specific language*, DSL) wird von Pietrek und Trompeter (2007) wie folgt definiert:

> *„Eine DSL ist eine Modellierungs-, Programmier- und/oder Spezifikationssprache, die sich auf die abstrakte Darstellung von Sachverhalten einer fachlichen und/oder technischen Domäne beschränkt und dies mit domänennahen Sprachmitteln effizient ermöglicht.“*

Der Begriff der Programmiersprache, der in dieser Definition auftaucht, ergibt sich aus der Möglichkeit, domänenspezifische Sprachen in der Informatik auch für spezielle Einsatzgebiete außerhalb der modellgetriebenen Softwareentwicklung zu verwenden. Als Beispiele hierfür sind die regulären Ausdrücke zu nennen, die auch als domänenspezifische Sprache bezeichnet werden können (vgl. Pietrek und Trompeter, 2007).

Domänenspezifische Sprachen können in unterschiedlichen Ausprägungen im Rahmen der modellgetriebenen Softwareentwicklung eingesetzt werden. Die drei möglichen Varianten sind dabei die grafische DSL, die textuelle DSL und die dialogbasierte DSL.

Bei der grafischen DSL beschreibt der Modellierer die Sachverhalte mittels grafischer Elemente. Als Beispiel kann hier die Modellierung mittels UML-Diagrammen und deren Profilen dienen, die von der OMG im Rahmen des Model Driven Architecture (MDA) Standards vorgeschlagen wird. Die OMG ist ein Konsortium, das sich zum Ziel gesetzt hat, verschiedene Sachverhalte im Gebiet der objektorientierten Programmierung zu standardisieren. Das wohl bekannteste Beispiel einer OMG-Standards ist die UML-Spezifikation. MDA ist ein weiterer Standard der OMG, der sich vor allem die Interoperabilität auf die Fahnen geschrieben hat und der auch zum modellgetriebenen Entwickeln einsetzbar ist (vgl. Stahl et al., 2007).

Bei einer textuellen DSL werden die spezifischen Eigenschaften des gewünschten Produkts mittels einer textuellen Sprache beschrieben. Dies mag auf den ersten Blick seltsam anmuten, da ein Text schwerer zu überblicken scheint als eine geeignete grafische Darstellung. Die Beschreibung mittels textueller DSLs hat aber mehrere Vorteile. Zum Beispiel können textuelle Modelle deutlich einfacher automatisch geprüft werden als dies bei grafischen Modellen möglich ist. Stahl et al. (2007) nennen zudem die Möglichkeit zur Tastaturbedienung als Vorteil. Mit der Möglichkeit das Modell komplett tastaturgestützt erstellen zu können, geht bei geübten Modellierern eine Produktivitätsteigerung einher. Stahl et al. (2007) führen als Weiteren Vorteil auf, dass Texte für die Verwaltung und Bearbeitung durch Tools, wie Diff, Grep oder Subversion zugänglich sind. Die Nachteile von textuellen DSLs können durch technische Hilfsmittel innerhalb der Editoren gemildert werden. Als

Beispiele für solche Hilfsmittel sind Code-Folding-Mechanismen und Code-Completion-Machnismen zu nennen, die Entwicklungsumgebungen wie Eclipse bereitstellen. Im Rahmen des openArchitectureWare-Framework, auf welches in Abschnitt 6.3.2 näher eingegangen wird, steht als Hilfmittel das XText-Framework bereit. Dies ermöglicht es, textuelle DSLs zu entwerfen.

Dialogbasierte DSLs führen den Nutzer mit Hilfe von Dialogen durch den Prozess der Modellierung. Häufig sind diese Dialoge mehrstufig und bieten verschiedene Wege durch den Prozess, je nachdem, welche Entscheidungen der Nutzer trifft.

DSLs können zudem in interne und externe DSLs unterschieden werden. Diese Differenzierung geht mutmaßlich auf einen Blogeintrag von Martin Fowler zurück (vgl. Fowler, 2010; Stahl et al., 2007). Fowler (2010) beschreibt interne DSLs als Wege, eine Ursprungssprache so zu nutzen, dass eine neue domänenspezifische Sprache entsteht. Hierbei wird mit den spracheigenen Mittel der Hostsprache eine neue eigenständige Sprache geschaffen. Ein Beispiel hierfür sind C++-Expression-Templates. Im Gegensatz dazu stellen externe DSLs eigenständige Sprachen dar. Sie können textuell, dialogbasiert oder grafisch sein und benötigen eine selbstständige Definition und Infrastruktur. Dazu gehören die Definition von Syntax, Semantik und die Bereitstellung von Werkzeugen zur Verarbeitung der neuen Sprache.

4.2.4 Modelltransformationen

Modelltransformationen sind ein zentraler Bestandteil der modellgetriebenen Softwareentwicklung. Im Rahmen des MDA-Ansatzes werden dabei vor allem die Syntax, die Semantik und die Darstellung plattformunabhängiger Modelle (Plattform Independent Models, PIMs) und plattformspezifischer Modelle beschrieben. Um aus diesen wiederum Code erzeugen zu können, muss es auch möglich sein, die PIMs in PSMs umwandeln zu können, da erst aus den plattformspezifischen Modellen Code erzeugt werden kann (vgl. Pietrek und Trompeter, 2007; Czarnecki und Helsen, 2003). Um Möglichkeiten zu schaffen, diese Umwandlungen vorzunehmen, wurden verschiedene Transformationssprachen geschaffen. Für den MDA-Ansatz empfiehlt die OMG dabei die Einhaltung der Query View Transformation Spezifikation (QVT). QVT bietet dabei zwei Transformationssprachen, von denen die eine deklarativ und die andere imperativ ist. Eine detailliertere Übersicht zu QVT bieten Stahl et al. (2007). Es gibt allerdings noch weitere bekannte Transformationssprachen. Im Rahmen dieser Arbeit kommt vor allem die Sprache Xtend zum Einsatz, welche im Rahmen des openArchitectureWare-Frameworks entwickelt wird.

Im Bereich der modellgetriebenen Softwareentwicklung dienen die Modelltransformationen vor allem der Überbrückung von Abstraktionsebenen und der Umwandlung von abstrakteren Modellen, um sie für die Codegenerierung nutzen zu können. Um dies zu ermöglichen, ist es meist nötig, die Modelle vor einer Transformation zu validieren (vgl. Stahl et al., 2007). Im Rahmen dieser Arbeit ist eine solche Transformation vom Modell zum Code nicht erforderlich, weswegen Transformationsarten, namentlich Modell-zu-Modell-Transformationen, zum Einsatz kommen, die die Verfeinerung auf höheren Abstraktions-

ebenen durchführen.

Stahl et al. (2007) trennen im Bereich der Modelltransformationen zusätzlich Modellmodifikationen und -transformationen. Der Unterschied zwischen beiden Varianten besteht darin, dass bei Modellmodifikationen das Ursprungsmodell verändert wird, wohingegen bei Modelltransformationen aus einem Modell ein neues erzeugt wird und die Änderungen nur an diesem neuen Modell nachvollziehbar sind. Ein zweiter Unterschied besteht darin, dass das Metamodell gleich bleibt, da nur das Modell verändert wird.

Im Rahmen der Arbeit kommen Modellmodifikationen nicht zum Einsatz, an einigen Stellen wären sie aber grundsätzlich einsetzbar. Der Autor hat sich bewusst dazu entschieden, keine Modelltransformationen durchzuführen, sondern statt dessen äquivalente Modell-zu-Modell-Transformationen zu verwenden. Der Grund hierfür ist, dass die einzelnen Transformationsschritte auf diese Weise nachvollziehbar bleiben. In den folgenden beiden Abschnitten sollen die beiden im Rahmen dieser Arbeit wichtigen Transformationsarten erläutert werden.

4.2.4.1 Modell-zu-Modell-Transformationen

Im Rahmen dieser Arbeit wird der Begriff der Modell-zu-Modell-Transformationen sehr eng gefasst verwendet. Der Grund dafür ist, dass der Autor zwischen Modelltransformationen, die genau ein Modell in ein anderes umwandeln, und Modelltransformationen, die zwei oder mehr Modelle in ein anderes umwandeln, differenzieren möchte. Diese Differenzierung soll Abb. 4.2 zeigen.

Modell-zu Modell-Transformation dienen im Rahmen dieser Arbeit dazu, genau ein Modell in ein anderes zu überführen. Die jeweiligen Modelle müssen dabei nicht demselben Metamodell gehorchen. Meist werden Modell-zu-Modell-Transformationen durchgeführt, um eine tiefere Abstraktionsebene zu erreichen, also um das Modell zu verfeinern. Dies kann dadurch nötig werden, dass man mehr als zwei mögliche Abstraktionsebenen nutzen

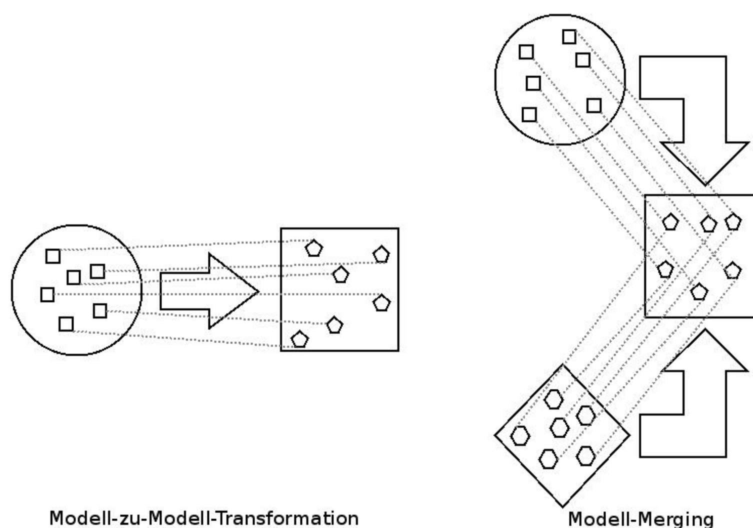

Abbildung 4.2: Modelltransformationen (in Anlehnung an: Stahl et al., 2007)

möchte. Bei zwei Abstraktionsebenen reicht die Ausführung einer Transformation im Rahmen der klassischen modellgetriebenen Softwareentwicklung aus. Es wird nur die Lücke zwischen Modell und Code überbrückt. Dies kann aber zu Nachteilen führen (vgl. Stahl et al., 2007). Zum Beispiel kann die Modelltransformation in solchen Fällen sehr komplex werden.

4.2.4.2 Modell-Merging

Modell-Merging wird von Stahl et al. (2007) als Spezialisierung der Modell-zu-Modell-Transformation angesehen. Im Rahmen dieser Arbeit besitzen allerdings Transformationen, mit denen aus zwei oder mehr Modellen ein Zielmodell erzeugt wird, besondere Bedeutung, zum Beispiel beim Zusammenführen von Metrik- und Strukturinformationen. Um Informationen aus unterschiedlichen Modellen zusammenzuführen, ist es erforderlich, dass es in jedem der Modelle eindeutige Primärschlüssel gibt, die einen Abgleich erlauben, sodass die Informationen in dem zu erstellenden Modell den richtigen Elementen zugewiesen werden. Auch beim Modell-Merging müssen die Modelle nicht Instanzen des gleichen Metamodells sein.

Modell-Merging ist eng mit dem sogenannten Modell-Weaving verwandt, bei dem mehrere Modelle miteinander verbunden werden. Das Modell-Weaving, auch Linking genannt, ist ein Spezialfall der Modellmodifikation. Es entsteht kein neues Zielmodell. Vielmehr werden Referenzen genutzt, um verschiedene Elemente von mehreren Modellen miteinander zu koppeln und sie somit miteinander zu verbinden (vgl. Stahl et al., 2007).

5 Generative Programmierung

5.1 Definition

In seiner Diplomarbeit zeigte Müller auf, dass das Paradigma der generativen Programmierung auch auf die automatische Generierung einer Softwarevisualisierung angewendet werden kann (vgl. Müller, 2009). Da der im Rahmen dieser Arbeit erstellte Generator auf der Arbeit von Müller aufbaut, sollen an dieser Stelle auch die Konzepte der generativen Softwareentwicklung und die Adaption dieser Konzepte für die Zwecke der Softwarevisualisierung vorgestellt werden.

Zur Erklärung der generativen Programmierung soll im Rahmen dieser Arbeit die Definition von Czarnecki und Eisenecker (2000) dienen:

> *„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.“*

Im Bereich der generativen Programmierung kommt somit vor allem den Softwaresystemfamilien große Bedeutung zu. Softwaresystemfamilien stellen eine Menge von Softwareprodukten dar, die alle aus einer gemeinsamen Basis von Implementierungskomponenten erstellt werden können. Die generative Programmierung versucht nun darauf aufbauend diese Softwaresystemfamilien so zu gestalten, dass die Erzeugung von Einzelprodukten aus dieser Menge von Produkten automatisiert werden kann. Die dafür verwendeten Komponenten sollen maximal kombinierbar und möglichst nicht redundant sein. Komponenten stellen eine Art Baustein zur Produktion von Produkten dar. Sie sind auch stets prozessspezifisch, es können also in einem Prozess Komponenten geeignet sein, die in einem anderen nicht verwendet werden können (vgl. Czarnecki und Eisenecker, 2000).

Um die Variabilität von Softwaresystemfamilien darzustellen, kommen meist Merkmalmodelle zum Einsatz. Ein Merkmalmodell besteht aus einem Merkmaldiagramm und zusätzlichen Informationen, wie zum Beispiel Beschreibungen der Merkmale. Merkmaldiagramme werden meist mit Hilfe einer modifizierten Feature Oriented Domain Analysis (FODA)-Notation beschrieben. Damit können zum Beispiel Auswahlmöglichkeiten zwischen zwei und mehr Merkmalen sowie optionale Merkmale eines Softwaresystems beschrieben werden (vgl. Czarnecki und Eisenecker, 2000).

Als Eingangsdokument wird eine Anforderungsspezifikation, die mittels einer domänenspezifischen Sprache beschrieben ist, benötigt. Auf diese wurde bereits im vorangehenden Kapitel näher eingegangen. Eine domänenspezifische Sprache muss dementsprechend problemorientiert und spezialisiert sein (vgl. Czarnecki und Eisenecker, 2000).

Wenn die Anforderungen an das Zielprodukt bekannt sind, kommt ein Generator zum

Einsatz, der die Anforderungsspezifikation verarbeitet, prüft, eventuell vervollständigt und dann die Generierung aus elementaren Komponenten möglichst optimiert durchführt.

Die Komponenten, die domänenspezifische Sprache und der Generator sind Produkte, die im Rahmen der Domänenentwicklung entstehen und im Rahmen der Anwendungsentwicklung genutzt werden. Diese Trennung der einzelnen Prozesse soll im folgenden Abschnitt näher betrachtet werden.

5.2 Generatives Domänenmodell

Czarnecki und Eisenecker (2000) definieren eine Domäne als ein Wissensgebiet, welches die Anforderungen der Interessensbeteiligten optimal umsetzt. Zudem umfasst eine Domäne die von den Domänenexperten genutzten Begriffe und Konzepte sowie das Wissen darüber, wie Softwaresysteme in diesem Gebiet erstellt werden.

Eine im Bereich der generativen Programmierung genutzte Domäne unterteilt sich in drei Unterbereiche: den Problemraum, das Konfigurationswissen und den Lösungsraum. Dieses generative Domänenmodell (Abb. 5.1) dient dazu, die einzelnen Familienmitglieder einer Softwaresystemfamilie automatisch generieren zu können (vgl. Czarnecki und Eisenecker, 2000). Die in Abb. 5.1 gezeigte Anordnung kann dabei noch variiert werden. So ist es möglich, dass der Lösungsraum gleichzeitig der Problemraum für einen weiteren Lösungsraum darstellt und somit eine verkettete Abbildung entsteht. Des Weiteren sind multiple und alternative Problem- beziehungsweise Lösungsräume denkbar. Multiple Problem- beziehungsweise Lösungsräume werden durch das gleiche Konfigurationswissen abgebildet, alternative Problem- beziehungsweise Lösungsräume werden jeweils durch ein eigenständiges Konfigurationswissen abgebildet (vgl. Czarnecki, 2005).

5.2.1 Problemraum

Der Problemraum stellt die notwendigen Mittel bereit, um ein Zielsystem aus der Menge der Softwaresystemfamilie spezifizieren zu können. Aus diesem Grund enthält der Problemraum

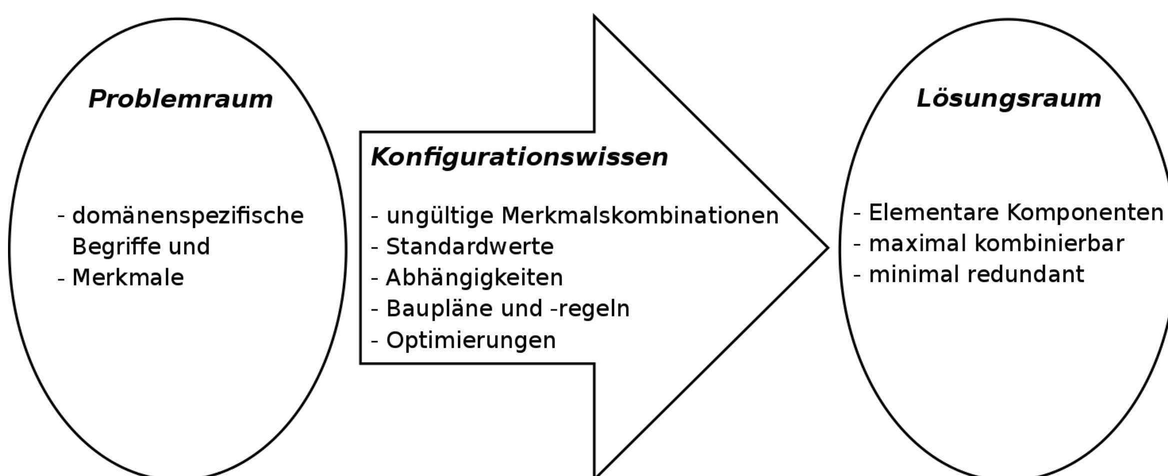

Abbildung 5.1: Generatives Domänenmodell (nach: Czarnecki und Eisenecker, 2000)

die domänenspezifischen Begriffe und Merkmale (vgl. Czarnecki und Eisenecker, 2000). Die Verwendung von Domänenspezifika ermöglicht Domänenexperten, ein System in einer ihnen vertrauten Weise zu bestellen (vgl. Czarnecki, 2005).

5.2.2 Konfigurationswissen

Das Konfigurationswissen hilft bei der Abbildung von Konzepten des Problemraums auf den Lösungsraum. Dazu muss es die Regeln umfassen, nach denen diese Abbildung ablaufen soll. Zusätzlich dazu kann das Konfigurationswissen auch ungültige Merkmalkombinationen, Standardwerte und verschiedene Abhängigkeiten enthalten. Ein Beispiel für eine solche Abhängigkeit ist folgender Sachverhalt: wenn Merkmal A selektiert ist, wird automatisch Merkmal B ausgewählt. Diese Regeln sollen verhindern, dass ein ungültiges System spezifiziert und erstellt wird (vgl. Czarnecki und Eisenecker, 2000).

5.2.3 Lösungsraum

Im Lösungsraum sind die verschiedenen Komponenten enthalten, die es ermöglichen, die einzelnen Systeme zusammenzustellen. Diese müssen so gestaltet sein, dass sie mit möglichst vielen weiteren Komponenten zusammenarbeiten können. Zudem sollten die Komponenten möglichst wenig Redundanz aufweisen. Die Trennung zwischen Problemraum und Lösungsraum ermöglicht es, in beiden Räumen unterschiedliche Schwerpunkte zu setzen und sie getrennt voneinander zu entwickeln (vgl. Czarnecki, 2005).

5.3 Technikprojektion

Die verschiedenen Elemente des generativen Domänenmodells können durch unterschiedliche Techniken realisiert werden. Hierfür kann auch eine einzelne Technologie alle Elemente des generativen Domänenmodells abbilden. Dies ist zum Beispiel bei der C++-Template-Metaprogrammierung der Fall. Hierbei dient C++ als Technologie, um den Problemraum, das Konfigurationswissen und den Lösungsraum gleichermaßen zu realisieren (vgl. Czarnecki, 2005).

5.4 Prozessmodell

Das generative Prozessmodell differenziert zwei verschiedene Prozesse, die sich gegenseitig beeinflussen, die Domänenentwicklung und die Anwendungsentwicklung. Der Grund für diese strikte Trennung ist die notwendige Trennung zwischen der Entwicklung *für* Wiederverwendung und die Entwicklung *mit* Wiederverwendung (vgl. Czarnecki, 2005). Einen Überblick über die beiden Prozesse gibt Abb. 5.2. Zusätzlich zu diesen beiden Prozessen kommt meist noch ein dritter Prozess hinzu, das Management der Prozesse (vgl. Czarnecki, 2005). Als übergeordneter Prozess wird es in der Darstellung nicht aufgeführt. Die beiden folgenden Abschnitte sollen die zwei Hauptprozesse näher beleuchten.

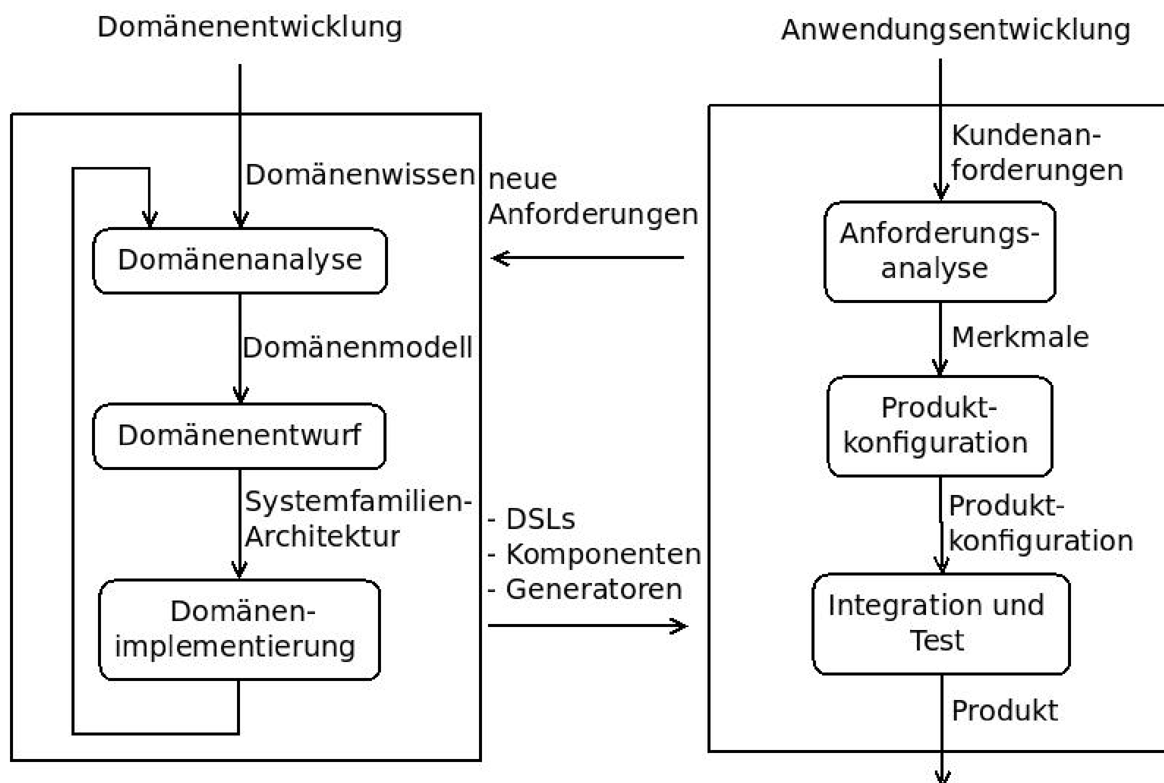

Abbildung 5.2: Prozesse der generativen Entwicklung (nach: Czarnecki und Eisenecker, 2000; Czarnecki, 2005)

5.4.1 Domänenentwicklung

Die Domänenentwicklung stellt einen Entwicklungsprozess *für* Wiederverwendung dar. Das Ziel dieses Prozesses ist die Bereitstellung einer Basis für die Anwendungsentwicklung. Dazu gehören domänenspezifische Sprachen, Komponenten und Generatoren.

Die Domänenanalyse soll die Domäne, in der sich die Softwaresystemfamilie befindet, definieren und beschreiben. Die im Rahmen dieser Analyse gefundenen und ausgewählten Informationen über die Domäne sollen dann in ein einheitliches Domänenmodell einfließen (vgl. Czarnecki und Eisenecker, 2000). Zu diesen Informationen gehört auch die Variabilität, die der Domäne innewohnt.

Die Domänenentwicklung nutzt die Ergebnisse aus der Domänenanalyse, um eine geeignete Systemfamilienarchitektur zu finden, mit der alle möglichen Einzelsysteme der Systemfamilie spezifiziert und gebaut werden können (vgl. Czarnecki, 2005).

Die Domänenimplementierung soll nun die so erarbeiteten Architekturen, domänenspezifischen Sprachen, Produktionspläne und Komponenten implementieren und zueinander passend konfigurieren. Daran anschließend werden sie der Anwendungsentwicklung bereitgestellt, sodass die Anwendungsentwicklung durchlaufen werden kann.

5.4.2 Anwendungsentwicklung

Bei der Anwendungsentwicklung wird die Entwicklung *mit* Wiederverwendung durchgeführt. Dazu werden die verschiedenen von der Domänenentwicklung bereitgestellten Elemente ge-

nutzt, um ein Produkt zu erstellen, das den Kundenanforderungen entspricht. Dazu müssen diese Anforderungen naturgemäß bekannt sein. Deswegen erfolgt zu Beginn der Anwendungsentwicklung zunächst eine Analyse der Anforderungen. Dazu gehört auch die Überprüfung, ob die Anforderungen mit den Möglichkeiten der im Rahmen der Domänenentwicklung entwickelten Systemfamilie umsetzbar sind. Falls dies nicht der Fall sein sollte, bestehen zwei Möglichkeiten, um diese Anforderungen dennoch in das Produkt einfließen zu lassen: die separate Umsetzung im Rahmen der Anwendungsentwicklung und die Weitergabe der Anforderungen an die Domänenentwicklung. Die separat umgesetzten Anforderungen sollten aber grundsätzlich später auch in die Domänenentwicklung einfließen, um zukünftig auch Systemfamilienmitglieder mit dieser Anforderung auswählen zu können (vgl. Czarnecki und Eisenecker, 2000). Die Spezifikation des gewünschten Systemfamilienmitglieds erfolgt mit Hilfe von domänenspezifischen Sprachen.

Aus den Anforderungen, die im Rahmen der Anforderungsanalyse gewonnen werden, wird dann ein Satz von Merkmalen abgeleitet, die das Zielprodukt besitzen soll. Mit dieser Information kann dann ein passendes System bestellt werden. Dieses System wird dann im Rahmen der Integration „gebaut“. Dieser Prozess kann entweder manuell oder automatisch ablaufen und wird im Rahmen der generativen Softwareentwicklung meist automatisch mit Hilfe von Generatoren durchgeführt (vgl. Czarnecki, 2005). Abschließend sollte das erstellte System noch geeigneten Tests unterworfen werden, bevor es dann ausgeliefert wird.

5.5 Verhältnis zur modellgetriebenenen Softwareentwicklung

Stahl et al. (2007) ordnen die generative Softwareentwicklung bezüglich der Ontologie als Spezialform der modellgetriebenen Softwareentwicklung ein. Es wird auch deutlich, dass einige Begrifflichkeiten sich in beiden Paradigmen wiederfinden.

Die Unterschiede zur „klassischen“ modellgetriebenen Softwareentwicklung sind aber recht groß. Bei der generativen Softwareentwicklung steht vor allem die formale Definition der Softwaresystemfamilie im Vordergrund, dieser Fokus ist bei der modellgetriebenen Softwareentwicklung nicht vorhanden, sie konzentriert sich auf die Entwicklung von Einzelsystemen. Die generative Programmierung versucht bei der Modellierung einer Systemfamilie vollständig von Implementierungsprinzipien, wie Klassenvererbung oder Aggregation, zu abstrahieren. So können auch bei der Auswahl, welches dieser Prinzipien in einem speziellen Fall eingesetzt wird, Optimierungen durchgeführt werden. Damit scheiden aber einige Modellierungsarten, wie UML-Klassendiagramme, für die Definition einer Softwaresystemfamilie im Rahmen der generativen Programmierung aus. UML-Klassendiagramme setzen bei der Modellierung voraus, dass diese Entscheidungen schon getroffen wurden (vgl. Czarnecki und Eisenecker, 2000). Allgemein unterscheiden sich die verwendeten Werkzeuge zur Modellierung zwischen modellgetriebener Softwareentwicklung und generativer Programmierung. Sie müssen aber nicht unbedingt spezifisch nur für das jeweilige Paradigma geeignet sein, sondern können durchaus im anderen Verwendung finden. Zum Beispiel können

Merkmalmodelle nicht nur in der generativen Programmierung, sondern auch im Rahmen der modellgetriebenen Softwareentwicklung eingesetzt werden.

Beide Paradigmen haben somit gewisse Schnittmengen, zum Beispiel die Definition von domänenspezifischen Sprachen und die Verwendung von Generatoren zur Erstellung des gewünschten Produkts. Es gibt aber deutliche Unterschiede, vor allem die Konzentration auf Softwaresystemfamilien bei der generativen Programmierung und auf Einzelsysteme bei der modellgetriebenen Softwareentwicklung.

5.6 Adaption an die Softwarevisualisierung

Müller (2009) zeigt, dass sich das generative Paradigma adaptieren lässt, um auch auf die Softwarevisualisierung Anwendung zu finden. Um diese Adaption vorzunehmen, muss die Zielstellung der generativen Programmierung von einem „System“ hin zu einer „Visualisierung“ geändert werden. Der Prozess, der bei der Erstellung der Visualisierung durchlaufen wird, ist zudem ein Visualisierungsprozess und kann bei vollständiger Automatisierung als Spezialfall einer Generierung angesehen werden.

Nach dieser Änderung kann das generative Domänenmodell für die Zwecke der Softwarevisualisierung eingesetzt werden. Da das Ziel dieser Arbeit die gemeinsame Darstellung von Metrik- und Strukturinformationen ist, müssen zudem multiple Problemräume eingesetzt werden (Abb. 5.3).

Um dieses Modell in die Praxis überführen zu können, ist eine Technikprojektion der einzelnen Bestandteile nötig. Müller (2009) verwendete zur Beschreibung des Problemraums verschiedene XML-Schemata und Ecore als Metamodelle. Um die Struktur eines Softwaresystems automatisiert extrahieren zu können, ist es zusätzlich erforderlich, auch Modelle in

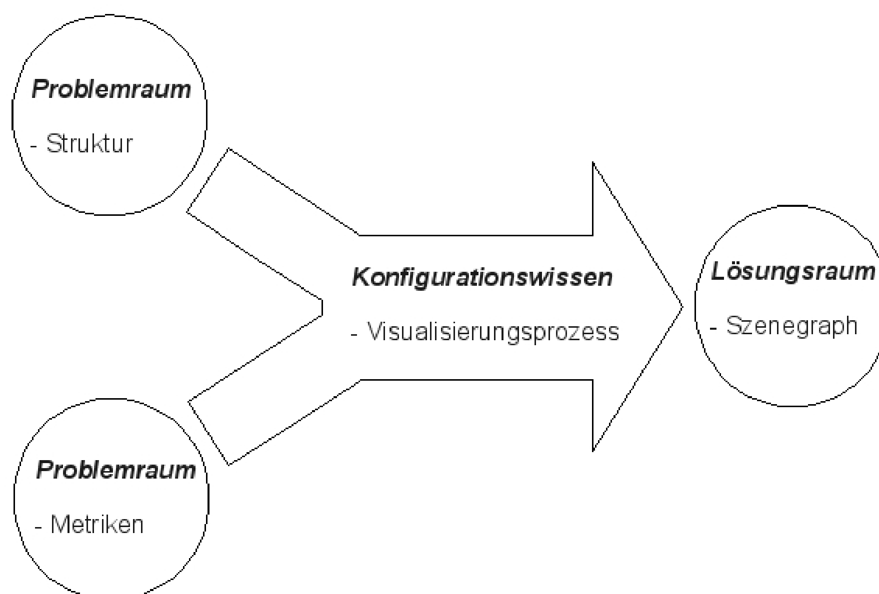

Abbildung 5.3: Generatives Domänenmodell zur Verbindung von Struktur und Metrikvisualisierung (nach: Müller, 2009; Czarnecki, 2005)

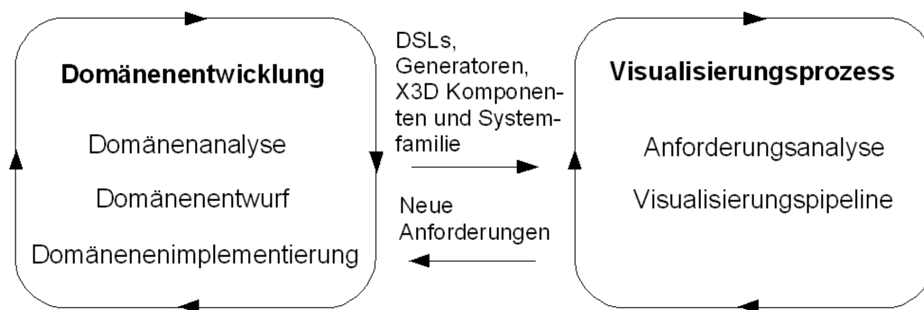

Abbildung 5.4: Prozessmodell zur Softwarevisualisierung in 3D (Quelle: Müller, 2009)

Form von Quellcode zuzulassen. Die abstrakte Syntax der jeweiligen Programmiersprache stellt hier ebenfalls ein Metamodell dar. Mit der konkreten Syntax der Programmiersprache wird dann darauf aufbauend die Kodierung von einzelnen Modellen definiert. Bei Java können zum Beispiel die Elemente der abstrakten Syntax wie Pakete, Klassen, Methoden und Attribute als Elemente eines Java-Metamodells verstanden werden. Ein Klasse `Generator` würde dann die Instanziierung des Metamodellelements `Klasse` darstellen. Zusätzlich muss auch eine domänenspezifische Sprache entwickelt werden, die es dem Nutzer des Visualisierungsgenerators ermöglicht, die gewünschte Visualisierung zu spezifizieren.

Müller (2009) verwendet als Techniken für das Konfigurationswissen verschiedene Tools aus dem openArchitectureWare-Framework. Vor allem kommen hierbei Modelltransformationstechniken zum Einsatz, die aus dem Bereich der modellgetriebenen Softwareentwicklung stammen.

Im Lösungsraum wählte Müller (2009) X3D zur Umsetzung. X3D bietet bereits eine Reihe von Elementen an, die im Rahmen eines generativen Prozesses zu einer fertigen Visualisierung zusammengestellt werden können. Somit übernehmen die einzelnen Elemente, die im X3D-Standard festgelegt wurden, die Rolle von elementaren Komponenten. Müller (2009) weist zudem darauf hin, dass der X3D-Standard auch Prototypen vorsieht. Dies sind aus elementaren Elementen zusammengestellte Vorlagen, die sich als gröbere Wiederverwendungseinheiten eignen.

Müller (2009) zeigt nach dieser Adaption des generativen Domänenmodells, dass sich auch das generative Prozessmodell mit kleinen Anpassungen auf die Softwarevisualisierung anwenden lässt (Abb. 5.4).

Im folgenden Kapitel soll die Entwicklung eines Generators vorgestellt werden, der diese Adaption der generativen Programmierung auf die Softwarevisualisierung unter Einbindung von statischen Codemetriken umsetzt.

6 Prototypentwicklung

6.1 Aufbau des Kapitels

Wie bereits in Kapitel 1 geschildert, baut der Prototyp auf der Arbeit von Müller (2009) auf, dessen Generator allerdings für die Zwecke dieser Arbeit an einigen Stellen angepasst und erweitert werden muss.

Abschnitt 6.3 soll hierbei die verwendeten Werkzeuge und Technologien aufzeigen und erläutern. Dabei soll an einigen Stellen auch kurz auf mögliche alternative Umsetzungsformen eingegangen werden, die nicht verwirklicht wurden. In Abschnitt 6.4 soll dann auf die vorgenommenen Erweiterungen zu der Arbeit von Müller (2009) eingegangen werden. Mit Abschnitt 6.5 schließen sich Erläuterungen zu den Metamodellen an, die im Rahmen des Prototyps verwendet werden. Dazu gehören sowohl bereits vorher definierte Metamodelle, als auch ein selbstdefiniertes Metamodellformat, welches die Abbildung von Metrikinformationen in einer Struktur ermöglicht. Diese entspricht der Struktur des Softwaresystems, dessen Metrikwerte gespeichert werden.

In Abschnitt 6.6 sollen dann zwei verschiedene Metaphern für Visualisierungen vorgestellt werden. Dabei soll zum einen jeweils der Algorithmus beschrieben werden, wie man von der Information über die Parametriesierung zum fertigen Bild kommt. Zum anderen soll auch untersucht werden, ob und inwieweit die Metaphern geeignet sind, um Softwaresysteme unter Berücksichtigung von Metriken darzustellen und wo die jeweiligen Schwächen der Metaphern liegen.

Abschnitt 6.7 soll dann die Entwicklung und die Verwendung der zwei verschiedenen domänenspezifischen Sprachen beleuchten, die entwickelt wurden, um die Konfiguration des Generators zu definieren.

Am Ende des Kapitels, in Abschnitt 6.8, sollen die Erstellung des Eclipse-Plugins und seine Bedienung kurz vorgestellt werden.

6.2 Ziele des Prototyps

Im Rahmen der vorliegenden Arbeit wurde ein Generator entwickelt, der es ermöglichen soll, automatisch ein Softwaresystem zu visualisieren und bei dieser Visualisierung verschiedene Parameter von Metrikinformationen beeinflussen zu lassen. Dies soll vor allem den Vergleich der zwei eingesetzten sowie die Bewertung von Möglichkeiten zur Darstellung verschiedener Informationen in einer Softwarevisualisierung ermöglichen. Zudem soll dabei gezeigt werden, dass dieser Schritt vollständig automatisiert ablaufen kann. Der Nutzer kann nach einer einmaligen Konfiguration der Art und Weise, wie er Informationen dargestellt haben möchte, „auf Knopfdruck“ eine Visualisierung erzeugen, ohne während des Visualisierungsprozesses weitere Einstellungen vornehmen oder manuelle Schritte durchführen zu müssen.

6.3 Verwendete Werkzeuge

6.3.1 Eclipse

Eclipse ist im Allgemeinen als Java-Entwicklungsumgebung bekannt. Diese Betrachtung greift aber deutlich zu kurz. Eclipse hat sich im Laufe der Zeit durch sein offenes Konzept zu einer Plattform entwickelt, mit der zum einen in vielen verschiedenen Programmiersprachen entwickelt und zum anderen sehr komplexe Rich-Client- sowie Server-Client-Anwendungen sehr effizient entwickelt werden können. So bietet Eclipse zum Beispiel mittlerweile durch Techniken wie das Graphical Modelling Framework Instrumente an, die es ermöglichen, komplexe grafische Editoren zu erstellen und zum Teil auch zu generieren.

Ursprünglich wurde Eclipse von der Firma IBM in den 90er Jahren des vorigen Jahrhunderts entwickelt, um die Entwicklungsumgebung Visual Age abzulösen. Man erkannte im Jahr 2001 die Perspektive, das Projekt unter Beteiligung der Open Source Gemeinde und unter Beteiligung anderer Firmen weiterzuführen und zu entwickeln. Seit 2004 befindet sich das Eclipse Projekt unter der Verwaltung der gemeinnützigen Eclipse Foundation, der sich mittlerweile viele andere Firmen angeschlossen haben. Darunter befinden sich auch viele bekannte Firmen wie *SAP*, *Oracle*, *Nokia* oder *Cisco*. Eine weitere Übersicht über die Mitglieder findet sich unter Eclipse Foundation (2010).

Einen wichtigen Beitrag zum Erfolg leistet sicherlich auch die Verwendung der Eclipse Public Licence für Eclipse und viele der für Eclipse entwickelten Plugins. Die Eclipse Public Licence ist eine Open Source Lizenz, die nicht den vererbenden Charakter der GNU Public Licence aufweist. Das bedeutet, dass Produkte, die unter Verwendung von Teilen von Eclipse erstellt wurden, nicht automatisch unter der Eclipse Public Licence verfügbar sein müssen. Sie können auch eine proprietäre Lizenz aufweisen. Das macht die Entwicklung mit und für Eclipse auch für Firmen interessant, die zu einem großen Teil ihre Produkte unter proprietären Lizenzen vertreiben möchten.

6.3.1.1 Eclipse Plugins

Eclipse basiert auf einem rudimentärem Kern und einem sehr mächtigen Plugin-Konzept. Der Kern besteht vor allem aus der Benutzungsoberfläche und der Bereitstellung des Plugin-Mechanismus. Ein Plugin stellt vereinfacht gesagt einen Container für Code dar. Es dient der Modularisierung des Codes und damit der Austausch- und Erweiterbarkeit. Ein weiteres wichtiges Konzept im Rahmen der Pluginentwicklung besteht im Konzept der Erweiterungspunkte (engl. *extension points*) und der dazu gehörigen Erweiterungen (engl. *extensions*). Diese beiden können mit einer Steckdose und einem Stecker verglichen werden. Hierbei stellt der Erweiterungspunkt die Steckdose dar, die die Voraussetzungen für den Betrieb liefert. Der Stecker mit der damit verbundenen elektrischen Anlage erweitert diese Steckdose um bestimmte Funktionalität (IBM Corporation, 2010). Dieser Mechanismus erlaubt eine möglichst lose Kopplung von Komponenten, da die Schnittstellen klar definiert sind. Ein Plugin kann sowohl Erweiterungspunkte als auch Erweiterungen definieren. Weitere wichti-

ge Konzepte des Plugin-Mechanismus sind das Merkmal (engl. *feature*) und das Fragment (engl. *fragment*). Ein Merkmal kapselt eine Gruppe von Plugins und Fragmenten um eine bestimmte Funktionalität zu liefern. Es beinhaltet eine Manifest-Datei. In dieser befinden sich die Informationen über den Inhalt des Merkmals. Ein Fragment dient dem Ersetzen und der Erweiterung des Codes eines bereits vorhandenen Plugins. Dies wird vor allem bei plattformspezifischem Code genutzt, sodass bei der Installation des Plugins anhand von Konfigurationseinstellungen automatisch das korrekte Fragment gefunden und installiert werden kann (vgl. IBM Corporation, 2010).

6.3.1.2 Java Model

Das *Java Model* ist eine von Eclipse automatisch erstellte leichtgewichtige Repräsentation eines Eclipse-Java-Projektes (vgl. Vogel, 2010). Es ist hierarchisch aufgebaut und dient unter Eclipse zum Beispiel der Darstellung der *Outline*, einer baumartigen Darstellung der Struktur des Quelltextes. Allgemein kann das *Java Model* gut dazu verwendet werden, um grundsätzliche Strukturinformationen eines Java-Projektes zu erhalten. Für diese Zwecke stellt Eclipse im Paket `org.eclipse.jdt.core` verschiedene Schnittstellen zur Laufzeit bereit. Die im Rahmen der Prototypentwicklung verwendeten Schnittstellen finden sich in Tabelle 6.3.1.2.

Eclipse bietet zudem mit den Technologien *Nature*, *Builder* und *Visitor* Techniken an, mit denen sich der abstrakte Syntaxbaum eines Eclipse-Java-Projektes auslesen lässt (vgl. Bach, 2009). Der abstrakte Syntaxbaum ist eine baumartige Struktur, die die Struktur eines Softwareprojektes widerspiegelt. Auf diese Weise könnten deutlich mehr Informationen zur Struktur eines Softwaresystems extrahiert werden, als dies mit dem *Java Model* möglich ist. Zum Beispiel könnte auf diese Weise auch die Kontrollflussstruktur extrahiert werden. Im Rahmen dieser Arbeit kamen diese Technologien nicht zum Einsatz, da die benötigten Informationen auch das *Java Model* liefern konnte. Für Weiterentwicklungen könnte es aber durchaus sinnvoll sein, auf diese Technologien zurückzugreifen.

Interface	Einsatzzweck
`IJavaProject`	repräsentiert das gesamte Java-Projekt und kann als Quelle für die Paketliste dienen
`IPackageFragment`	stellt ein einzelnes Paket dar
`ICompilationUnit`	repräsentiert eine Übersetzungseinheit
`IType`	stellt einen Java-Typ (Klasse, Schnittstelle, Enumeration usw.) dar; dient als Quelle für Methoden und Attribute
`IMethod`	bietet Informationen über eine Methode
`IField`	stellt Informationen zu einem Klassenattribut bereit
`IPackageFragmentRoot`	stellt das Wurzelelement für die Pakete eines `IJavaProject` dar

Tabelle 6.1: Verwendete *Java Model*-Schnittstellen

6.3.1.3 Metrics für Eclipse

Das Eclipse-Plugin *Metrics* ist in der Lage, für Java-Projekte 21 verschiedene Metriken zu erzeugen. Es liegt aktuell in der Version 1.3.6 vor. Eine Aufstellung der einzelnen Metriken findet sich in Tabelle 6.2.

Wenn die Erzeugung der Metriken für ein Java-Projekt über das Kontextmenü aktiviert wurde, steht eine zusätzliche Sicht zur Verfügung, die die Metriken anzeigt und dabei die kritischen Metriken einfärbt. Die Bestimmung der Metrikwerte wird bei einem erneuten Bauen des Projektes aktualisiert, damit der Nutzer stets über die aktuellen Werte verfügt. Zusätzlich verfügt das Plugin über ein grafisches Tool zur Analyse von Paketabhängigkeiten.

Metrik	Kürzel	Typ	Funktion
Abstractness	RMA	P	Anteil der abstrakten Klassen und Schnittstellen eines Paketes an der Gesamtzahl der Typen
Afferent Coupling	CA	P	Anzahl der Klassen außerhalb des Paketes, die von Klassen innerhalb des Paketes abhängen
Depth of Inheritance Tree	DIT	K	Abstand zur Klasse `Object` in der Vererbungshierarchie
Efferent Coupling	CE	P	Anzahl der Klassen innerhalb eines Paketes, die von Klassen außerhalb des Paketes abhängen
Instability	RMI	P	$CE/(CA+CE)$
Lack of Cohesion of Methods	LCOM	K	Mangel an Methodenkohäsion für eine Klasse nach der Henderson-Sellers-Methode
McCabe Cyclomatic Complexity	VG	M	zyklomatische Komplexität einer Methode nach McCabe (McCabe, 1976)
Method Lines of Code	MLOC	M	Anzahl der Codezeilen einer Methode
Nested Block Depth	NBD	M	maximale Blocktiefe einer Methode
Normalized Distance	RMD	P	$\lvert RMA+RMI-1\rvert$
Number of Attributes	NOF	K	Anzahl der Attribute einer Klasse
Number of Children	NSC	K	Anzahl der direkten Unterklassen einer Klasse
Number of Classes	NOC	P	Anzahl der Klassen in einem Paket
Number of Interfaces	NOI	P	Anzahl der Schnittstellen in einem Paket
Number of Methods	NOM	K	Anzahl der Methoden in einer Klasse
Number of Overridden Methods	NORM	K	Anzahl der überschriebenen Funktionen einer Klasse
Number of Parameters	PAR	M	Anzahl der Parameter einer Methode
Number of Static Attributes	NSF	K	Anzahl der statischen Attribute einer Klasse
Number of Static Methods	NSM	K	Anzahl der statischen Methoden einer Klasse
Specialization Index	SIX	K	$(NORM \times DIT)/NOM$
Weighted Methods per Class	WMC	K	Summe der zyklomatischen Komplexitäten für alle Methoden einer Klasse

Tabelle 6.2: Vom *Metrics*-Plugin bestimmbare Metriken (P = Paketmetrik, K = Klassenmetrik, M = Methodenmetrik) (vgl. Sauer, 2009)

Für die Extraktion der Metriken für den Generator wurde ein Tool gesucht, das eine Reihe von Metriken erzeugen kann und es ermöglicht, diese Erzeugung programmatisch anzustoßen. Das *Metrics*-Plugin erfüllt diese Anforderung durch die Bereitstellung mehrerer *Ant*-Tasks. *Ant* ist ein Tool zum automatischen Erzeugen von Programmen. Es wird über eine XML-Datei oder über seine Java-Schnittstelle gesteuert und führt nacheinander verschiedene Arbeitsschritte aus, um ein bestimmtes Endergebnis zu erreichen. Diese Arbeitsschritte werden „*Tasks*" genannt. *Ant-Tasks* können aber auch programmatisch durch Java aufgerufen und durchgeführt werden. Dieser Mechanismus wird im Rahmen des Prototyps genutzt, um eine XML-Datei mit den Metriken für ein Projekt zu erzeugen.

6.3.2 openArchitectureWare

OpenArchitectureWare (oAW) ist ein Framework, das im Rahmen der modellgetriebenen Softwareentwicklung diverse Möglichkeiten zur Modelltransformation und Codegenerierung bietet. Ursprünglich wurde openArchitectureWare von der b+m Informatik AG unter dem Namen „b+m Generator Framework" entwickelt. Diese stellte es 2003 auf der Internetplattform SourceForge unter dem Namen openArchitectureWare 3 unter die freie Eclipse Public License (vgl. Pietrek und Trompeter, 2007). Der Name oAW ist mittlerweile etwas irreführend, da die einzelnen Teilprojekte mittlerweile im Eclipse Modeling Project aufgegangen sind. Damit wäre auch eine Eingliederung der in diesem Abschnitt aufgeführten Techniken in Abschnitt 6.3.1 möglich. Um die Wichtigkeit von oAW für die Zwecke dieser Arbeit zu betonen, sollen die Fähigkeiten von oAW separat betrachtet werden. Der Name oAW soll beibehalten werden, um die verwendeten Kerntechnologien darunter subsumieren zu können, zudem firmieren die verwendeten Hilfsmittel in der verwendeten Version noch unter dem Namen openArchitectureWare. oAW beeinhaltet vier verschiedene Kernprojekte, deren Zusammenspiel die Möglichkeiten des Frameworks ausreizen. Diese vier Kernprojekte sind die Workflowsprache sowie die Sprachen XText, XPand und XTend. Auf die Möglichkeiten der Workflowsprache und XTend soll in den Abschnitt 6.3.2.1–6.3.2.2 näher eingegangen werden, da sie wichtige Grundlagen für den entwickelten Prototyp legen. XPand dient zur Definition von Templates, mit denen Generierungen von Quellcode- und anderen Testdateien definiert werden, die vom Workflow aufgerufen werden können. XText ist ein Werkzeug, das dem Nutzer die Definition domänenspezifischer Sprachen in Textform ermöglicht. Diese beiden Technologien kommen im Rahmen dieser Arbeit nicht zum Einsatz.

6.3.2.1 oAW-Workflow

In openArchitectureWare können die Abläufe, die im Rahmen einer modellgetriebenen Softwareentwicklung ablaufen sollen, mittels einer XML-basierten Sprache beschrieben werden. Hierbei können die einzelnen Aufrufe der unterschiedlichen Möglichkeiten, wie

- Modelltransformationen,
- Codegenerierungen,

- Lese- und Schreibaktionen für unterschiedliche Dateien
- usw.

orchestriert werden. Die im Rahmen des Prototyps verwendeten Elemente werden in Tabelle 6.3 aufgeführt. Den Aufruf einer solchen Komponente zeigt Listing 6.1.

Dieses Listing ruft eine Modelltransformation namens `setNames` auf, die sich in einem gleichnamigen Verzeichnis befindet. Dieses Verzeichnis liegt seinerseits in einem Verzeichnis, das durch die Variable `extensions.dir.metrics` definiert wird. Der Modelltransformation wird ein Modell übergeben, das in der Variable `rootXmlModel` abgelegt ist und das eine Instanz eines Metamodells darstellt. Dieses Metamodell wurde in der Variable `metrics` abgelegt. Das durch die Transformation entstehende Modell wird in der Variable `completeRootXmlModel` abgelegt und steht damit im weiteren Verlauf des Workflows zur Verfügung.

Die Modelltransformation dient dazu, in dem erstellten Modell aus den vollqualifizierten Bezeichnern des übergebenen Modells die darzustellenden Namen zu extrahieren. Diese Daten werden dann zusätzlich zu den bereits in dem Modell `rootXmlModel` vorhandenen Daten in dem neuen Modell abgelegt.

Um den Workflow einfach parametrisieren zu können, stehen innerhalb der Workflow-Sprache unterschiedliche Elemente bereit, die diese Aufgabe übernehmen. Eine Möglichkeit, um Parameter zu setzen, stellt das Auslesen einer Konfigurationsdatei dar. Dies kann mit der Komponente `TextConfigurationReader` umgesetzt werden. Eine andere Möglichkeit stellt die Verwendung der `property` dar, welche die zwei Attribute `name` und `value` besitzt und damit ein klassisches Name-Wert-Tupel definiert.

Weitere Dokumentation zur Workflow-Sprache findet sich unter Efftinge et al. (2008).

6.3.2.2 XTend

XTend ist eine funktionale Sprache, die im openArchitectureWare-Projekt dazu dient, Modelltransformationen zu definieren. Da XTend sowohl eine Verzweigungsanweisung als auch, durch die Möglichkeiten der Rekursion, ein Schleifenkonstrukt anbietet, handelt es sich um eine turingvollständige Programmiersprache. Durch das einheitliche Typsystem des oAW-Projekts arbeitet XTend sehr gut mit den anderen Werkzeugen, die oAW den Nutzern anbietet, zusammen. Es bietet durch einen ausgereiften Umgang mit Listen sehr mächtige,

```
<!-- set the names in the rootmodel-->
<component class="oaw.xtend.XtendComponent">
  <metaModel idRef="metrics"/>
  <invoke value="${extensions.
          dir.metrics}setNames::setNames(rootXmlModel)"/>
  <outputSlot value="completeRootXmlModel"/>
</component>
```

Listing 6.1: Aufruf einer Komponente

Komponente	Einsatzzweck
`TextConfigurationReader`	liest eine Textdatei als Konfiguration ein und speichert die gesetzten Werte für den Lauf des Workflows
`XMLReader`	liest ein Modell in Form einer XML-Datei ein; dazu muss ein Metamodell in Form einer XML-Schema-Datei angegeben sein
`XtendComponent`	ruft eine XTend-Modelltransformation auf und führt sie aus
`XMLWriter`	schreibt ein Modell als XML-Datei aus; dazu muss ein Metamodell in Form einer XML-Schema-Datei angegeben sein
`XmiWriter`	schreibt eine XMI-Datei aus; diese Komponente wird im Rahmen des Prototyps genutzt um ein Ecore-Modell zu schreiben
`CheckComponent`	ruft eine Check-Datei auf und prüft eine angegebene Datei mittels der dort definierten Prüfungen
`DTDComponent`	fügt eine DTD-Definition in eine XML-Datei ein
`XSLTransformerComponent`	ruft eine XSL-Transformation auf und führt sie auf einer XML-Datei aus

Tabelle 6.3: Verwendete Workflowkomponenten

aber dennoch relativ einfach zu handhabende Möglichkeiten zum Umgang mit unterschiedlichen Modellen. Zu den Modellen, mit denen XTend umgehen kann, gehören unter anderem Ecore-Modelle, aber auch XML-Dateien, die mit Hilfe von XML-Schema spezifiziert wurden. Vor allem die Fähigkeit, mit XML-Dateien umzugehen, macht XTend zu einem wichtigen Werkzeug im Rahmen dieser Arbeit. Für die Zwecke dieser Arbeit wird XTend ähnlich wie XSLT verwendet. Es ist allerdings nicht ganz so wortreich und bietet in der Verbindung mit Eclipse sehr ausgereifte Codevervollständigungsfunktionen, die die Arbeit mit diesem Werkzeug deutlich erleichtern.

Listing 6.2 stammt aus einer der im Rahmen des Prototyps verwendeten XTend-Transformationen. Aufgabe dieser Modelltransformation ist es, ein Modell, das dem Metamodell `metricXml` entspricht, mit einem Element `name` anzureichern, da zu dem Zeitpunkt nur die vollqualifizierten Namen in dem Modell gespeichert werden. Das Ergebnis dieser Anreicherung wird in einem neuen Modell gespeichert. Mittels der `import` Anweisung wird das entsprechende Metamodell geladen. Die Funktion `toPackage` liefert ein neues Objekt vom Typ `Package` und bekommt zwei Parameter übergeben, `pack` und `metrics`. Das mag verwirrend sein, da der Aufruf dieser Funktion mittels `metrics.package.toPackage(this)` erfolgt, wobei scheinbar nur ein Parameter übergeben wird. Dieser Aufruf ist allerdings nur eine alternative Schreibweise zu `toPackage(metrics.package,this)`.

```
import metricXml;
//...
create Metrics setNames(Metrics metrics):
  this.setPackage(metrics.package.toPackage(this));

create Package toPackage(Package pack, Metrics metrics):
  this.setFqn(pack.fqn)->
  this.setDepth(pack.depth)->
  if (pack.class.size > 0) then this.setClass(pack.class.
                 toClass(metrics))->
  if (pack.package.size > 0) then this.setPackage(pack.package.
     toPackage(metrics))->
  if(pack.eContainer.metaType==Metrics) then
    this.setName(pack.fqn)
   else
    this.setName(getName(pack.fqn,((Package)pack.eContainer).
        fqn));
//...
String getName(String elementfqn, String parentfqn):
        JAVA org.x3d.generator.Helper.getName(java.lang.String,
            java.lang.String);
```

Listing 6.2: XTend-Modelltransformation zum Zuordnen der gewählten Metriken zum jeweiligen Paket

In der Funktion `getName` sieht man den Aufruf einer Java-Methode, der zwei Parameter übergeben werden. Solche Erweiterungen sind vor allem dann sinnvoll, wenn Funktionen genutzt werden sollen, die in XTend nur sehr schwer umzusetzen sind.

6.4 Erweiterung der Visualisierungspipeline

Als Ausgangspunkt für die Visualisierung der Metriken soll der Generator von Müller (2009) dienen. Müller (2009) beschreibt in seiner Arbeit die Möglichkeiten zur automatisierten Visualisierung einer Software, deren Struktur durch ein Ecore-Modell definiert ist. Als Tool zur Positionierung und Größenbestimmung der einzelnen Objekte der Visualisierung nutzt er dabei die Software *WilmaGraph*, die verschiedene Darstellungsalgorithmen im dreidimensionalen Raum anbietet. Darunter befindet sich auch ein kraftgerichteter Algorithmus, den Müller (2009) nutzt, um eine Darstellung mittels der Schachtelungsmetapher zu generieren und die einzelnen Objekte auszurichten. Abbildung 6.1 zeigt die Visualisierung, die von Müller (2009) im Rahmen seiner Arbeit verwendet wird und die damit auch als Grundlage für die in dieser Arbeit verwendete Pipeline dient.

Um die Möglichkeiten der Beeinflussung der Visualisierung durch Metriken zu zeigen, wurde diese Pipeline an drei verschiedenen Punkten erweitert beziehungsweise modifiziert. Die einzelnen Punkte sollen in den folgenden Abschnitten näher betrachtet werden. Das Ergebnis dieser Erweiterungen stellt Abb. 6.2 dar. Die Pfeile stehen in dieser Abbildung für Modell-zu-Modell-Transformationen. Die grob gestrichelten Pfeile sollen aufzeigen, dass

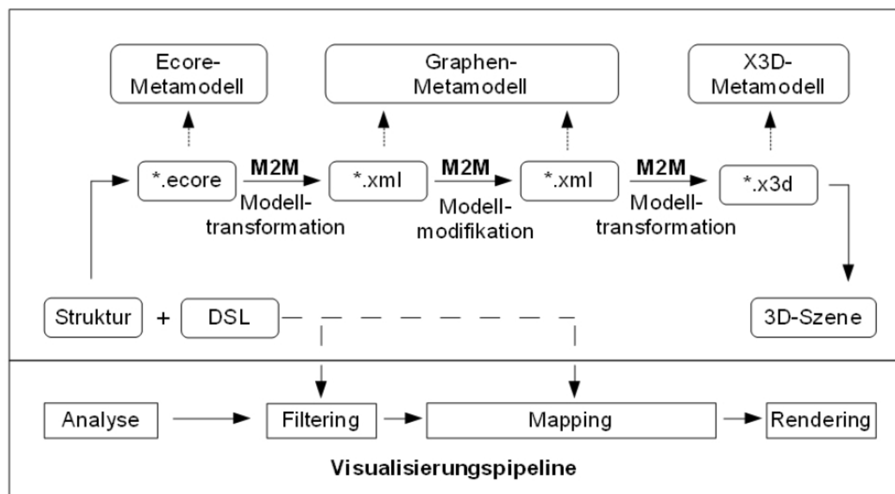

Abbildung 6.1: Visualisierungspipeline von Müller (2009)

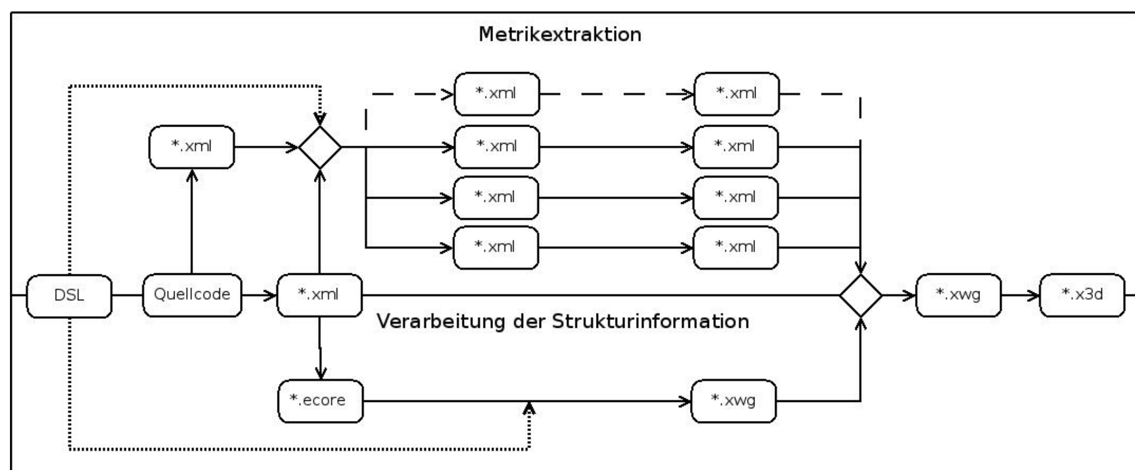

Abbildung 6.2: Erweiterte Visualisierungspipeline (in Anlehnung an: Müller, 2009)

die jeweilige Modell-zu-Modell-Transformation nur durchgeführt wird, wenn die Metapher eine ausreichende Zahl von Visualisierungsparametern ermöglicht. Dies zeigt auch, wo bei einer neuen Metapher die Ansatzpunkte liegen würden, um mehr Parameter darstellen zu können. Die fein gestrichelten Pfeile sollen eine Einflussnahme der Konfiguration auf die Transformationen zeigen. Die Rauten zeigen Modell-Merging-Vorgänge.

6.4.1 Modifikation des Visualisierungsalgorithmus

Im Rahmen der Entwicklung des Prototyps war zuerst angedacht, ebenfalls einen kraftgerichteten Algorithmus zu verwenden, um die einzelnen Metriken darzustellen. Diese Arten von Algorithmen haben aber das Problem, dass sie nicht vorhersagbar sind (vgl. Panas et al., 2007). Als Folge können die Metriken und ihre Relationen nicht exakt wiedergegeben werden, wenn die Metriken die Kräfte beeinflussen, die zwischen den Elementen auftreten. Diese Überlegungen führten dazu, von den kraftgerichteten Algorithmen wieder Abstand zu nehmen und stattdessen das Layout der Visualisierung mit einem selbstdefinierten Algorithmus zu bestimmen.

Um auch die Möglichkeiten unterschiedlicher Metaphern zu bestimmen, wurden zwei verschiedene Algorithmen entwickelt. Der erste Algorithmus erstellt in Abhängigkeit von den Metriken eine Visualisierung, die die Schachtelungsmetapher von Müller (2009) nutzt. Der zweite Algorithmus erzeugt eine Darstellung, die auf einer abstrakten Weltraummetapher aufbaut. Auf beide Metaphern und ihre zugehörigen Algorithmen wird in Abschnitt 6.6 näher eingegangen. Diese beiden Algorithmen ersetzen damit jeweils die WilmaGraph-Komponente, die im Generator von Müller (2009) genutzt wird um die Positionen und Größen der Elemente zu bestimmen. Dies führt zum einen zu der bereits erwähnten Möglichkeit, die Metriken in exakten Verhältnissen darzustellen. Zum anderen führt es aber auch dazu, das Programm plattformübergreifend einsetzen zu können, da Eclipse und openArchitetureware sowohl für Windows als auch Linux verfügbar sind. Dies gilt zwar im Grunde auch für WilmaGraph, allerdings gelang es dem Autor nicht, WilmaGraph unter Linux einzusetzen.

6.4.2 Extraktion von Strukturinformationen

Eine zusätzliche Erweiterung der Pipeline von Müller (2009) ist die Extraktion der Strukturinformationen aus dem Quellcode der Anwendung. Wie bereits oben erwähnt, verwendet Müller (2009) als Ausgangsbasis für die Visualisierung ein Ecore-Modell, welches die Strukturinformationen beinhaltet. Dieser Ansatz ist beim Merging von Struktur und Metrikinformationen zwar theoretisch ebenfalls möglich, aber nicht wirklich praktikabel. Dies liegt daran, dass die in einem Ecore-Modell vorgehaltenen Informationen nicht unbedingt aktuell sein müssen. Dadurch könnte die Vereinigung von Struktur- und Metrikinformationen Probleme bereiten, durch die das Merging der Modelle unmöglich würde.

Um diesem Problem vorzubeugen, bestand der Ansatz im Rahmen dieser Arbeit darin, die Struktur- und die Metrikinformationen auf der gleichen Datenbasis zu erheben. Diese Basis kann nur der Quellcode des Softwaresystems sein, weil er sowohl für die Metrikextraktion als auch für die Strukturinformation den Ausgangspunkt bildet. Eine weitere Bedingung bei der Erweiterung der Pipeline in diesem Punkt war der Wunsch, weiterhin Ecore bei der Visualisierung zu verwenden. So soll es möglich sein, aus dem Quellcode auch mit dem Algorithmus von Müller eine Visualisierung zu erzeugen und des Weiteren auch möglichst große Teile des bestehenden Generators weiter nutzen zu können.

Die Notwendigkeit, die Extraktion von Strukturinformationen selbst umzusetzen hatte zwei Hauptgründe. Der erste Grund war, dass es nur wenige verfügbare Tools gibt, die aus einem Eclipse-Java-Projekt direkt ein Ecore-Modell erzeugen können, ohne den Quellcode annotieren zu müssen. Der zweite Grund bestand darin, dass die gefundenen Tools ein ungewöhnliches Ecore-Modell erzeugen. Dazu bestanden mehrere Möglichkeiten diese Informationen abzugreifen, die zum Teil auch in Abschnitt 6.3 aufgeführt werden. Die einfachste war die Verwendung des Java-Modells, das Eclipse selbst in Form von Schnittstellen vorhält, um die gewünschten Strukturinformationen zu erhalten. Um ein weiterverarbeitbares Zielmodell zu erhalten, wurde ein Metamodell gewählt, das auf XML-Schema aufbaut,

```
IType[] allTypes = unit.getAllTypes();
for (IType type : allTypes) {
 if (type.isClass()) {
   IMethod[] methods = type.getMethods();
   for (IMethod method : methods) {
    methodelements.add(type.getFullyQualifiedName()
     + "." + method.getElementName());
   }
   IField[] attributes = type.getFields();
   for (IField attribute : attributes) {
        attributeelements.add(type.getFullyQualifiedName()
         + "." + attribute.getElementName());
   }
 }
}
```

Listing 6.3: Strukturextraktion mit Java

also eine valide XML-Datei darstellt. Diese XML-Datei wird mit Hilfe von DOM4J erzeugt. Listing 6.3 zeigt einen Ausschnitt aus dem Code zur Strukturextraktion.

In dem Beispiel werden für alle in einem Projekt definierte Klassen, die Methoden und Attribute jeweils in einer Liste abgelegt.

Die aus der Extraktion der Struktur entstehende XML-Datei wird einerseits dazu verwendet, die unterschiedlichen Metriken und die jeweiligen Werte für den gewünschten Parameter, wie z.B. Farbe oder Größe, aufzunehmen. Andererseits dient sie auch als Ausgangsmodell einer Modelltransformation, dessen Zielmodell ein Ecore-Modell ist, welches von der erstellten Pipeline und auch von der Pipeline von Müller zur Erzeugung einer Visualisierung genutzt werden kann.

6.4.3 Metrikextraktion und Merging mit Strukturinformationen

Eine weitere Erweiterung der Visualisierungspipeline stellt die zusätzliche Aufbereitung der Metriken, die aus dem Plugin *Metrics* gewonnen werden, dar. Je nach verwendetem Algorithmus steht eine unterschiedliche Art von Darstellungsparametern bereit, die durch Informationen aus Metriken beeinflusst werden können. Im Fall der Schachtelungsmetapher stehen vier verschiedene Slots zur Verfügung, im Fall der Weltraummetapher drei. Um die verschiedenen Metriken sauber extrahieren und zuordnen zu können, wird die vom *Metrics*-Plugin durch einen Ant-Task erzeugte XML-Datei mit der XML-Datei mit den Strukturinformationen vereinigt. Dadurch ensteht für jeden möglichen Slot eine separate XML-Datei, die sowohl Metrik- als auch Strukturinformationen enthält.

Diese Datei enthält grundsätzlich jedes darzustellende Element. Es wird aber nicht jedem dieser Elemente auch eine Metrik zugewiesen, sondern nur der Klasse von Elementen, für die die vom Nutzer ausgesuchte Metrik auch verfügbar ist. Wenn der Nutzer zum Beispiel mittels der Elementfarbe die McCabe-Komplexitätsmetrik abgebildet sehen möchte, ist diese Metrik definitionsgemäß nur für Methoden verfügbar. Pakete, Klassen und Attri-

bute können damit nicht bewertet werden. Die Farben der übrigen Elementkategorien entsprechen dann einem konfigurierbaren Standardwert. Diese Belegung mit Standardwerten erschien sinnvoll, da der Nutzer ansonsten für alle Objektkategorien die Parameter mit Metriken belegen müsste. Dies würde wahrscheinlich zu einer deutlichen Überlastung des Nutzers mit Informationen führen, sodass das Ziel der Darstellung, kritische Bereiche schnell erkennbar zu gestalten, nicht mehr gegeben wäre. Als weitere Folge würde die Konsistenz innerhalb der Darstellung unter dieser freien Konfiguration leiden, zum Beispiel wenn der Nutzer über die Paketfarbe Größenmaße und über die Methodenfarbe Komplexitätsmaße abbildet.

Die so erzeugten Dateien werden nun dahingehend aufbereitet, dass in ihnen nicht nur die reinen Metrikwerte zu finden sind, sondern ebenfalls die Ausprägungen der einzelnen Parameter. Dies führt zu einer separaten Berechnung der einzelnen Parameter, was zur Folge hat, dass dabei keine Seiteneffekte auftreten können und dass die Implementierung der jeweiligen Parameterbestimmung auch relativ leicht wieder ausgetauscht werden kann.

Um die Konfiguration möglichst einfach zu halten, wurde eine einfach gehaltene Berechnungsvorschrift für die Bestimmung der jeweiligen Parameterwerte erstellt. Diese erzeugt aus den minimalen, maximalen und der aktuellen Ausprägung einen Prozentwert, der dann wiederum auf eine Spanne von Parameterwerten gelegt wird:

$$a = \frac{m - m_{\min}}{m_{\max} - m_{\min}} \,. \tag{6.1}$$

a ist der ermittelte Anteil, den die aktuelle Metrikausprägung m auf einer Skala einnimmt, die von der minimalen Metrikausprägung $m_{\min}$ bis zur maximalen Metrikausprägung $m_{\max}$ reicht. Dieser Wert muss nun auf eine bestimmte Spanne von möglichen Werten für den Parameter abgebildet werden. Dazu müssen die maximal und minimal möglichen Parameterausprägungen bestimmt werden. Die Bestimmung unterscheidet sich je nach Art des Parameters signifikant. Bei der Farbe wird der Wert auf einen Bereich zwischen der Minimalfarbe und der Maximalfarbe abgebildet:

$$\begin{pmatrix} c_{\mathrm{R}} \\ c_{\mathrm{G}} \\ c_{\mathrm{B}} \end{pmatrix} = \left(\begin{pmatrix} c_{\mathrm{R_{max}}} \\ c_{\mathrm{G_{max}}} \\ c_{\mathrm{B_{max}}} \end{pmatrix} - \begin{pmatrix} c_{\mathrm{R_{min}}} \\ c_{\mathrm{G_{min}}} \\ c_{\mathrm{B_{min}}} \end{pmatrix} \right) \cdot a + \begin{pmatrix} c_{\mathrm{R_{min}}} \\ c_{\mathrm{G_{min}}} \\ c_{\mathrm{B_{min}}} \end{pmatrix} \,. \tag{6.2}$$

c_{R}, c_{G} und c_{B} stehen dabei für den jeweiligen Farbanteil der Grundfarben Rot, Grün und Blau. Die jeweiligen Minimal- und Maximalwerte wurden vom Nutzer festgelegt.

In den Fällen von Größenangaben ist die Bestimmung der Werte einfacher:

$$r = (r_{\max} - r_{\min}) \cdot a + r_{\min} \,. \tag{6.3}$$

Hierbei ist r der Radius des jeweiligen Elements und die Werte $r_{\max}$ und $r_{\min}$ stehen für die maximalen und minimalen Radien. Die Werte für die Positions- und Abstandsparameter werden analog zu Gl. 6.3 berechnet.

Die Bestimmung der Minimal- und Maximalwerte für die einzelnen Parameter unterscheidet sich dabei vor allem in Abhängigkeit von der Metapher. Während bei der Schachtelungsmetapher die Minimalgrößen der einzelnen Elemente von der Größe der darin enthaltenen Elemente abhängt, sind solche Berechnungen im Bereich der abstrakten Weltraummetapher nicht von Nöten. Dort ergeben sich die Minimal- und Maximalgrößen für alle Elemente aus den vom Nutzer vorgegebenen Werten.

Bei der Schachtelungsmetapher können nach der Parameterberechnung die jeweiligen Werte direkt übernommen werden, da hier die Werte für y- und z-Achse bereits absolut berechnet wurden. Einzig die Ausrichtung entlang der x-Achse wird in Abhängigkeit der Position der Elemente in einer geordneten Liste bestimmt. Diese Dimension wird dazu verwendet, um Kollisionen von Elementen zu vermeiden. Die Werte für Farbe, Form und Position werden nun mit einer WilmaGraph-Datei verwoben. Diese Datei entsteht durch die von Müller (2009) konzipierte Pipeline aus dem Ecore-Modell, das wiederum aus den gewonnenen Strukturinformationen abgeleitet wurde. Zieldatei dieser Transformation ist ebenfalls eine WilmaGraph-Datei, welche aber alle notwendigen Darstellungsinformationen beinhaltet. Diese Transformation ersetzt damit den Schritt bei Müller (2009), in dem WilmaGraph die Positionierung der Elemente mittels einem kraftgerichteten Algorithmus übertragen wird. Das so gewonnene Modell kann nun mittels einer Transformation von Müller (2009) in das gewünschte X3D-Format umgewandelt und dann mittels eines entsprechenden Browsers erkundet werden.

Bei der verwendeten abstrakten Weltraummetapher, die in Abschnitt 6.6.2 näher beschrieben wird, musste die Vorgehensweise variiert werden, um die gewünschten Daten zu generieren. Die zu bestimmenden Parameter umfassen nur die Größe, die Farbe und den Abstand zum Elternelement und nicht die Positionierung entlang der Koordinatenachsen im dreidimensionalen Raum. Um die Positionswerte festzulegen, werden zunächst vom Wurzelelement beginnend die einzelnen Elemente in einem flachen Raum kreisförmig um ihr Elternelement angeordnet. Ist der Abstand zum Elternelement durch eine Metrik beeinflusst, so wird der konkrete Abstand zum Elternelement in Abhängigkeit der Metrik festgelegt. Da der Platz im dreidimensionalen Raum besser genutzt werden soll, werden darauf folgend die Kindelemente gleichmäßig um ihre Elternelemente rotiert. Im Rahmen des Prototyps kommen für diese Rotation separate Drehungen um die x- und z-Achse zum Einsatz. Um die neuen Positionen zu bestimmen, werden die Ortsvektoren der Punkte mit der jeweiligen Drehmatrix multipliziert:

$$\begin{pmatrix} x_{\text{neu}} \\ y_{\text{neu}} \\ z_{\text{neu}} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix} \cdot \begin{pmatrix} x_{\text{alt}} \\ y_{\text{alt}} \\ z_{\text{alt}} \end{pmatrix}, \tag{6.4}$$

$$\begin{pmatrix} x_{\text{neu}} \\ y_{\text{neu}} \\ z_{\text{neu}} \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_{\text{alt}} \\ y_{\text{alt}} \\ z_{\text{alt}} \end{pmatrix}. \tag{6.5}$$

Gleichung 6.4 zeigt die Berechnung des neuen Vektors eines Punktes bei einer Drehung um die x-Achse von α Grad. Gleichung 6.5 zeigt das gleiche Vorgehen, allerdings bei Drehung um die z-Achse. Um α zu erhalten, werden derzeit innerhalb des Prototyps Zufallszahlen verwendet, da die Bestimmung des Drehwinkels auf diese Weise sehr einfach ist. Perspektivisch wäre es aber wünschenswert, wenn die jeweiligen Drehwinkel optimal oder zumindest in einer reproduzierbaren Weise gewählt würden. Die Bestimmung mittels eines reproduzierbaren Algorithmus würde es ermöglichen, bei mehreren aufeinander folgenen Visualisierungen möglichst ähnliche Darstellungen zu erhalten, sodass die einmal gewonnene Orientierung im dreidimensionalen Raum nicht bei jedem neuen Visualisierungslauf verworfen werden muss. Der Nutzer könnte sich so wahrscheinlich schneller wieder zurecht finden.

Auch bei der abstrakten Weltraummetapher wird der Aufruf der WilmaGraph-Komponente durch einen Modell-Merging-Schritt ersetzt, der die vorher berechneten Ausprägungen der einzelnen Parameter mit einer WilmaGraph-Datei zusammenführt, aus der dann in einer weiteren Modelltransformation ein X3D-Modell erzeugt werden kann.

6.5 Metamodelle

Im Rahmen der Prototypentwicklung kommen fünf unterschiedliche Metamodelle zum Einsatz. Die unterschiedlichen diese Metamodelle instanziierenden Modelle werden im Rahmen des Workflows hin zum fertigen dreidimensionalen Modell ineinander umgewandelt.

6.5.1 Abstrakte Java-Syntax

Im Rahmen des Prototyps stellt die abstrakte Java-Syntax ein wichtiges Metamodell dar, denn aus den in Java geschriebenen Quellcodes kann die Struktur des Softwaresystems extrahiert werden. Die Möglichkeit dieser Extraktion wird vor allem in Abschnitt 6.3.1.2 beschrieben.

Als Parser für dieses Metamodell dient Eclipse, welches mit dem Java-Model eine Schnittstelle bereitstellt, aus der die Struktur auf einfache Weise über verschiedene bereitgestellte Schnittstellen ausgelesen werden kann. Der Vorgang des Auslesens wird in Abschnitt 6.4.2 näher beschrieben.

6.5.2 XML-Schema

XML-Schema ist eine der beiden meistgenutzten Techniken zur Spezifizierung der möglichen Struktur eines XML-Dokuments. Die alternative Technik zu XML-Schema stellt die Document Type Definition (DTD) dar. Beide Techniken legen fest, wie ein XML-Dokument, das einer solchen Spezifikation gehorcht, aussehen muss, damit es als valide gilt. Ohne eine solche Möglichkeit könnten XML-Dokumente nur auf Wohlgeformtheit geprüft werden, was allerdings beim elektronischen Datenaustausch oder bei domänenspezifischen Dokumenten nicht ausreichend ist. Die Document Type Defintion bietet weniger Gestaltungs- und Einschränkungsmöglichkeiten als XML-Schema. Zudem kann bis jetzt nur XML-Schema

im Rahmen von openArchitectureWare zur Definition von Metamodellen eingesetzt werden beziehungsweise es können nur XML-Schemata und nicht DTDs als Metamodelle genutzt werden.

Im Rahmen der Prototypentwicklung kommen zwei unterschiedliche Metamodelle zum Einsatz, die auf XML-Schema aufbauen. Sie werden in den folgenden Absätzen näher vorgestellt. Grundsätzlich können auch Ecore-Modelle als XML-Dateien repräsentiert werden. Da Ecore als Metamodell der Eclipse Modelling Foundation allerdings auf dem Essential Meta-Object Facility Standard (EMOF) aufbaut, wird im Rahmen dieser Arbeit separat auf Ecore eingegangen.

6.5.2.1 Flaches Metrikschema

Das Plugin Metrics für Eclipse in Version 1.3.6. ist in der Lage, mittels eines Ant-Skripts eine XML-Datei zu erzeugen, die alle berechneten Werte für die verfügbaren Metriken beeinhaltet. Diese XML-Datei gehorcht einem flachen XML-Schema. Das heißt, die erstellte Datei enthält keine Strukturinformationen beziehungsweise die Strukturinformationen müssten aus den in der Datei zum Teil verfügbaren vollqualifizierten Bezeichnern extrahiert werden. Es wäre sehr aufwendig, Informationen zur Struktur des Java-Projektes aus dieser Datei zu extrahieren. Zudem ist es für die Weiterentwicklung des Prototyps auch von Vorteil, wenn die Erhebung der Strukturdaten unabhängig von der Erhebung der zu visualisierenden Zusatzinformationen geschieht. So ist es einfacher möglich, andere Informationen in den Generator einzubinden oder die Art der Metriken aufgrund der besseren Modularisierung einfacher auszutauschen.

Die Datei selbst enthält zu jeder erstellten Metrik einen eindeutigen Bezeichner in Form einer Kurzbezeichnung. In diesem Element mit dem Namen `Metric` sind Informationen wie der Minimal- und der Maximalwert, verschiedene unterschiedliche statistische Maße wie der Durchschnitt und die Standardabweichung zu finden. Zudem finden sich in diesem Element auch Angaben, zu welcher Objektkategorie die jeweilige Metrik Informationen liefert. In der dem `Metric`-Element untergeordneten Hierarchiestufe findet sich dann ein `Values`-Element, das für jedes bewertete Objekt ein `Value`-Element enthält. Dieses `Value`-Element enthält, je nachdem ob die Metrik für Pakete, Klassen oder Methoden verfügbar ist, unterschiedliche Angaben. Im Fall von Paketmetriken enthält es den vollqualifizierten Namen des Pakets und den Wert der Metrik. Im Fall von Klassen enthält es den Klassennamen, die zugehörige Quellcodedatei und den Namen des Pakets, in dem sich die Klasse finden lässt. Aus diesen Angaben lässt sich der vollqualifizierte Name der Klasse bilden, der als Primärschlüssel im Rahmen des Mergings benötigt wird. Für die Metriken, die Methoden Werte zuordnen, sind die Angaben leider unzureichend. Hier werden der Name der Methode, die Quellcodedatei, in der die Methode gefunden wurde, und das Paket angegeben. Aus dem Namen der Quellcodedatei lässt sich in der Regel der Name der Klasse ablesen, zu der die Metrik gehört. Leider gibt es hiervon auch Ausnahmen. Diese treten vor allem dann auf, wenn in einer Quellcodedatei mehrere Klassen zu finden sind. Das führt dazu, dass der Primärschlüssel

der Methoden, also ihr vollqualifizierter Name leider im Rahmen des Prototyps nicht immer exakt bestimmt werden kann.

Zusätzlich zu den Metrikinformationen bietet das Metamodell auch die Möglichkeit, mittels eines `Cycle`-Elements zyklische Paketabhängigkeiten abzubilden.

6.5.2.2 Strukturerhaltendes Metrikschema

Um die Metrikdateien mit den Strukturinformationen verbinden zu können und auf diese Weise auch die Berechnungen der Parameterwerte unabhängig von weiteren Dateien durchführen zu können, wurde vom Autor der vorliegenden Arbeit ein XML-Schema entwickelt, das eine Abbildung der Werte der jeweiligen Metriken und der Ausprägungen der Parameter zusammen mit den Strukturinformationen in einer Datei ermöglicht.

Grundprinzip des Schemas ist die Abbildung der Struktur, die auch später für die Visualisierung benötigt wird. Da die Angabe von Metrikwerten optional ist, ist es möglich, das Schema auch ausschließlich als Modell der Struktur zu verwenden. Diese Möglichkeit wird bei der Extraktion der Struktur genutzt, die in Abschnitt 6.4.2 näher beschrieben wird. Das daraus resultierende Modell kann dann mittels einer XTend-Transformation in ein Ecore-Modell überführt werden, um dann weiterverwendet zu werden.

Der Aufbau einer durch dieses Schema definierten Datei sieht als einziges Element unter dem Wurzelelement `Metrics` ein `Package`-Element vor. Dies soll ein zentrales Element definieren, an dem sich der Nutzer in der Visualisierung orientieren kann. Um diese Restriktion sicherzustellen, ist auch der Code für die Extraktion so angepasst, dass er im Fall mehrerer Pakete auf der untersten Ebene eines Projektes ein zusätzliches künstliches Oberpaket einfügt. In einem `Package`-Element können wiederum mehrere `Package`-Elemente geschachtelt sein, sodass sich eine beliebige Schachtelungstiefe ergibt. Zudem können auch `Class`-Elemente den `Package`-Elementen zugewiesen werden. Die `Class`-Elemente können keine weiteren Klassen enthalten. In Java ist es zwar grundsätzlich möglich, innere Klassen zu definieren, allerdings stellen diese zumeist nur Helferklassen dar, von denen im Rahmen der Visualisierung abstrahiert werden sollte. Auch Schnittstellen werden in dem strukturerhaltenden Schema nicht repräsentiert. Dies ist darauf zurückzuführen, dass Schnittstellen nicht selbst Code enthalten und deshalb der statischen Codeanalyse nicht zugänglich sind.

Die beiden Elemente `Method` und `Attribute` sind unter einem `Components`-Element subsumiert, das wiederum Bestandteil des `Class`-Elements ist.

Das Wurzelelement, die Paketelemente und die Klassenelemente können zusätzlich Elemente besitzen, die einige statistische Angaben enthalten. Zum Beispiel enthält das Element `ClassSummary` Werte für die Anzahl der Klassenkomponenten sowie separat aufgeschlüsselt die Anzahl der Methoden und Attribute. Zusätzlich enthalten sind außerdem noch die Minimal- und Maximalwerte sowie die Durchschnittswerte der Metriken. Mit diesen Angaben werden die Modelle nach dem Merging der Struktur- mit den Metrikinformationen in einem separaten Modelltransformationsschritt angereichert.

Während des Mergings der Struktur mit den aus dem flachen Metrikmodell entnommenen Informationen wird den Elementen, für die ein Metrikwert verfügbar ist, dieser Wert durch ein `Metric`-Element zugewiesen. In die anderen Elemente wird ein `isDefault`-Element eingefügt, sodass sich in späteren Modelltransformationen bestimmen lässt, ob für das jeweilige Element ein Wert vorhanden ist oder ob die Standardberechnung für den jeweiligen Parameter durchgeführt werden soll.

Zusätzlich zu diesem `Metric`- beziehungsweise `isDefault`-Element wird den einzelnen Elementen in einem späteren Transformationsschritt ein `Value`-Element zugewiesen. Dieses enthält dann den Wert, den der jeweils bestimmte Parameter für dieses Element annimmt, sodass dieser Wert dann beim Merging mit der WilmaGraph-Datei nur noch übernommen werden muss.

6.5.2.3 WilmaGraph-Schema

Das *WilmaGraph*-Schema stellt das Metamodell für *WilmaGraph*-Modelle dar. Die Modelle werden vor allem deshalb im Rahmen des Prototyps genutzt, da es auch weiterhin möglich bleiben soll, die Elemente von WilmaGraph anordnen zu lassen, ohne die Metrikinformationen zu generieren und auszuwerten. Das WilmaGraph-Schema stellt die Spezifikation für ein einfaches 3D-Modell-Format dar. Es definiert unterschiedliche dreidimensionale Primitive wie Quader und Kugeln, die dann den verschiedenen `Cluster`- oder `Node`-Elementen zugewiesen werden. Ein `Cluster` stellt in WilmaGraph ein Oberelement für mehrere `Node`-Elemente dar. Sowohl `Cluster`- als auch `Node`-Elemente haben ein `ViewType`-Element, dessen `Name`-Attribut die Form definiert. Zusätzlich können mittels des `Property`-Elements noch Name-Wert-Paare definiert werden. Im Rahmen des Prototyps werden diese Name-Wert-Paare zum Speichern von Informationen über das Objekt genutzt.

6.5.2.4 Extensible 3D

Dieses Metamodell stellt die Basis für das gewünschte Zielmodell dar. Wie bereits in Abschnitt 3.6 beschrieben, ist Extensible 3D ein XML-basiertes Format, das für die Informationsvisualisierung und damit die Visualisierung von Software gut geeignet ist. Im Rahmen der Prototypentwicklung kommt das vom Web3D-Consortium für den X3D-Standard herausgegebene XML-Schema zum Einsatz. Die Umwandlung des WilmaGraph-Modells in das X3D-Modell übernimmt eine XTend-Transformation, die von Müller im Rahmen seiner Diplomarbeit erstellt wurde (vgl. Müller, 2009). Die im Rahmen des Prototyps genutzten X3D-Elemente finden sich in Tabelle 6.4

Bei Bedarf kann das so entstandene X3D-Modell zudem noch in sein Vorgängerformat VRML umgewandelt werden, um das dreidimensionale Modell im Virtual Reality Labor des Instituts für Wirtschaftsinformatik in Leipzig zu untersuchen. Um die Transformation ins VRML-Format vorzunehmen, kommt ein XSLT-Stylesheet zum Einsatz, das von Pascal Kovacs im Rahmen seiner Diplomarbeit entwickelt wurde (Kovacs, 2010).

Elemente	Komponente	Funktion
`Scene`	–	stellt das Wurzelelement des Szenegraphen dar
`Transform`	*Grouping*	dient der Gruppierung von Elementen und Definition eines Koordinatensystems für die Kindknoten
`MetadataSet`	*Core*	enthält eine Menge an Elementen vom Typ `MetaDataString`
`MetadataString`	*Core*	enthält einen String mit Metadaten
`Shape`	*Shape*	nimmt die Werte auf, die benötigt werden um visuelle Elemente zu rendern; enthält die Elemente, die die visuellen Eigenschaften und die Geometrie bestimmen
`Appearance`	*Shape*	definiert die visuellen Eigenschaften; enthält den `Material`-Knoten
`Material`	*Shape*	definiert das Oberflächenmaterial des darzustellenden Elements
`Text`	*Text*	spezifiziert einen zweidimensionalen zweiseitigen Textstring
`Box`	*Geometry3D*	bestimmt die Geometrie, Würfelform
`Sphere`	*Geometry3D*	bestimmt die Geometrie, Kugelform
`Cone`	*Geometry3D*	bestimmt die Geometrie, Kegel
`Cylinder`	*Geometry3D*	bestimmt die Geometrie, Zylinder

Tabelle 6.4: Verwendete X3D-Elemente

6.5.3 Ecore

Ecore ist das Metamodell des Eclipse Modelling Frameworks (EMF). Ziel von EMF ist es, für die Entwicklung mit Java und Eclipse verschiedene Möglichkeiten zur modellgetriebenen Softwareentwicklung zur Verfügung zu stellen. Dafür wurde ein einheitliches Metamodell entwickelt, nämlich Ecore. Es basiert auf der Meta Object Facility-Spezifikation der Object Modelling Group, genauer gesagt auf dem Essential Meta Object Facility-Metamodell (vgl. Budinsky, 2005; OMG, 2006).

Das Eclipse Modelling Framework bietet unterschiedliche Repräsentationen von Modellen an. So ist es zum Beispiel möglich, annotierten Java-Quellcode zur Definition eines EMF-Modells zu verwenden. Eine weitere Möglichkeit besteht in der Definition eines EMF-Modells, mit Hilfe eines UML-Modelierungswerkzeugs, wie zum Beispiel *Omondo* (Omondo Corp., 2010). Um die verschiedenen Modellarten untereinander austauschbar zu gestalten, werden alle Modelle auf Ecore-Modelle abgebildet. So sind für alle Modellarten Mappings verfügbar, die festlegen, wie die einzelnen Elemente der jeweiligen Modellart auf ein Ecore-Modell abgebildet werden. OpenArchitectureWare ist in der Lage, dieses Mapping transparent durchzuführen und somit Modelle unterschiedlicher Art zu transformieren und zu verbinden.

Die grundlegende Repräsentation für Ecore-Modelle stellt XML Metadata Interchange (XMI) dar. XMI ist, wie der Name bereits vermuten lässt, eine XML-basierte Repräsenta-

tion. Eine solche Datei kann mit den üblichen XML- oder Texteditoren bearbeitet werden. Allerdings stellt EMF auch einen baumbasierten Ecore Editor bereit, mit dem es möglich ist, die XMI-Dateien einfacher zu bearbeiten.

Im Rahmen des Prototyps kommt Ecore in seiner XMI-Repräsentation als Zwischenstufe für die Erzeugung eines WilmaGraph-Modells zum Einsatz. Dies dient vor allem dazu, das von der Strukturextraktion erzeugte Strukturmodell auch ohne die Anreicherung mit Metrikinformationen mit Hilfe des Generators von Müller weiterverarbeiten zu können.

6.6 Evaluierung von Metaphern

Unterschiedliche Metaphern besitzen verschiedene Vor- und Nachteile. Um dem Nutzer des Generators verschiedene Möglichkeiten an die Hand zu geben, die spezifischen Vorteile zu nutzen, beinhaltet der Prototyp zwei verschiedene Darstellungsmöglichkeiten, zwischen denen der Nutzer wählen kann. Die erste mögliche Darstellungsform ist eine Schachtelungsmetapher, die sich an der Arbeit von Rekimoto und Green (1993) und an der Umsetzung im Rahmen der Arbeit von Müller (2009) orientiert. Die zweite Möglichkeit zur Darstellung ist eine abstrakte Weltraummetapher. Diese beiden Metaphern sollen in den folgenden Abschnitten näher beleuchtet werden. Dabei sollen auch die Eigenschaften, wie Vor- und Nachteile sowie Möglichkeiten und Beschränkungen erläutert werden.

6.6.1 Schachtelungsmetapher

Die Schachtelungsmetapher war zunächst die naheliegendste Darstellungsform, um die Visualisierung im Rahmen des Prototyps durchzuführen. Der Prototyp baut auf dem Generator von Müller auf und dieser nutzt bereits die Schachtelungsmetapher. Zunächst stand die Frage im Raum, welche Parameter diese Darstellung bietet, um Metrikwerte zu visualisieren. Die Parameter, die wohl am häufigsten im Rahmen von unterschiedlichen Visualisierungen genutzt werden, sind die Größe, die Ausrichtung in x-, y- und z-Richtung, die Farbe von Objekten, ihre Form und ihre Neigung im dreidimensionalen Raum.

Müller (2009) nutzte die Form von Objekten zur Unterscheidung von verschiedenen Objektklassen, ein Vorgehen, das auch für den Prototyp im Bereich dieser Arbeit sinnvoll erscheint. Die Form der Objekte kann nur für eine sehr beschränkte Anzahl an Metriken genutzt werden kann. Wenn zum Beispiel die Anzahl der Ecken eines Objektes einen bestimmten Metrikwert repräsentieren soll, müsste die Metrik auf einer Absolutskala arbeiten und die Werte dieser Absolutskala dürften maximal so hoch sein, dass der normale Nutzer noch intutitv Vergleiche ziehen kann. Die Notwendigkeit, Unterschiede zwischen einem Objekt mit 45 und einem mit 46 Ecken zu erkennen, kann nicht als intuitiv angesehen werden. Aufgrund dieser Einschränkungen wird im Rahmen dieser Arbeit die Form zur Differenzierung unterschiedlicher Objektklassen genutzt.

Die Farbe von Objekten kann bei der Schachtelungsmetapher frei zur Darstellung von Metriken eingesetzt werden. Wie bereits in Abschnitt 3.4.2.2 beschrieben, ist dabei zu be-

achten, dass die Schachtelungsmetapher keine Objekte nutzt, die aus Gitterlinien bestehen, sondern transparente Objekte (vgl. Rekimoto und Green, 1993). Der Nachteil davon ist, dass die Farben der Objekte durch ihre eigene Transparenz, durch in ihnen dargestellte Objekte und durch die beinhaltenden Objekte verfälscht werden können.

Die Größe von Objekten ist metapherntypisch beschränkt. Die Objekte müssen immer so groß gewählt werden, dass die eingeschlossenen Objekte genug Raum haben, um sich anordnen zu können. Dies beschränkt die Nutzungsmöglichkeiten auch im Rahmen dieser Metapher. Im Rahmen des Prototyps werden diese Einschränkungen berücksichtigt, indem zuerst die Größen aller Klassenbestandteile entsprechend den Nutzervorgaben berechnet werden. Danach werden die Größen aller Klassen in Abhängigkeit von der maximalen Anzahl an Klassenbestandteilen in einer Klasse berechnet. Dies führt dazu, dass sowohl die Größe der Klassenbestandteile als auch die Größe der Klassen vergleichbar bleiben, auch wenn sie sich auf verschiedenen Schachtelungsebenen befinden. Bei Paketen ist dies leider nicht möglich, da die Metapher einen ebenenübergreifenden Größenvergleich nicht zulässt. Dies wäre zum Beispiel ein Problem, wenn ein Paket mit einem hohen Wert in einer Metrik in ein Paket mit einem niedrigen Wert geschachtet wird. Hier könnte aufgrund der Metaphernrestriktionen kein Vergleich der Metriken durchgeführt werden.

Ein weiteres Problem ist die Anordnung der Objekte im Raum. Zuerst war angedacht, alle drei Dimensionen im Raum auszunutzen, um Metriken zu visualisieren. Die nicht von Metriken belegten Ausrichtungsparameter wurden dabei zufällig bestimmt. Dieses Vorgehen wurde im Rahmen der Prototypentwicklung später wieder verworfen. Es zeigte sich, dass durch die eventuell große Anzahl an unbestimmten Ausrichtungen eine große Wahrscheinlichkeit zur Kollision von Objekten im Raum besteht. Zum anderen ist es bezüglich einer eventuellen Weiterentwicklung des Generators im Hinblick auf die Visualisierung von Veränderungen während des Softwareentwicklungsprozesses wünschenswert, wenn keine Zufallswerte zum Einsatz kommen, sondern der Algorithmus bei gleichen Eingangsdaten immer das gleiche Ausgangsmodell erstellt.

Die Rotation von Objekten kam im Rahmen des Prototyps nicht zum Einsatz, da fraglich erscheint, ob eine solche Rotation verschiedener abstrakter Primitive vergleichbar ist. Grund dafür ist vor allem, dass zum Teil symmetrische Primitive wie Kugeln und Würfel zum Einsatz kommen. Bei Kugeln ist der Einsatz von Rotation zur Darstellung von Metriken oder von anderen Informationen allgemein unmöglich, beim Einsatz von Würfeln schwierig. Eine weitere Information, die im Prototyp nicht dargestellt wird, sind die Beziehungen zwischen Elementen. Diese könnten zum Beispiel durch Verbindungslinien gezeigt werden. Darauf wurde verzichtet, da die Darstellung solcher Linien einen komplexen Algorithmus erfordern würde. Dieser müsste sicherstellen, dass Verbindungen sich zum einen nicht schneiden, aber dabei auch nicht mit weiteren Objekten kollidieren. Müller (2009) zeigt im Rahmen seines Generators die Möglichkeit, Vererbungsbeziehungen durch die Inklusion der Basisklasse in die erbende Klasse darzustellen, da diese alle nicht privaten Elemente ihrer Basisklasse erbt. Auch diese Möglichkeit wurde im Rahmen der hier aufgezeigten Prototypentwicklung ver-

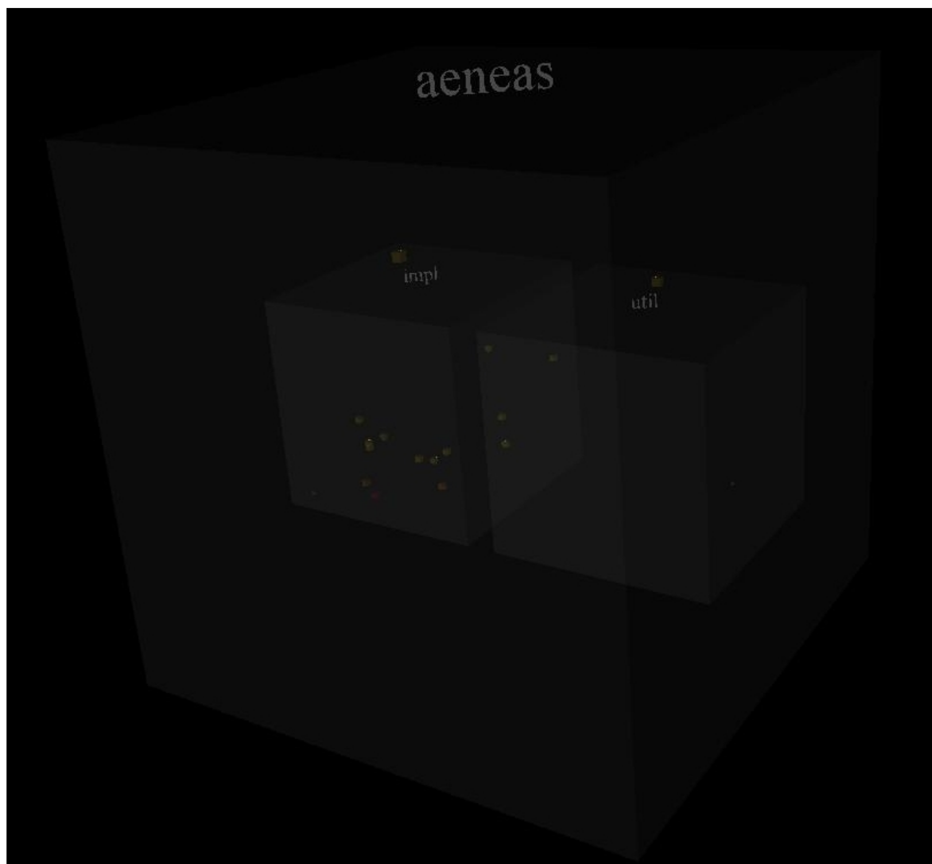

Abbildung 6.3: Visualisierung mit der Schachtelungsmetapher

worfen, um die Darstellung übersichtlich zu halten und den Fokus auf die Metrikvisualisierung zu legen.

Ein Beispiel für eine Visualisierung mittels der Schachtelungsmetapher im Rahmen des Prototyps zeigt Abb. 6.3. Nach mehreren Testprojekten, die mit unterschiedlichen Visualisierungen dargestellt wurden, entstand die Einschätzung, dass die Schachtelmetapher Vorteile bei der Untersuchung einzelner Klassen mit ihren Bestandteilen hat. Hier können die einzelnen Klassenbestandteile gut bis sehr gut in bis zu vier unterschiedlichen Metriken verglichen werden. Die Schachtelung bietet zudem dem Nutzer die Möglichkeit, sich gut zurecht zu finden, da ihm unterschiedliche Orientierungspunkte in seinem Blickfeld gegeben werden.

Der Nachteil der Metapher ist die fehlende Möglichkeit, Klassenbestandteile und Klassen ebenenübergreifend ausreichend gut vergleichen zu können. Dieses Problem entsteht zum einen dadurch, dass die einzelnen Objekte mit zunehmender Schachtelungstiefe zunehmend opak werden. Zum anderen liegt es daran, dass durch die Restriktionen der Metapher die einzelnen Objekte zum Teil sehr viel kleiner als ihre Elternelemente dargestellt werden.

6.6.2 Abstrakte Weltraummetapher

Die abstrakte Weltraummetapher wurde im Rahmen der Prototypentwicklung entworfen, um einige Probleme der Schachtelungsmetapher zu lösen (Abb. 6.4). Das Prinzip der Metapher besteht darin, die Kindelemente eines bestimmten Elements kreisförmig um dieses anzu-

ordnen. Aufrund dieser Ähnlichkeit zu den Verhältnissen im Weltraum, wird die Metapher als Weltraummetapher bezeichnet. Die Darstellung von weiteren Beziehungen zwischen den Objekten wurde auch in dieser Metapher nicht implementiert, da auch hier die bereits oben geschilderten Probleme mit der optimalen Positionierung der Verbindungen bestehen, sollten sie durch Linien dargestellt werden. Im Gegensatz zur Metapher, die bei Graham et al. (2004) zum Einsatz kommt, wurde bei der Entwicklung des Prototyps im Rahmen dieser Arbeit die Form zur Unterscheidung von unterschiedlichen Objektklassen gewählt. Bei Graham et al. (2004) kommt zur Differenzierung die Farbe zum Einsatz. Die Form wurde, wie oben bereits beschrieben, zur Differenzierung gewählt, da sie nicht gut bei der Metrikvisualisierung eingesetzt werden kann. Die abstraktere Darstellung hat nach Meinung des Autors dieser Arbeit zudem den Vorteil, dass in eventuellen späteren Arbeiten leichter zusätzliche Elemente, wie z. B. Verbindungen zwischen den Objekten eingeführt werden können. Die Nutzer der Metapher akzeptieren wahrscheinlich zusätzliche weltraumuntypische Elemente in einer bereits abstrakten Weltraummetapher eher, als dies in einer den bekannten Gegebenheiten entsprechenden Metapher gegeben wäre.

Um den Platz zu nutzen, den die dritte Dimension bietet, werden die einzelnen Objektebenen gekippt. Diese Kippung, die auf zwei Kippungen entlang der x- und der z- Achse beruht, wird aktuell durch Zufallszahlen bestimmt. Wie bereits oben beschrieben, wäre es wünschenswert, wenn dies nicht zufällig geschehen würde. Das Finden eines geeigneten Algorithmus, der die Objekte möglichst optimal im Raum anordnet, hätte den Rahmen dieser Arbeit überstiegen.

Die Probleme, die diese Darstellungsform löst, sind vor allem die Schwierigkeiten mit der Farbe und der Größe. Da alle Objekte ohne Transparenz dargestellt werden können und auch keine Restriktionen der Größe vorhanden sind, können zwei der größten Probleme der

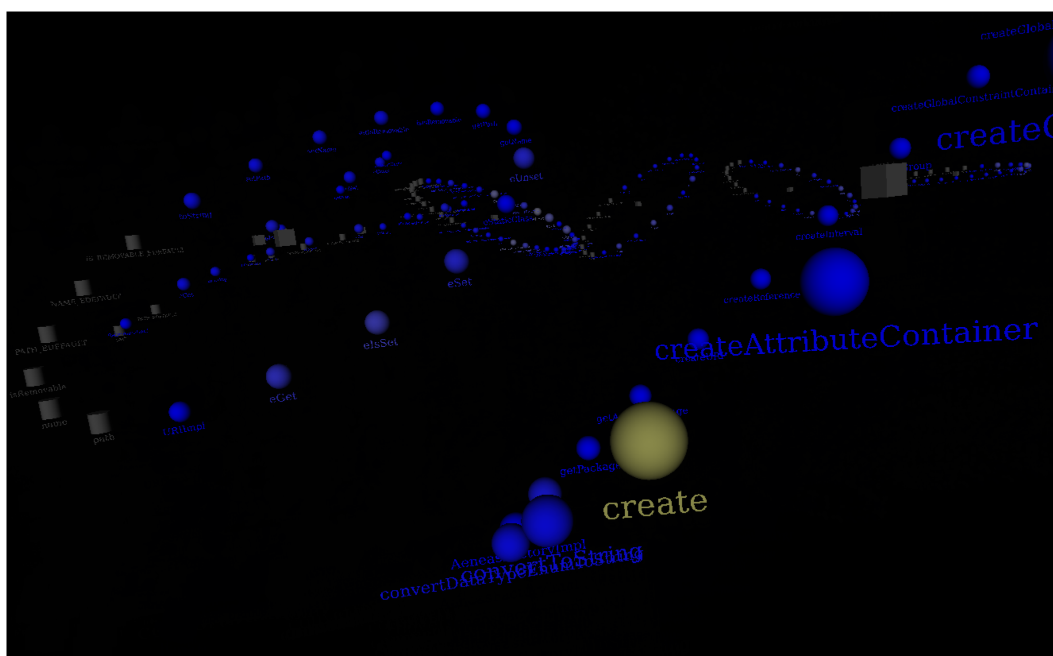

Abbildung 6.4: Visualisierung mit der abstrakten Weltraummetapher

Schachtelungsmetapher durch diese Darstellungsform gelöst werden.

Aber diese Vorteile müssen mit gewissen Einschränkungen in anderen Bereichen erkauft werden. Da die Anordnung der Objekte die Zugehörigkeit zu Elternelementen darstellt, ist die Parametrisierung der Ausrichtung in x-, y- oder z-Richtung nicht mehr möglich. Um außer der Farbe und der Größe der Objekte noch eine weitere Parametrisierungsmöglichkeit zu bieten, wurde zuerst die Rotation der Ebenen überlegt. Grundsätzlich wäre ein solches Vorgehen möglich. Ein großer Vorteil wäre zum Beispiel die Konsistenz der Rotation, die in der Folge die Visualisierung vorhersagbar machen würde. Ein erster großer Nachteil würde aber bei den nicht mit Metriken belegten Objekten auftreten. Hier müsste als Standardwert ein Rotationsgrad α von 0 Grad in x- und z-Richtung angenommen werden, da sonst der Nutzer nicht mehr unterscheiden könnte, welche Rotation durch eine Metrik verursacht wird und welche einen Zufallswert abbildet. Dies ist soweit möglich, allerdings wäre dann der Platz im dreidimensionalen Raum sehr unzureichend ausgenutzt. Ein zweiter großer Nachteil der Verwendung der Rotation ist das Problem bei Attributen und Methoden. Diese besitzen in der gewählten Darstellungsform keine Unterelemente. Dies hat zur Folge, dass allein die Objekte im Raum rotiert wären. Wie bereits in Abschnitt 6.6.1 beschrieben, könnte es bei ungünstig gewählten Formen nicht mehr möglich sein, den Grad der Rotation genau erkennen zu können. Aus diesen Gründen würde die Möglichkeit, die Rotation mit einer Metrik belegen zu können, nicht implementiert.

Da die reine Ausrichtung in den drei Dimensionen nicht zur Metrikdarstellung verwendbar war, wurde im Rahmen der Prototypentwicklung ein weiterer Parameter gesucht und mit dem Abstand zum Elternelement gefunden. Dieser Parameter rückt Elemente, deren Metrikwert hoch ist, näher an ihr Elternelement heran als die Elemente, deren Metrikauspägung niedriger ist. Somit werden, gemäß dem *Gesetz der Nähe* aus der Gestalttheorie (Graham, 2008, vgl.), das Eltern- und das Kindelement stärker gruppiert wahrgenommen. Mit diesem Vorgehen sind innerhalb eines Elternelements recht einfach Vergleiche der unterschiedlichen Kindelemente möglich.

Ein Nachteil der abstrakten Weltraummetapher ist, ähnlich wie bei dreidimensionalen Bäumen (vgl. Robertson et al., 1991), dass die Metapher nicht sehr gut geeignet ist, um tiefe Strukturen oder Strukturen mit vielen Elementen darzustellen. Allerdings kann die abstrakte Weltraummetapher mehrere Metriken ohne Einschränkungen darstellen und bietet so auch die Möglichkeit, kritische Punkte zu entdecken, bei denen Korrelationen von mehreren Metriken auftreten. Es zeigt sich also, dass sowohl die Schachtelungsmetapher, als auch die abstrakte Weltraummetapher in bestimmten Bereichen der Metrikvisualisierung ihre Stärken ausspielen können, aber dafür in anderen Bereichen Schwächen zeigen.

6.7 Entwicklung der domänenspezifischen Sprachen

Für die Prototypentwicklung kommen zwei verschiedene domänenspezifische Sprachen zum Einsatz, eine textuelle und eine dialogbasierte DSL. Diese legen die Parameter des Generatorlaufs fest und bestimmen so verschiedene Eigenschaften des zu erstellenden Modells.

Dadurch soll es ermöglicht werden, unterschiedliche Konfigurationen zu evaluieren und für unterschiedliche Anforderungen passende Informationen visualisieren zu können.

Die Erstellung von zwei verschiedenen DSLs ist vor allem darauf zurückzuführen, dass die Einbindung des erstellten oAW-Workflows in ein Plugin erst in einer späteren Entwicklungsphase vorgenommen wurde. Der Funktionsumfang beider DSLs ist identisch. Nichtsdestotrotz bieten die beiden DSLs jeweils unterschiedliche Vorteile.

6.7.1 Textuelle DSL

Die textuelle DSL wurde im Rahmen der Prototypentwicklung zuerst entwickelt, vornehmlich deshalb, weil openArchtectureWare innerhalb des Workflows die Möglichkeit bietet, Textdateien zur Parametrisierung einzulesen und diese innerhalb des Workflows den Transformationen bereitzustellen. Die Parameter können in einer XTend-Transformation mit Hilfe der Funktion `getProperty()` ausgelesen werden. Listing 6.4 zeigt dieses Auslesen am Beispiel der Eigenschaft `radius.min`.

Bei der Festlegung der zu bestimmenden Parameter wurden einige der Parameter, die bereits in Müller (2009) verwendet wurden, übernommen, weil sie auch im Rahmen der Metrikvisualisierung zum Einsatz kommen. Die festlegbaren Parameter des Generators zeigt Tabelle 6.5. Einige der Parameter werden nur für eine der beiden bereitgestellten Metaphern benötigt. Sie müssen deshalb nur angegeben werden, wenn die Darstellung die jeweilige Metapher nutzen soll.

Listing 6.5 zeigt beispielhaft einen Ausschnitt aus einer solchen Konfigurationsdatei. Dabei ist zu beachten, dass der Wert immer vom Datentyp `String` ist. Einige Parameter verwenden auch Zahlenwerte oder Listen, wie es bei der Definition von `colour.default` zu sehen ist. Um diese Daten im Rahmen des Workflows weiterverarbeiten zu können, müssen die Werte deshalb eventuell noch geparsed werden.

Die Vorteile der textuellen DSL liegen vor allem darin, dass sie mit jedem beliebigen Texteditor erstellt werden kann und sie den verschiedenen Möglichkeiten der Textverarbeitung zugänglich ist. Die so erstellten Dateien lassen sich einfach innerhalb eines Versionsverwaltungssystems versionieren. Sie sind einfach zwischen unterschiedlichen Anwendern austauschbar und für unterschiedliche Zwecke können unterschiedliche Konfigurationen fertig vorbereitet werden. Zudem muss der korrespondierende oAW-Workflow nicht innerhalb eines Plugins ausgeführt werden.

```
cached String getMinSize():
        getProperty("radius.min");
```

Listing 6.4: Auslesen einer Eigenschaft

```
package.shape=Box
colour.default=0.3 0.3 0.3
```

Listing 6.5: Auszug aus einer Konfigurationsdatei

Variable	Einsatzzweck	Metapher
background.sky.colour	Farbe des Himmels der Szene	SW
background.ground.colour	Farbe des Bodens der Szene	SW
package.shape	Form der Pakete	SW
class.shape	Frm der Klassen	SW
attribute.shape	Form der Attribute	SW
operation.shape	Form der Methoden	SW
colour.default	Farbe der Elemente ohne Metrik	SW
colour.min	Startwert der Farbe für Elemente mit Metrik	SW
colour.max	Endwert der Farbe für Elemente mit Metrik	SW
radius.min	minimaler Radius der Objekte	SW
radius.factor	Faktor zur Bestimmung des maximalen Wertes	SW
distance.min.factor	Parameter zur Layoutkonfiguration; Bestimmt den minimalen Abstand	W
distance.max.factor	Parameter zur Layoutkonfiguration; bestimmt den maximalen Abstand	W
distance.depth.pow	Parameter zur Layoutkonfiguration; bestimmt den Einfluss der Schachtelungstiefe von Elementen	W
rotation.package.max	maximale Rotation der Elemente um Pakete	W
rotation.class.max	maximale Rotation der Elemente um Klassen	W
metric.color	Metrik zur Festlegung der Farbe	SW
metric.size	Metrik zur Festlegung der Größe	SW
metric.distance	Metrik zur Festlegung des Abstands	W
metric.yaxis	Metrik zur Festlegung der Ausrichtung entlang der y-Achse	S
metric.zaxis	Metrik zur Festlegung der Ausrichtung entlang der z-Achse	S

Tabelle 6.5: Schlüsselwörter der textuellen DSL. S = Schachtelungsmetapher, W = abstrakte Weltraummetapher

6.7.2 Dialogbasierte DSL

Die dialogbasierte DSL wurde im Rahmen der Pluginentwicklung entwickelt. Sie nutzt die Möglichkeiten von Eclipse, um die Erstellung einer Konfiguration einfacher zu gestalten (Abb. 6.5). Mit Hilfe dieser Möglichkeiten war es zum Beispiel möglich, dem Nutzer ein geeignetes Werkzeug zur Auswahl von Farben an die Hand zu geben und nicht mehr von ihm zu verlangen, die Farbwerte für jede Farbe selbst zu berechnen. Auch die einzelnen gewünschten Metriken können so komfortabel aus einem Drop-Down-Menü gewählt werden, anstatt die gewünschten Kürzel einer Liste zu entnehmen.

Der zweite Teil der DSL besteht in zwei Popup-Menü-Einträgen, die erscheinen sollen, wenn der Nutzer ein Java-Projekt in Eclipse ausgewählt hat und das Kontextmenü mit einem Rechtsklick aufruft. Diese beiden Einträge bieten die Möglichkeit zur Auswahl einer der beiden unterschiedlichen Metaphern. Grundsätzlich hätte diese Auswahl auch in der Eigenschaftsseite aufgeführt werden können. In diesem Fall wäre nur noch ein Kontext-

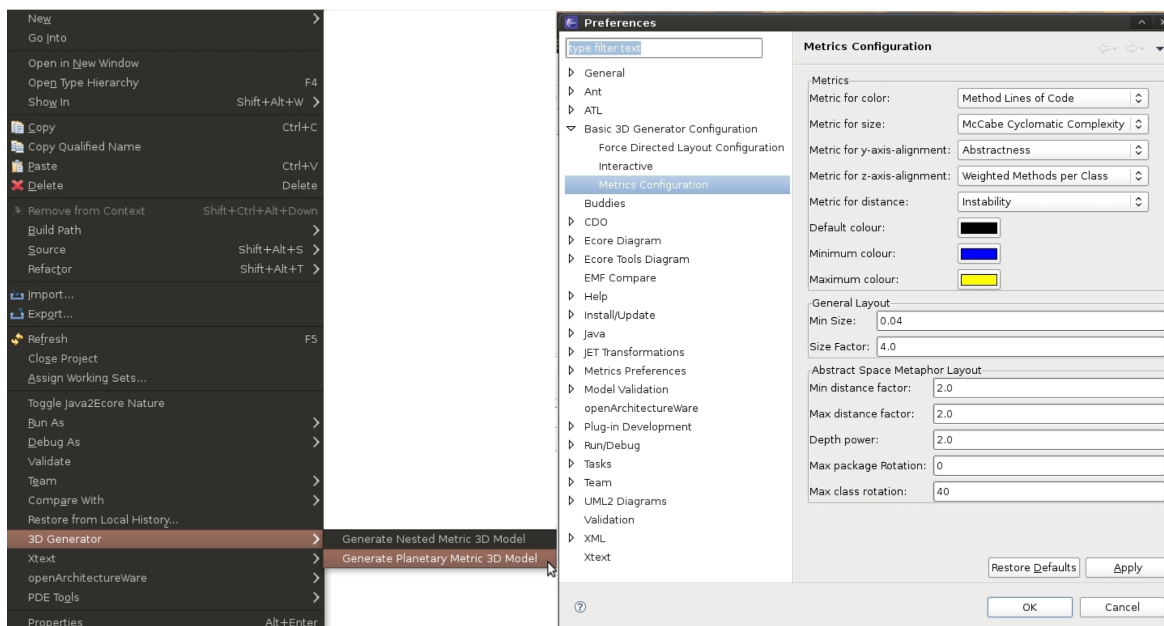

Abbildung 6.5: Dialogbasierte DSL mit Aufruf des Generators und Eigenschaftsseite

menü-Eintrag von Nöten gewesen. Der Grund für die Aufnahme dieser Auswahl in das Kontextmenü war, dass die beiden unterschiedlichen Metaphern ihre Vorteile in jeweils anders gearteten Projekten ausspielen können. So ist zum Beispiel die Schachtelungsmetapher eher geeignet, Projekte mit einer sehr tiefen Schachtelung darzustellen, wobei jedoch die Einschränkungen bezüglich der Metrikdarstellung zu berücksichtigen sind. Es erschien deshalb zweckmäßiger, bei der Arbeit mit unterschiedlichen Projekten nicht immer die Eigenschaftenseite aufrufen zu müssen, sondern bei jedem Generatorlauf entscheiden zu können, welche Metapher genutzt werden sollen. Die Parameter für die abstrakte Weltraummetapher sind auch von denen der Schachtelungsmetapher soweit verschieden, dass eine Konfiguration sowohl für die eine als auch die andere Metapher genutzt werden kann.

Die Vorteile der dialogbasierten DSL liegen in der leichteren Erstellung einer Konfiguration, da der Nutzer durch die verschiedenen Eingabeelemente zum Teil schon auf eine bestimmte Menge an Werten eingeschränkt wird und zwischen ihnen wählen kann. Falscheingaben sind somit nur bei den Dezimalwerten für das Layout möglich, da hierfür kein geeignetes Auswahlelement verfügbar war.

Der Nachteil der dialogbasierten DSL besteht vor allem darin, dass nicht mehrere Konfigurationen vorgehalten werden können, sondern nur die aktuell spezifizierte Konfiguration gespeichert werden kann. Somit besteht keine Möglichkeit, unterschiedliche Konfigurationen, die bereits vorbereitet und getestet wurden, untereinander auszutauschen. Dieser Nachteil wiegt nicht allzu schwer, da die Einstellungsmöglichkeiten nicht so umfangreich sind, als dass sich nicht in akzeptabler Zeit eine geeignete Konfiguration finden ließe. Da die Vorteile den Nachteil aufwiegen, wurde die dialogbasierte DSL als Konfigurationsmöglichkeit im erstellten Eclipse-Plugin gewählt.

6.8 Erstellung des Plugins

Die Erstellung des Plugins sollte die einzelnen Komponenten des Prototyps zusammenführen. Dadurch soll die Visualisierung vollständig automatisiert erstellt werden. Zu diesem Zweck wurden die zwei erstellten unterschiedlichen openArchitectureWare-Workflows so eingebunden, dass über unterschiedliche Einträge in einem Popup-Menü die beiden Workflows separat ausführbar sind und somit eine Auswahl der Darstellungsmetapher möglich wird.

Um die Einträge, die in der dialogbasierten DSL als Auswahlmöglichkeiten festgelegt wurden, in das Plugin einzubinden, musste der Extension Point `org.eclipse.ui.popupMenus` erweitert werden. Listing 6.6 zeigt die Erweiterung des Popup-Menüs um den Menüpunkt „Generate Planetary Metric 3D Model“. Um die Eigenschaftsseite, die ebenfalls in der dialogbasierten DSL definiert wurde, umzusetzen, musste zusätzlich der Extension Point `org.eclipse.ui.preferencePages` erweitert werden. Das Vorgehen war dabei ähnlich. Auch hier musste in der Datei `plugin.xml` eine Erweiterung des Extension Points festgelegt werden.

Der erste der verbliebenen Schritte der Generatorentwicklung ist nun der Aufruf des Codes für die Metrik- und Strukturextraktion. Den dafür zuständigen Funktionen muss dazu das gerade ausgewählte `IJavaProject` übergeben werden. Mit dieser Angabe lässt sich dann sowohl die Metrikextraktion mittels einem Ant-Target anstoßen als auch die Strukturextraktion durchführen. Das Plugin schreibt die so erstellten Dateien in einen Ordner `src-gen`, der, sofern nicht vorhanden, im Wurzelverzeichnis des Java-Projektes erstellt wird. Der zweite Schritt ist das Anstoßen des openArchitectureWare-Workflows. Listing 6.7 zeigt

```
<extension point="org.eclipse.ui.popupMenus">
  <objectContribution
     id="org.x3d.generator.plugin.contribution1"
     objectClass="org.eclipse.jdt.core.IJavaProject">
    <menu
       id="org.x3d.generator.plugin.menu"
       label="3D Generator" path="additions">
     <separator name="group"/>
    </menu>
    <action
       class="org.x3d.generator.plugin.popup.actions.
          PlanetaryGeneratorAction"
       enablesFor="1"
       id="org.x3d.generator.plugin.planetary"
       label="Generate Planetary Metric 3D Model"
       menubarPath="org.x3d.generator.plugin.menu/group">
    </action>
  </objectContribution>
</extension>
```

Listing 6.6: Erweitern des Kontextmenüs

```
WorkflowRunner runner = new WorkflowRunner();
runner.run(workflowFile, new NullProgressMonitor(), properties,
    null);
```

Listing 6.7: Aufruf eines Workflow mittels Java

das Ausführen eines Workflows, dessen Pfad in dem String `workflowFile` und dessen Eigenschaften in einer Variable `properties` vom Typ `Map` gespeichert sind. Nach diesen Schritten war es zur Fertigstellung des Plugins noch nötig, in der *Manifest*-Datei des Plugins die benötigten Abhängigkeiten festzulegen und weitere Einstellungen vorzunehmen. Dies geschah mit dem von Eclipse mitgelieferten grafischen *Manifest*-Editors. Alle Eclipse-Projekte des Generators befinden sich auf dem Datenträger, der dieser Arbeit beiliegt.

7 Fazit und Ausblick

Statische Codemetriken leisten einen großen Beitrag zur Beurteilung eines Softwaresystems in allen Bereichen des Softwarelebenszyklus. Die Vielzahl der verfügbaren Metriken zeigt auch, dass Softwarequalität ein komplexes Gebilde ist, das aus unterschiedlichen Perspektiven betrachtet werden kann und sollte. Damit wird es immer wichtiger, mehrere Metriken kombiniert betrachten zu können, um einen Gesamteindruck des Systems zu gewinnen. Die meisten Darstellungsformen von Metriken haben hier den Nachteil, sich nur auf eine Metrik zu konzentrieren. Deshalb fällt auch den Möglichkeiten der dreidimensionalen Darstellung eine große Bedeutung zu, da hiermit ein Ausbrechen aus den Beschränkungen der zweidimensionalen Darstellung möglich ist.

Die gleiche Erkenntnis trifft auch auf die Strukturdarstellung von Software zu. Auch hier werden durch die dreidimensionale Visualisierung Möglichkeiten erschlossen, die vorher versperrt waren. Die Arbeit zielte deshalb darauf ab, diese beiden Möglichkeiten zu verbinden und zu untersuchen, wie beide Informationen zusammen dargestellt werden können. Hierbei konnte gezeigt werden, dass sich statische Codemetriken grundsätzlich als Bestandteile dreidimensionaler Softwarevisualisierungen eignen.

Um eine geeignete Darstellungsform zu finden, musste evaluiert werden, welche Metaphern geeignet sind, um die gewünschte Darstellung zu ermöglichen. Hierbei konnten Hinweise gefunden werden, welche Metaphern sich für welche Arten von Softwareprojekten eignen. Eine perfekte Metapher, die für die Darstellung aller Arten von Software optimal geeignet ist, konnte dabei nicht gefunden werden. Es ist auch zu bezweifeln, dass sich eine solche Metapher in naher Zukunft finden lässt. Dies drückt sich darin aus, dass derzeit unterschiedlichste Metaphern zur Visualisierung von Software genutzt werden. Die Spannbreite reicht dabei von sehr abstrakten Darstellungen, die dem abstrakten Charakter von Software Rechnung tragen, bis hin zu Echtweltmetaphern, die den Versuch unternehmen, den abstrakten Charakter von Software auf bekannte Konzepte der Welt zu übertragen. Es konnte beobachtet werden, dass einige Metaphern gut für die Darstellung von Zusatzinformationen geeignet sind, während sich andere hierfür weniger eignen.

Des Weiteren wurde aufgezeigt, wie X3D dazu genutzt werden kann, diese Zusatzinformationen in eine Darstellung der Struktur eines Softwaresystems zu integrieren. X3D kann hierfür als sehr gute Basis angesehen werden, deren Möglichkeiten nicht bei der Strukturdarstellung enden. Vielmehr können die Möglichkeiten dazu genutzt werden, die gewünschten Zusatzinformationen mit der Strukturdarstellung zu verbinden. Es wäre wünschenswert, wenn mehr Betrachter für X3D zur Verfügung stünden und wenn die vorhandenen einen größeren Teil der Spezifikation umsetzen würden. Dies würde dazu führen, dass auch die besonderen Eigenschaften von X3D genutzt werden könnten. Zum Beispiel könnte auch Ton in einer Szene eingesetzt werden, um verschiedene Elemente hervorzuheben, ohne dass dies die Auswahl der möglichen X3D-Browser zu stark einschränkt.

X3D eignet sich durch die Wahl einer XML-Repräsentation auch zu einer automatischen Verarbeitung. Die dabei im Rahmen der Arbeit gezeigte Vorgehensweise basiert auf dem generativen Paradigma. Es konnte gezeigt werden, dass sich die von Müller (2009) vorgenommene Adaption des generativen Paradigmas um die für die Zielstellung dieser Arbeit benötigten Elemente erweitern lässt. Mit der Wahl des generativen Paradigmas wurde eine weitestgehende Vollautomatisierung angestrebt, die auch erreicht werden konnte.

Des Weiteren wurde aufgezeigt, wie Techniken aus der modellgetriebenen Softwareentwicklung genutzt werden können, um die verschiedenen Informationen aufzubereiten und zusammenzuführen. OpenArchitectureWare erwies sich hierbei als sehr gut geeignetes Werkzeug, um die Modelltransformationen und Modell-Merging-Schritte durchzuführen. Durch die von openArchitectureWare bereitgestellten Mittel zur Modularisierung eines Arbeitsablaufs ist auch eine Weiterentwicklung des Prototyps in verschiedenen Bereichen möglich.

Der im Rahmen dieser Arbeit erstellte Prototyp kann in seinem jetzigen Zustand für die kombinierte Visualisierung von Metrik- und Strukturinformationen genutzt werden, stellt aber kein fertiges Produkt dar. Er liefert weiterhin eine gute Grundlage für weitere Arbeiten. Da eine solche Arbeit nur einen begrenzten Rahmen zur Verfügung stellt, konnten auch nicht alle Ideen, die während der Entwicklung das Prototyps aufkamen, umgesetzt werden. Zu diesen Ideen gehören zuvorderst die Evaluation von Möglichkeiten, wie die Darstellungsmöglichkeiten von dreidimensionalen Visualisierungen besser für die Verbindung von Metrik- und Strukturvisualisierung genutzt werden kann. Diese Überlegungen im Speziellen gehen vor allem in drei Richtungen: die Anpassung der bestehenden Metaphern, die Entwicklung neuer Metaphern und die Modifikation der Metrikabbildungen.

Bei der Anpassung der bestehenden Metaphern ergeben sich unterschiedliche Ansatzpunkte. Als Beispiel wäre hier die Verwendung der Rotation als Informationsträger zu nennen. Diese wurde bereits in Abschnitt 6.6.2 diskutiert und dort für die hier vorliegende Arbeit verworfen. Durch unterschiedliche Adaptionen der Metrikabbildungen könnte sie aber durchaus genutzt und umgesetzt werden. Bei der Adaption der Metrikabbildungen müsste darauf geachtet werden, dass den zu rotierenden Elementen grundsätzlich ein Wert zugewiesen wird. Sonst würde eine flache Darstellung entstehen, die sehr viel Platz verbraucht und die Möglichkeiten der dritten Dimension nur unzureichend nutzt. Eine weitere mögliche Anpassung der Weltraummetapher wäre, das Prinzip des kreisförmigen Anordnens der Elemente um ihr Elternelement für ausgewählte Elementkategorien zu durchbrechen. So könnten beispielsweise die Pakete in einem dreidimensionalen Raster angeordnet werden, das in geeigneter Weise die logische Schachtelung wiedergibt. Durch ein solches Vorgehen könnte der Platz, der in den drei Dimensionen zur Verfügung steht, besser genutzt werden. Es gibt sicherlich noch deutlich mehr Möglichkeiten, die im Prototyp bereitgestellten Metaphern zu verändern oder andere Metaphern zu entwickeln.

Auch bei der Anpassung der Metrikabbildungen kamen während der Entwicklung des Prototyps unterschiedliche Ansätze auf, die ebenfalls eine Betrachtung wert sind. Als Bei-

spiel seien hier die verschiedenen möglichen Abbildungen der Metrikwerte auf eine andere Skaleneinteilung genannt. Die Metrikwerte des Prototyps werden ausschließlich auf eine lineare Skala abgebildet. Dies muss aber nicht in allen Fällen der optimale Weg sein. So kann es Werte geben, die besser logarithmisch oder exponentiell skaliert repräsentiert werden. Es wäre auch möglich, die Messwerte in Kategorien einzuteilen, die dann auf unterschiedliche Weise dargestellt werden könnten. Eine weitere interessante Möglichkeit, das Finden von Schwachpunkten in einer zu untersuchenden Software zu erleichtern, wäre die Aggregation von Metrikwerten in den übergeordneten Elementen. Einige Metriken, wie zum Beispiel die Metrik *Weighted Methods per Class*, sind bereits so gestaltet, dass sie die Werte der Unterelemente aufsummieren. Die Möglichkeit der Summenbildung könnte auf die meisten Metriken erweitert werden. Diese Summierungen könnten zum Beispiel ein Paket größer darstellen oder stärker einfärben, wenn die Summe der Metrikwerte der in diesem Paket enthaltenen Elemente höher ist. Dies wäre vor allem im Bereich der Schachtelungsmetapher sehr hilfreich, da auf diese Weise die Elemente, in denen Problemstellen liegen, hervorgehoben werden. Der Nutzer könnte daraufhin in die entsprechenden Elemente eintauchen und sie näher untersuchen.

Eine andere mögliche Erweiterung des Prototyps ist die Darstellung von dynamischen Informationen oder beliebigen weiteren Informationen, die für die einzelnen Elemente eines Softwaresystems verarbeitbare Werte liefern. Es gibt eine Reihe dieser Informationen, die in den verschiedenen Abschnitten des Softwarelebenszyklus erhoben werden. Beispiele könnten hier das Laufzeitverhalten oder die gemessene Häufigkeit von Änderungen an einzelnen Komponenten darstellen. Die Einbindung solcher Informationen in den Softwarevisualisierungsprozess des Prototyps sollte relativ einfach möglich sein, wenn die Werte zusammen mit einem eindeutigen Bezeichner gespeichert sind. Von Vorteil wäre auch eine XML-Repräsentation der Daten, da die so vorliegenden Daten sehr einfach in den openArchitectureWare-Ablauf eingebunden werden könnten.

Eine deutlich weiter gefasste Erweiterung wäre die Abbildung unterschiedlicher Entwicklungsstände von Software in einer Visualisierung. Hierbei könnte von den Möglichkeiten der Animation in X3D Gebrauch gemacht werden. Diese Animationen könnten auch zur Darstellung des Laufzeitverhaltens eines Systems genutzt werden. Dabei wären zum Beispiel die Darstellung des Programmablaufs oder die Kommunikationen der einzelnen Elemente untereinander interessant.

Ein weiterer Ansatzpunkt zur Erweiterung des Szenegraphs von X3D ist zum Beispiel die Erforschung der Möglichkeiten zur Interaktion mit einem dreidimensionalen Modell. Diesen Bereich untersucht derzeit Kovacs im Rahmen seiner Diplomarbeit (vgl. Kovacs, 2010). Diese Ansätze könnten dann wiederum bei der Entwicklung von Roundtrip-Engineering-Möglichkeiten weiterverarbeitet werden. Damit würde dann nicht nur der Weg von der Software zur Visualisierung offen stehen, sondern auch der Weg von der Visualisierung zur Software. Die grafischen Modelle könnten dann gleichbedeutend mit Quellcode sein, was auch ein Ziel der modellgetriebenen Softwareentwicklung ist (vgl. Stahl et al., 2007).

Neben den Möglichkeiten zur Interaktion und Animation bietet X3D noch viel Funktionalität an, die perspektivisch der einfacheren Zugänglichkeit von Software dienen könnte. So wäre beispielsweise der Einsatz von Audio in einer Softwarevisualisierung ein sehr interessanter Punkt, aber auch die Einsatzmöglichkeiten von Videos bei der Visualisierung von Software wären ein sehr verlockendes Untersuchungsfeld.

All diese Möglichkeiten könnten und sollten noch weiter erforscht werden. Die Fülle der Ansatzpunkte zeigt aber auch, dass die Forschung im Bereich der Softwarevisualisierung erst am Anfang steht. Dieser vergleichsweise junge Forschungszweig bietet dennoch zahlreiche Perspektiven. Die Weiterentwicklungen in diesem Gebiet führen hoffentlich zu einem besseren und leichteren Verständnis von Software und damit zu effizienterer Softwareentwicklung, einfacherer Softwarewartung und einer Erhöhung der Softwarequalität allgemein.

Literaturverzeichnis

Anslow, C., Marshall, S., Noble, J., und Biddle, R.: Evaluating X3D for use in software visualization. In: Proceedings of the 2006 ACM symposium on Software visualization, Brighton, UK, S. 161–162, 2006

Bach, M.: Spaziergang: Eclipse-Plug-in für die Quelltextanalyse implementieren. *iX*, (7), S. 138–141, 2009

Balzer, M. und Deussen, O.: Hierarchy based 3D visualization of large software structures. In: Proceedings of the conference on Visualization ’04, S. 598.4, 2004

Basili, V. und Weiss, D.: A methodology for collecting valid software engineering data. IEEE Transactions on Software Engineering, 1, S. 728–738, 1984

Battista, G. D., Eades, P., Tamassia, R., und Tollis, I. G.: Graph Drawing: Algorithms for the Visualization of Graphs. Prentice-Hall, Upper Saddle River, NJ, 1999

Behr, J., Dähne, P., und Roth, M.: Utilizing X3D for immersive environments. In: Proceedings of the ninth international conference on 3D Web technology, Monterey, California, S. 71–78, 2004

Bitmanagement (2010). BS Contact – Bitmanagement – Interactive Web3D Graphics – Visualization for VRML, X3D, Collada, Kmz, CityGML. http://www.bitmanagement.de/en/products/interactive-3d-clients/bs-contact, abgerufen am: 12.02.2010

Böhme, R. und Freiling, F.: On metrics and measurements. In: Dependability Metrics. Springer-Verlag, Berlin Heidelberg, S. 7–13, 2008

Budinsky, F.: The eclipse modeling framework: Moving into model-driven development. http://www.drdobbs.com/linux-open-source/184406198, abgerufen am: 14.02.2010, 2005

Czarnecki, K.: Overview of generative software development. In: Unconventional Programming Paradigms, Lecture Notes in Computer Science, Nr. 3566, S. 326–341. Springer, Berlin Heidelberg, 2005

Czarnecki, K. und Eisenecker, U. W.: Generative Programming. Methods, Tools and Applications: Methods, Techniques and Applications. Addison-Wesley Longman, Amsterdam, 2000

Czarnecki, K. und Helsen, S.: Classification of model transformation approaches. In: OOPSLA 03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003

dos Santos, S. und Brodlie, K.: Gaining understanding of multivariate and multidimensional data through visualization. Computers & Graphics, 28(3), S. 311–325, 2004

Dwyer, T.: Three dimensional UML using force directed layout. In: Proceedings of the 2001 Asia-Pacific symposium on Information visualisation - Volume 9, S. 77–85. Australian Computer Society, Inc., Sydney, Australia, 2001

Efftinge, S., Friese, P., Haase, A., Hübner, D., Kadura, C., Kolb, B., Köhnlein, J., Moroff, D., Thoms, K., Völter, M., und Schönbach, P.: Workflow reference. http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/workflow_reference.html, abgerufen am: 29.01.2010, 2008

Eclipse Foundation (Hrsg.): Eclipse membership. http://www.eclipse.org/membership/exploreMembership.php, abgerufen am: 17.02.2010, 2010

Fowler, M.: DomainSpecificLanguage. http://www.martinfowler.com/bliki/DomainSpecificLanguage.html, abgerufen am: 26.01.2010

Fraunhofer IDG (Hrsg.): instantreality 1.0 - home. http://www.instantreality.org/home/, abgerufen am: 12.02.2010

Geroimenko, V. und Geroimenko, L.: SVG and X3D: new XML technologies for 2D and 3D visualization. In: Visualizing the Semantic Web, S. 124–133. Springer, Berlin, 2006

Ghezzi, C., Jazayeri, M., und Mandrioli, D.: Fundamentals of Software Engineering, 2. Auflage. Prentice Hall, Upper Saddle River, NJ, 2002

Gračanin, D., Matković, K., und Eltoweissy, M.: Software visualization. Innovations in Systems and Software Engineering, 1(2), S. 221–230, 2005

Graham, L.: Gestalt theory in interactive media design. Gestalt Theory, 2(1), S. 1–12.

Graham, H., Yang, H. Y., und Berrigan, R.: A solar system metaphor for 3D visualisation of object oriented software metrics. In: Australasian Symposium on Information Visualisation, 2004

Haber, R. B. und McNabb, D. A.: Visualization idioms: A conceptual model for scientific visualization systems. In: Proceedings of IEEE ’90 Visualization, S. 74–93, 1990

Halstead, M. H.: Elements of Software Science. North-Holland, New York, 1977

Hoffmann, D. W.: Software-Qualität. Springer, Berlin Heidelberg, 2008.

IBM Corporation (Hrsg.): Eclipse documentation: Plug-in development environment guide. http://help.eclipse.org/galileo/nav/4_1, abgerufen am: 11.02.2010

Knight, C. und Munro, M.: Comprehension with[in] virtual environment visualisations. In: International Conference on Program Comprehension, 1999

Kovacs, P.: Ansätze zur Interaktion mit dreidimensional visualisierten Softwaremodellen. Universität Leipzig, Leipzig, bisher unveröffentlichte Diplomarbeit, 2010

Koziolek, H.: Goal, question, metric. In: Dependability Metrics, Lecture Notes in Computer Science, Nr. 4909, S. 39–42. Springer, Berlin Heidelberg, 2008.

Laird, L. M. und Brennan, M. C.: Software Measurement and Estimation: A Practical Approach, 1. Aufl. Quantitative Software Engineering Series. John Wiley & Sons, Hoboken, New Jersey, 2006

Lakoff, G.: Leben in Metaphern: Konstruktion und Gebrauch von Sprachbildern 4. Aufl. Carl-Auer-Systeme-Verlag, Heidelberg, 2004

Liggesmeyer, P.: Software-Messung. In: Software-Qualität, 2. Aufl., S. 231–268. Spektrum Akademischer Verlag, Heidelberg, 2009

Liggesmeyer, P., Heidrich, J., Münch, J., Kalcklösch, R., Barthel, H., und Zeckzer, D.: Visualization of software and systems as support mechanism for integrated software project control. In: Human-Computer Interaction. New Trends, S. 846–855. Springer, Berlin Heidelberg, 2009

Mäkelä, S. und Leppänen, V. (2006). Observations on lack of cohesion metrics. In: International Conference on Computer Systems and Technologies - CompSysTech '06, 2006

Maletic, J. I., Leigh, J., Marcus, A., und Dunlap, G.: Visualizing Object-Oriented software in virtual reality. In: Proceedings of the 9th International Workshop on Program Comprehension, S. 26. 2001

Maletic, J. I., Marcus, A., und Collard, M. L.: A task oriented view of software visualization. In: VISSOFT 2: Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis, S. 32–40. 2002

McCabe, T. J.: A complexity measure. IEEE Transactions on Software Engineering, SE-2(4), S. 308–320, 1976

McIntosh, P., Hamilton, M., und van Schyndel, R.: X3D-UML: enabling advanced UML visualisation through X3D. In: Proceedings of the tenth international conference on 3D Web technology, Bangor, United Kingdom, S. 135–142, 2005

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

Musa, J. D.: Software Reliability: Measurement, Prediction, Application, 1. Aufl. McGraw-Hill, New York, 1987

Octaga: Octaga - bringing enterprise data to life - home. http://www.octaga.com/, abgerufen am: 12.02.2010, 2010

Object Modelling Group (OMG) (Hrsg.): MOF 2.0. http://www.omg.org/spec/MOF/2.0/PDF/, abgerufen am: 14.02.2010, 2006

Omondo Corp. (Hrsg.): Omondo corp - the modeling eclipse UML model driven tool. http://www.omondo.com/, abgerufen am: 14.02.2010, 2010

Panas, T., Epperly, T., Quinlan, D., Saebjornsen, A., und Vuduc, R.: Communicating software architecture using a unified Single-View visualization. In: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems, S. 217–228, 2007

Pietrek, G. und Trompeter, J.: Modellgetriebene Softwareentwicklung: MDA und MDSD in der Praxis. entwickler.press, Frankfurt am Main, 2007

Price, B. A., Baecker, R., und Small, I. S.: A principled taxonomy of software visualization. J. Vis. Lang. Comput., 4(3), S. 211–266, 1993

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

Robertson, G. G., Mackinlay, J. D., und Card, S. K.: Cone trees: animated 3D visualizations of hierarchical information. In: CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems, S. 189–194, 1991

Roman, G. und Cox, K. C.: A taxonomy of program visualization systems. Computer, 26(12), S. 11–24, 1993

Sauer, F.: Metrics 1.3.6. http://metrics.sourceforge.net/, abgerufen am: 28.12.2009

Stahl, T., Völter, M., Efftinge, S., und Haase, A.: Modellgetriebene Softwareentwicklung - Techniken, Engineering, Management. dpunkt.verlag, Heidelberg, 2007

van Solingen, R. und Berghout, E.: The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development. McGraw-Hill Professional, New York, 1999

Vogel, L.: Eclipse JDT - abstract syntax tree (AST) and the java model - tutorial. http://www.vogella.de/articles/EclipseJDT/article.html, abgerufen am: 26.01.2010, 2009

Web3D Consortium (Hrsg.): Extensible 3D (X3D) part 1: Architecture and base components. http://www.web3d.org/x3d/specifications/ISO-IEC-19775-1.2-X3D-AbstractSpecification/index.html, abgerufen am: 26.01.2010, 2008

Wettel, R. und Lanza, M.: Visualizing software systems as cities. In: 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis – VISSOFT 2007, S. 92–99, 2007

Wikipedia (Hrsg.): Datei:M0-m3.png – wikipedia. http://de.wikipedia.org/w/index.php?title=Datei:M0-m3.png&filetimestamp=20080909201009, abgerufen am: 29.01.2010, 2008

Wikipedia (Hrsg.): Datei:Netzdiagramm-Beispielp.svg – wikipedia.http://de.wikipedia.org/w/index.php?title=Datei:Netzdiagramm-Beispielp.svg&filetimestamp=20090220210856, abgerufen am: 06.01.2010, 2009

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Diplomarbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommene Gedanken sind als solche kenntlich gemacht. Bei der Auswahl und Auswertung des Materials sowie bei der Herstellung des Manuskripts habe ich Unterstützungsleistungen von folgenden Personen erhalten:

1. Prof. Dr. Ulrich W. Eisenecker
2. Anja Tausendfreund (Korrekturlesen)
3. Lars Rohne (Korrekturlesen)

An der geistigen Herstellung der vorliegenden Diplomarbeit war außer mir niemand beteiligt. Insbesondere habe ich nicht die Hilfe eines Diplomberaters in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorliegenden Diplomarbeit stehen. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form oder auszugsweise einer Prüfungsbehörde vorgelegt.