

Universität Leipzig
Wirtschaftswissenschaftliche Fakultät
Institut für Wirtschaftsinformatik
Prof. Dr. Ulrich W. Eisenecker
Richard Müller

Thema

Generative und modellgetriebene Softwarevisualisierung am Beispiel der Stadtmetapher

Bachelorarbeit zur Erlangung des akademischen Grades
Bachelor of Science - Wirtschaftsinformatik

vorgelegt von: Zilch, Denise

Matrikelnummer: 2880950

Leipzig, den 3. Februar 2015

Abstract

Für den Visualisierungsgenerator der Forschungsgruppe „Softwarevisualisierung in drei Dimensionen und virtueller Realität“ soll eine Stadtmetapher zur Darstellung von Software implementiert werden. Als Vorlage dient „CodeCity“, dessen Umsetzung der Stadtmetapher auf den Generator übertragen werden soll. Die Anforderungsermittlung basiert auf der Analyse beider Bestandteile, um ein strukturiertes Vorgehen zu gewährleisten. Die Implementierung der Generatorartefakte erfolgt mittels Xtext zur Erstellung eines Metamodells, das die Entitäten der neuen Metapher beschreibt, und Xtend, das genutzt wird um die Datenmodelle zu modifizieren und in Quelltext umzuwandeln.

Darauf aufbauend folgt abschließend die Abstraktion zu einem Prozessmodell für die generative und modellgetriebene Softwarevisualisierung, das als Leitfaden für zukünftige Implementierungen dienen soll.

Schlüsselwörter

Stadtmetapher, generative und modellgetriebene Softwareentwicklung, „CodeCity“, Prozessmodell

Gliederung

Gliederung	I
Abbildungsverzeichnis	III
Tabellenverzeichnis	IV
Verzeichnis der Listings	V
Abkürzungsverzeichnis	VI
1 Einleitung	1
1.1 Motivation und Problemstellung	1
1.2 Zielstellung der Arbeit.....	1
1.3 Aufbau der Arbeit.....	2
2 Grundlagen des Visualisierungsgenerator	4
2.1 Generative und modellgetriebene Softwareentwicklung.....	4
2.2 FAMIX	6
2.3 Xtext und Xtend	6
2.4 X3D	7
3 Implementierung des Prototyps	8
3.1 Analyse der Zielmetapher.....	8
3.1.1 Grundlagen von „CodeCity“	8
3.1.2 Anforderungen.....	9
3.1.3 Analyseergebnisse	13
3.2 Auswahl und Analyse der Referenzmetapher	13
3.2.1 Grundlagen der Referenzmetapher	15
3.2.2 Erweiterung der Anforderungen	16
3.3 Das Metamodell.....	18
3.4 Der Workflow	20
3.5 Modell-zu-Modell-Transformation	22
3.6 Modellmodifikation	23
3.7 Modell-zu-Text-Transformation.....	27
3.8 Anpassungen und Ergänzungen.....	28
4 Abstrahiertes Prozessmodell	31
5 Zusammenfassung und Ausblick	34
Anhang A – Metamodell Recursive Disk-Metapher	VII

Anhang B – Hilfestellung für Eclipse-Konfigurationen.....	VIII
Anhang C – Konzepte zur Durchführung der Modellmodifikation	X
Anhang D – Entwicklungsstadien der Stadtmetapher	XII
Quellen- und Literaturverzeichnis	XIV
Ehrenwörtliche Erklärung.....	XVI

Abbildungsverzeichnis

Abbildung 2.1: Generatives Softwarevisualisierungsdomänenmodell.....	4
Abbildung 3.1: „Freemind“ visualisiert durch „CodeCity“, Konfiguration „Magnitude“ ..	10
Abbildung 3.2: Das „View Configuration“-Werkzeug	10
Abbildung 3.3: Metaphern des Visualisierungsgenerators.....	14
Abbildung 3.4: Klassendiagramm der Stadtmetapher	17
Abbildung 3.5: „CodeCity“ Pseudocode	24
Abbildung 3.6: Beispiel: Vier Elemente (oben) angeordnet (Mitte) mittels KD-Baum (unten)	25
Abbildung 4.1: Abstrahiertes Prozessmodell basierend auf der Implementierung der Stadtmetapher	32
Abbildung A.1: Klassendiagramm der Referenzmetapher mit Attributen	VII
Abbildung B.1: Eclipse-Assistent zur Erstellung eines neuen Xtext-Projekts.....	VIII
Abbildung B.2: Projekt-Abhängigkeiten über MANIFEST.MF hinzufügen	IX
Abbildung C.1: Hilfsklassen des Layout-Algorithmus (Entity-Klasse vereinfacht)	X
Abbildung D.1: Zusammenfassende Beispielvisualisierung in X3D	XII
Abbildung D.2: Visualisierung nach der Implementierung der Kernaspekte	XII
Abbildung D.3: Visualisierung nach der Implementierung von Abständen	XII
Abbildung D.4: Visualisierung nach der Implementierung von Farbgradienten	XIII

Tabellenverzeichnis

Tabelle 3.1: Standardwerte der Elementattribute in „Magnitude“	11
Tabelle 3.2: Standardwerte der Anordnung von Kindelementen in „Magnitude“	12

Verzeichnis der Listings

Listing 3.1: Vererbungsbeziehung - Entity wird zur Superklasse	18
Listing 3.2: Implementierung von District	18
Listing 3.3: Implementierung der „Container“-Elemente	19
Listing 3.4: Workflow-Komponenten zur Modell-zu-Modell-Transformation	20
Listing 3.5: Workflow-Komponenten zur Modell-zu-Text-Transformation.....	21
Listing 3.6: Suchoperation für FAMIX-Namensräume der ersten Hierarchieebene.....	22
Listing 3.7: Operation zur Erzeugung einer neuen X3D-Datei	27
Listing 3.8: Datenübertragung auf X3D	28
Listing C.1: Implementierung des Sortieralgorithmus für Rectangle	X
Listing C.2: Berechnung absoluter Elementpositionen unter Berücksichtigung von Abständen	XI

Abkürzungsverzeichnis

3D	dreidimensional, drei Dimensionen
GSVDM	Generatives Softwarevisualisierungsdomänenmodell
ID	Identity Mapping
KD	k-dimensional
LIN	Linear Mapping
LOC	Quellecodezeilen (engl. <i>lines of code</i>)
NOA	Anzahl der Attribute (engl. <i>number of attributes</i>)
NOM	Anzahl der Methoden (engl. <i>number of methods</i>)
VRML	Virtual Reality Modeling Language
X3D	Extensible 3D
XML	Extensible Markup Language

1 Einleitung

Durch die Arbeit der Forschungsgruppe „Softwarevisualisierung in drei Dimensionen (3D) und virtueller Realität“ an der Universität Leipzig entstand ein Generator basierend auf den Prinzipien der generativen und modellgetriebenen Softwareentwicklung. Dieser Visualisierungsgenerator analysiert Softwareartefakte, um in ihnen enthaltene Informationen zu extrahieren. Durch ein Mapping auf eine Metapher wird eine graphische Darstellung erzeugt, um die Komplexität zu reduzieren und Zusammenhänge von Softwarekomponenten anschaulich zu machen.

1.1 Motivation und Problemstellung

Softwarevisualisierung gewinnt immer mehr an Bedeutung, um Software nicht mehr nur anzuwenden, sondern auch besser analysieren zu können und damit einhergehend Verständnis, Weiterentwicklung, Qualität, Wartung und Fehlererkennung zu verbessern. Eigenschaften wie Komplexität, Robustheit und Modularität lassen sich schwer definieren und noch schwerer ermitteln. Somit sollen durch Softwarevisualisierung Kosten- bzw. Zeiteinsparungen realisiert werden, was als direkte Konsequenz eine Steigerung von Produktivität und Qualität zur Folge hätte (vgl. [Bassil & Keller 2001]). Um in diesem Zusammenhang empirische Studien durchführen zu können, benötigt der Visualisierungsgenerator mehrere Darstellungsmöglichkeiten. Anhand der Ergebnisse dieser Untersuchungen kann man darauf schließen, wo die Vor- und Nachteile der Visualisierungsmetaphern liegen.

Der Generator erzeugt momentan Strukturvisualisierungen, denen die Schachtelungs-, die Weltraum- und die Recursive Disk-Metapher zugrunde liegen. Im Verlauf dieser Arbeit soll eine weitere Alternative zur Darstellung implementiert werden, um das Repertoire zu erweitern.

1.2 Zielstellung der Arbeit

Im Rahmen dieser Bachelorarbeit wird die von Wettel und Lanza entworfene Softwarevisualisierung „CodeCity“ (vgl. [Wettel 2010]) als Metapher mittels des generativen und modellgetriebenen Ansatzes umgesetzt. Basierend auf dieser Umsetzung soll ein Leitfaden entstehen, der das Implementieren und/oder Anpassen von zukünftigen Metaphern erleichtern soll. Dabei wurde zu Beginn schon festgelegt, welche Eigenschaften der „CodeCity“-Visualisierung übernommen werden sollen. Aufgrund dessen entfällt eine Abhandlung über existierende Visualisierungen und welche grundlegenden Unterscheidungen existieren, da dies schon in vorangegangenen Arbeiten behandelt wurde (vgl. [Müller 2009], [Schilbach

2010]). Ebenso wird in dieser Arbeit nicht der Fragestellung nachgegangen, welche Vor- und Nachteile die zu implementierende Visualisierung birgt.¹

Die bereits implementierte Recursive Disk-Metapher bietet einen wichtigen Referenzpunkt, an deren Aufbau sich orientiert wird. Dabei sind zwei Abschnitte zu unterscheiden. Zum einen der Code, der die erforderlichen Ausgangsdaten einliest, aufbereitet und somit das erforderliche Grundgerüst der Visualisierung aufbaut. Zum anderen die eigentliche Nutzung der erhaltenen und ggf. erzeugten Daten, um die Visualisierung zu generieren. Während der erste Code-Teil für alle vorhandenen und noch folgenden Metaphern sehr ähnlich ist, ist der zweite Teil sehr spezifisch und der Kern der Implementierung, da die verwendeten Algorithmen abhängig von der Metapher sind.

Somit impliziert die angestrebte Abstraktion Einfachheit. Das heißt, dass nicht nur der Prozess der Implementierung selbst, sondern auch der Gegenstand der Implementierung klar begrenzt sein muss. Damit ein hilfreicher Leitfaden zur Implementierung einer Visualisierungsmetapher entstehen kann, werden nur die damit verbundenen grundlegenden Aspekte abgedeckt, die für die Visualisierung essentiell sind.

Die Zielstellung dieser Arbeit ist somit, den grundlegenden Implementierungsprozess einer Softwaremetapher zu abstrahieren, die nach den Prinzipien generativer und modellgetriebener Softwareentwicklung umgesetzt wurde. Dies erfolgt durch die Implementierung einer Stadtmetapher, die sich an „CodeCity“ orientiert, um eine Orientierungshilfe für zukünftige Umsetzungen zu schaffen.

1.3 Aufbau der Arbeit

Nach der Vorstellung einleitender Sachverhalte wird nun kurz das Vorgehen und damit einhergehend der Aufbau dieser Arbeit erläutert, um einen Überblick über die zu behandelnde Materie zu geben.

Diese Arbeit nutzt den bestehenden Visualisierungsgenerator, weshalb zu Beginn in Kapitel 2 seine grundlegenden Technologien vorgestellt werden und wie aus einer vorliegenden Datenmenge ein dreidimensionales Modell entsteht. Kapitel 3 befasst sich mit der Implementierung der Stadtmetapher. Dieser Vorgang unterteilt sich in die Analyse der Metaphern (s. Abschnitte 3.1 und 3.2), um den Aufbau der Implementierung zu entwerfen, und die Ausarbeitung der Generatorartefakte (s. Abschnitte 3.3 bis 3.7), z.T. unter Verwendung von Quelltextauszügen zur Verdeutlichung. Hierbei werden zuerst die Kerninhalte der Metapher implementiert, damit die Durchführung der Tests vereinfacht wird. Anschließend erfolgt in

¹ Auch unter Stadtmetaphern gibt es verschiedene Darstellungsweisen. Panas u. a. [2007] sind zu einer Stadtmetapher gelangt, die einen komplett anderen Ansatz verfolgt als „CodeCity“ (vgl. [Panas u. a. 2007]).

Abschnitt 3.8 die Modifikation der Darstellung, um die Visualisierung so gut wie möglich an „CodeCity“ anzupassen. Aus dieser Implementierungsarbeit geht in Kapitel 4 ein abstrahierter Prozessablauf hervor, der den Implementierungsvorgang in der generativen und modellgetriebenen Softwarevisualisierung auf Basis eines Generators verallgemeinern soll. Abschließend werden in Kapitel 5 die Ergebnisse der Arbeit zusammengefasst, weiterführende Gedanken erläutert und dargelegt, welche weiteren Aspekte von dieser Basis ausgehend umgesetzt werden können.

2 Grundlagen des Visualisierungsgenerator

Um aufzuzeigen, wie der Generator arbeitet, werden zunächst die grundlegenden Konzepte im Zusammenhang erklärt, daraufhin werden die wichtigsten Technologien, die für die Implementierung der Stadtmetapher essentiell sind, etwas genauer betrachtet.

Der Generator basiert auf dem Generativen Softwarevisualisierungsdomänenmodell (engl. *Generative Software Visualization Domain Model*, GSVD) (vgl. Abbildung 2.1), das sich am Generativen Domänenmodell (vgl. [Czarnecki & Eisenecker 2000, 132]) orientiert.

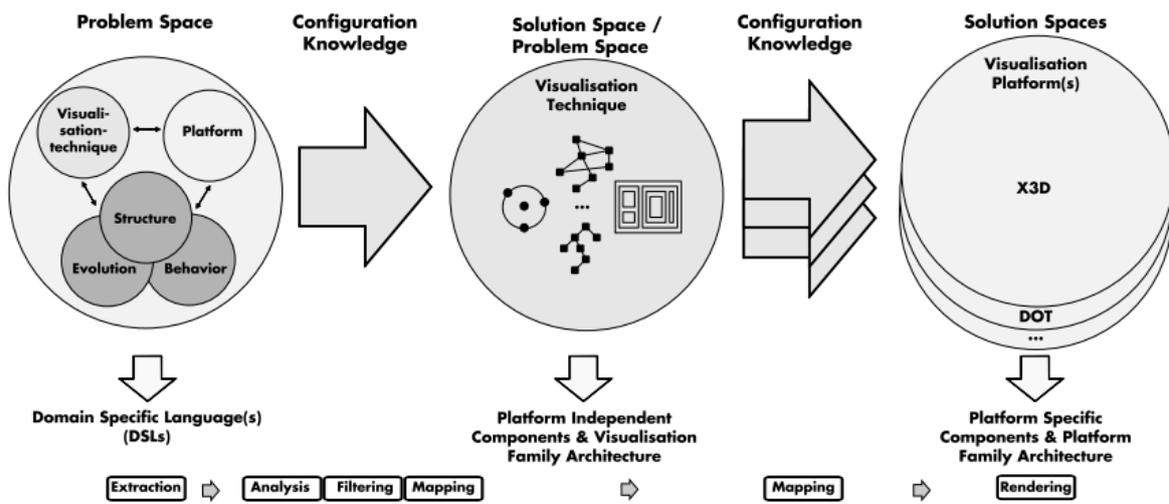


Abbildung 2.1: Generatives Softwarevisualisierungsdomänenmodell (vgl. [Müller u. a. 2011])

Der Problemraum stellt den Ausgangspunkt dar. Er umfasst das zu visualisierende System mitsamt Struktur, Verhalten, Historie, und der zu verwendeten Metapher. Diese Daten bezieht der Generator aus generierten Quellen. Nach möglichen Analysen und Filterungen der Ausgangsdaten, erzeugt er durch sein Konfigurationswissen eine Abbildung auf den ersten Lösungsraum aus plattformunabhängigen Komponenten. Das Konfigurationswissen umfasst in diesem Kontext im Besonderen das Mapping, die Zuweisungsregeln, um die vorhandenen Daten den Komponenten zuteilen zu können. Im zweiten Schritt wird der Lösungsraum zum neuen Problemraum und durch eine weitere Zuweisung auf den zweiten Lösungsraum abgebildet. Dieser besteht aus plattformspezifischen Komponenten zur Darstellung der Visualisierung (vgl. [Müller u. a. 2011]).

2.1 Generative und modellgetriebene Softwareentwicklung

Das GSVD basiert auf zwei Konzepten, die nachfolgend kurz erläutert werden sollen. Zum einen die generative Programmierung, aus der das Domänenmodell und der Begriff des Konfigurationswissens entlehnt und angepasst wurden, und zum anderen die modellgetriebene

Softwareentwicklung, deren Verwendung von Modellen bzw. (Meta-)Metamodellen² zur Strukturierung der Problem- und Lösungsräume dient (vgl. [Müller u. a. 2011]).

Die generative Programmierung nach Czarnecki & Eisenecker [2000] beschäftigt sich damit, dass sich die Softwareentwicklung einem Wandel von der händischen Produktion von Einzelsystemen hin zu einer automatisierten Erstellung von Softwaresystemfamilien zu unterziehen hat. Dabei wird hervorgehoben, dass äußerst angepasste und optimierte, generierte Produkte erstellt werden sollen (vgl. [Czarnecki & Eisenecker 2000, 5]).

Als Basis für dieses Prinzip werden Systeme erzeugt, deren Aufbau ähnlich genug ist, um sie aus demselben Repertoire von Grundkomponenten zusammensetzen (vgl. [Czarnecki & Eisenecker 2000, 8]). Der zweite Teil des Wandels besteht aus dem automatisierten Zusammenstellen einer konkreten Software der Systemfamilie durch einen Generator (vgl. [Czarnecki & Eisenecker 2000, 1]). Dieser verfügt über das Konfigurationswissen (engl. *configuration knowledge*), das Informationen über die einzelnen Komponenten bereithält, z.B. zulässige Kombinationen, durch die Auswahl zu beachtende Abhängigkeiten und Möglichkeiten für Optimierungen (vgl. [Czarnecki & Eisenecker 2000, 6]). Der Generator in der generativen Programmierung überführt, ebenso wie im GSVDm gezeigt, den Problemraum anhand des definierten Konfigurationswissens in den Lösungsraum (vgl. [Czarnecki & Eisenecker 2000, 132]).

Die modellgetriebene Softwareentwicklung ergänzt den Visualisierungsgenerator durch formale Modelle. Sie beinhalten die Aussagen über eine klar definierte Menge an Aspekten einer Domäne (vgl. [Stahl u. a. 2007, 11f]). Das Ziel der modellgetriebenen Softwareentwicklung ist es, aus diesen Modellen lauffähige Software automatisiert herzustellen. Wo zuvor Modellierungssprachen, wie z.B. die Unified Modeling Language, genutzt wurden, um sozusagen „Quelltext-Rohfassungen“ zu erstellen, die anschließend manuell erweitert wurden, werden durch sie beschriebene Modelle fester Bestandteil der Softwareerzeugung. Änderungen erfolgen primär am betreffenden Modell, aus dem anschließend aktuellere Komponenten neu generiert werden. (vgl. [Stahl u. a. 2007, 13]).

Dieses Prinzip bedient sich mehrerer Arten von Transformationen. Erfolgt eine Zustandsänderung des Modells, bspw. die Veränderung von bestehenden Elementen, sprechen Stahl u. a. [2007] von der sog. Modellmodifikation (vgl. [Stahl u. a. 2007, 199f]). Im Gegensatz dazu stehen zwei Modelltransformationen. Wird ein Modell in ein anderes überführt, wobei

² Metaisierung kann unter verschiedenen Gesichtspunkten erfolgen (sprach- prozessbasiert, ontologisch etc.), folgt aber oftmals einer vierschichtigen Hierarchie, wobei jede Ebene eine Instanz der übergeordneten ist. Während ein Modell also ein bestehendes System beschreibt, wird es selbst durch ein übergeordnetes Metamodell beschrieben, das ebenfalls beschrieben werden kann (Meta-Metamodell) (vgl. [Atkinson & Kuhne 2003]).

das Ausgangsmodell nicht verändert wird, liegt eine Modell-zu-Modell-Transformation vor. Dieser Vorgang ist abzugrenzen von der Modell-zu-Text-Transformation. Sie generiert aus dem Modell lauffähigen, plattformspezifischen Quelltext. (vgl. [Stahl u. a. 2007, 33]).

Im Wesentlichen werden beide Paradigmen im Rahmen des Generators dadurch verwoben, dass die Modelle an die Stelle der Problem- und Lösungsräume treten und durch das Konfigurationswissen transformiert werden.

2.2 FAMIX

Wie zuvor erwähnt, extrahiert der Generator die Ausgangsdaten aus formalen Quellen. Sie liegen als FAMIX-Dateien vor, die durch Parser erzeugt werden.

FAMIX ist eine Metamodell-Familie, die im Rahmen des Moose-Projekts³ entstand. Die durch sie erzielte Sprachenunabhängigkeit ermöglicht die Analyse einer umfangreichen Menge an Quellsystemen, bspw. aus C++, C#, Java und Smalltalk (vgl. [Wettel 2010, 102]). Der FAMIX-Parser, durch den ein Softwaresystem analysiert wird, erzeugt eine Datei, die Informationen über alle Systementitäten speichert. Jede Entität erhält einen einzigartigen Bezeichner, und wird mit ihren entsprechenden Eigenschaften in einem Block zusammengefasst, umschlossen von einem Paar Klammern. Entitätseigenschaften können sowohl Referenzen auf andere Entitäten sein, die durch ihren Bezeichner ausgedrückt werden, als auch primitive Typen, wie Zeichenketten (vgl. [Ducasse u. a. 2011]). Auf diese Weise werden Daten über Namensräume, Pakete, Klassen, Methoden, Attribute und Relationen aufbereitet.

2.3 Xtext und Xtend

Während FAMIX für die Bereitstellung der Softwaredaten sorgt, wird für den Generator ein Metamodell benötigt, welches die zu visualisierende Metapher beschreibt. Ohne dieses Metamodell kann keine Transformation erfolgen, in der die FAMIX-Daten auf den ersten Lösungsraum abgebildet werden können.

Zu diesem Zweck wird Xtext verwendet, ein Framework, das sich auf die Erstellung von domänenspezifischen Sprachen (engl. *domain-specific languages*) und Programmiersprachen spezialisiert und Bestandteil der Eclipse Entwicklungsumgebung ist (vgl. [o. V. 2014b, 113]). Die dabei erstellten Komponenten sind unabhängig von Eclipse und können an jede Sprachplattform angepasst werden (vgl. [o. V. 2014b, 113f]). Domänenspezifische Sprachen zeichnen sich dadurch aus, dass sie speziell auf eine einzige Domäne zugeschnitten sind. Auf was für einen Aspekt sich diese Domäne bezieht ist beliebig (vgl. [o. V. 2014b, 114]).

³ <http://www.moosetechnology.org>

Im Falle des Generators wird durch Xtext pro Metapher eine Infrastruktur erzeugt, die aus dem verfassten Metamodell generiert wird.

Ergänzt wird die Verarbeitung durch Xtend, einer Programmiersprache, die nahtlos auf Java aufsetzt, mit dem Ziel den Quelltext lesbarer zu machen und den Programmieraufwand zu minimieren (vgl. [o. V. 2014a, 6]). Dies wird durch syntaktische Anpassungen erzielt. Nähere Informationen bzgl. beider Sprachkonzepte können den Dokumentationen entnommen werden.⁴

2.4 X3D

Um die verarbeiteten Daten visualisieren zu können, was der zweiten Transformation aus dem GSVDM entspricht, bedarf es einer Plattform. Zurzeit verwendet der Visualisierungsgenerator Extensible 3D (X3D), die auf der Extensible Markup Language (XML) basiert. X3D ist eine Beschreibungssprache für 3D-Inhalte, die vom Web3D Konsortium⁵ entwickelt wird und der offizielle Nachfolger der Virtual Reality Modeling Language (VRML) wurde, die immer noch Teil von X3D ist. Neben der Darstellung von 3D-Objekten sind auch Animationen, Navigation und Manipulation durch Scripting möglich (vgl. [o. V. 2015]).

Im Verlauf der Modell-zu-Text-Transformation werden durch den Visualisierungsgenerator die Modellentitäten auf die Grundformen der Zielplattform abgebildet: Quader, Kegel, Kugel und Zylinder. Ihre Maße ergeben sich aus wichtigen Metriken des visualisierten Systems, um so eine 3D-Darstellung entsprechend einer Metapher zum Zweck der Analyse von Software zu erstellen.

⁴ Xtext: <http://www.eclipse.org/Xtext/documentation.html>,
Xtend: <http://www.eclipse.org/xtend/documentation.html>

⁵ <http://www.web3d.org/>

3 Implementierung des Prototyps

Mit dem Hintergrundwissen über den Generator kann nun der Implementierungsprozess beginnen. Nach Analyse der Zielmetapher sollte eine Referenzmetapher gewählt werden an der sich die neu zu implementierende Metapher orientieren wird. Dies hat den Vorteil, dass zum einen Quellcode wiederverwendet werden kann und zum anderen Ähnlichkeiten im Code das Einarbeiten in eine neue Metapher erleichtern. Aus beiden Analysen ergeben sich die Spezifikationen, welche die zu implementierenden Komponenten des Generators zu erfüllen haben. Während die Zielmetapher die funktionalen Aspekte bereitstellt, kann eine Referenzmetapher die Anforderungen um systemspezifische Eigenheiten erweitern, insofern dies sinnvoll und erwünscht ist.

Darauf folgt das Erstellen eines Prototyps. Als Prototyp wird im Kontext dieser Arbeit eine Gruppe an Artefakten verstanden, die – durch einen ausführenden Workflow verbunden – eine Visualisierung der eingelesenen Daten erzeugt. Abhängig von der Komplexität der zu implementierenden Metapher oder der Ungenauigkeit der Spezifikation ist es denkbar, dass mehrere Prototypen entworfen werden müssen, um ein zufriedenstellendes Ergebnis durch Tests zu ermitteln. Zu diesem Zweck werden nur die wichtigsten Anforderungen an die Visualisierung implementiert.

Nach der Auswahl des Prototyps, der alle Anforderungen am besten erfüllt, wird dieser abschließend inkrementell und iterativ um jene Aspekte erweitert, die die Visualisierung vervollständigen. Dabei ist zu beachten, dass die zu ergänzenden Komponenten auf einer strukturell optimalen Basis aufbauen. D.h. dass der Prototyp zuvor ausgiebig getestet und auch hinreichend optimiert sein sollte, bevor die restlichen Ergänzungen hinzugefügt werden. Diese Absicherung vermeidet Fehlerquellen und mehrfache Überarbeitung.

3.1 Analyse der Zielmetapher

Für die Implementierung der Stadtmetapher des Visualisierungsgenerators dient „CodeCity“ als Vorlage. Um diese Vorlage korrekt bewerten zu können und die wichtigen Anforderungen, welche die zu implementierende Metapher erfüllen muss, abgrenzen zu können, sollte man sich zuvor mit ihr vertraut machen.

3.1.1 Grundlagen von „CodeCity“

Das Visualisierungswerkzeug ist unter anderem Gegenstand von Wettels Dissertation und spezialisiert sich darauf, Software unter unterschiedlichen Gesichtspunkten zu analysieren (vgl. [Wettel 2010]). Dabei stützt es sich auf verschiedene Konzepte. Im Zentrum steht die Stadtmetapher.

Als dreidimensionale Echtweltmetapher, die zu visualisierende Entitäten auf Gegenstände abbildet, die dem Menschen aus seinem realen Umfeld bekannt sind, bietet sie dem Nutzer eine natürlich gegebene Orientierung, aber auch die damit einhergehenden Einschränkungen. Da das Verständnis des Menschen bzgl. der Metapher durch eigene Erfahrungen geprägt ist, ist die Zuordnung von bspw. abstrakten Elementen erschwert (vgl. [Maletic u. a. 2001]). Meist wird das Problem nur dadurch lösbar, indem dem Abbild Objekte hinzugefügt werden, die im realen Umfeld nicht existieren, wodurch die Metapher durchbrochen wird (vgl. [Wettel 2010, 38ff]). Entitäten der Metapher verkörpern Softwareelemente, die durch ihre Charakteristiken Wiedererkennungswert haben und als Landmarken dienen. Hierbei stellen Distrikte Pakete dar. Die in ihnen liegenden Klassen werden als die zugehörigen Gebäude visualisiert, deren Maße auf der Anzahl ihrer Attribute und Methoden basieren und ihre Farbe durch die Anzahl der Quelltextzeilen (engl. *lines of code*, LOC) festgelegt wird (vgl. [Wettel 2010, 30]).

Ein weiterführendes Konzept zur Analyse sind „Age Maps & Time Travel“ (vgl. [Wettel 2010, 55ff]). Anhand der Einfärbung, entsprechend der Lebenszeit des Softwareelements, wird eine Entwicklungsanalyse möglich. Allerdings wird diese Darstellungsoption im Rahmen dieser Arbeit nicht implementiert, da Versionsanalysen in einem anderen Teilprojekt umgesetzt werden. Dasselbe gilt für die „Disharmony Maps“, die Darstellung von Relationen und zusätzliche feingranulare Ansichten, um Methoden und Attribute zu visualisieren. Um dieses System sind die Funktionalitäten des Werkzeugs konzipiert. Zu ihnen gehört bspw. eine Benutzungsschnittstelle, die dem Nutzer während der Betrachtung der Visualisierung zusätzliche Informationen zu einzelnen Entitäten bereitstellt und vielfältige Navigation erlaubt (vgl. [Wettel 2010, 100]). Des Weiteren gibt es eine „View Configuration“, die vordefinierte Konfigurationen anbietet, dem Nutzer die Möglichkeit bietet diese anzupassen oder durch ein Scripting-Modul eigene Anordnungen für die Entitäten zu entwerfen (vgl. [Wettel 2010, 96ff]). Das Werkzeug übernimmt den gesamten Prozess der Softwarevisualisierung. Angefangen bei einem FAMIX-Modell, das wie beim Visualisierungsgenerator eingelesen wird, bis hin zur Darstellung.

3.1.2 Anforderungen

Mit diesem Wissen über „CodeCity“ werden nun die Anforderungen an die Zielmetapher definiert. Da lediglich die Stadtmetapher umgesetzt werden soll, entfallen alle Funktionalitäten zur Interaktion und Konfiguration. Im Rahmen dieser Arbeit wird die Visualisierung „Magnitude“ als Standard festgelegt (vgl. Abbildung 3.1), die alle zu implementierenden Anforderungen enthält.

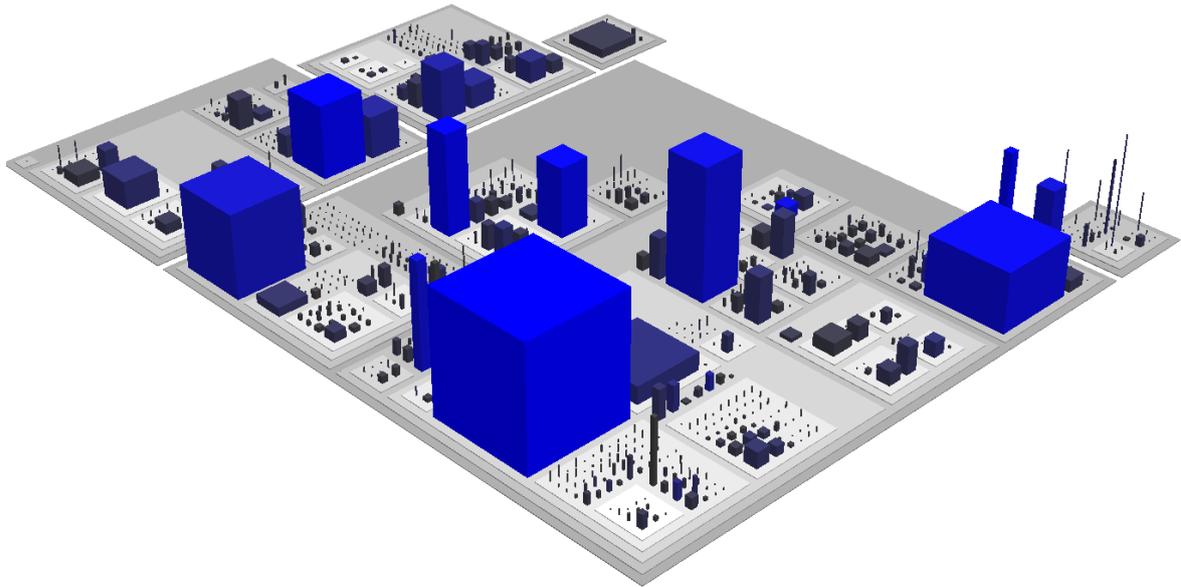


Abbildung 3.1: „Freemind“ visualisiert durch „CodeCity“, Konfiguration „Magnitude“

Durch die „View Configuration“ können die wichtigsten Metriken direkt aus dem Werkzeug entnommen werden. Wie Abbildung 3.2 zeigt, können links alle darstellbaren Elemente angewählt werden. Das Kontrollkästchen ermöglicht es, diese bei Bedarf in der Visualisierung nicht darstellen zu lassen. In einem Dropdown-Menü wird aufgelistet, welche Form das ausgewählte Element annehmen soll. Darunter kann man spezifische Eigenschaften anpassen. Im Allgemeinen umfasst dies die Farbe, die Abmessungen und die Transparenz (alpha). Die Rubrik „Inner Layout“ modifiziert die Anordnung der Kindelemente innerhalb des aktuell angewählten Elements.

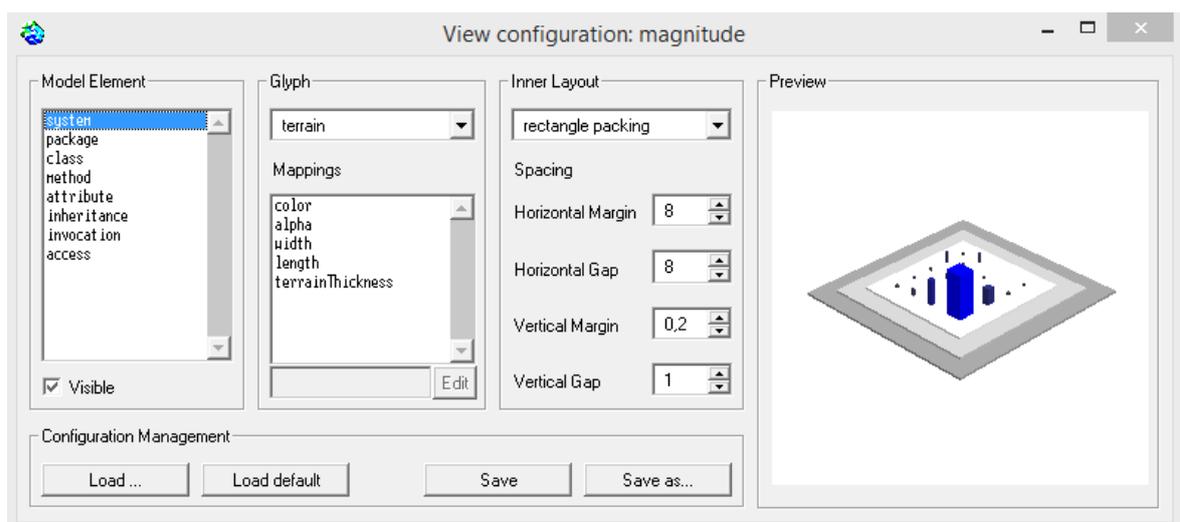


Abbildung 3.2: Das „View Configuration“-Werkzeug

Die nachfolgende Tabelle 3.1 listet alle Attributwerte für die 3D-Elemente auf. Da nur System, Paket und Klasse in der Standardvisualisierung genutzt werden, werden die übrigen Elemente in der Übersicht vernachlässigt.

Element	Figur	Farbe	Alpha	Breite	Länge	Dicke	Höhe
System	Terrain	rgb(128,128,128)	0	1	1	1	---
Paket	Terrain	LIN[Hierarchie]	1	15	15	1	---
Klasse	Quader	LIN[LOC]	1	ID[NOA]	ID[NOA]	---	ID[NOM]

Tabelle 3.1: Standardwerte der Elementattribute in „Magnitude“

System bezeichnet den „Container“ in dem die 3D-Abbildung erzeugt wird. Durch einen Alpha-Wert von null wird dieses Element vollständig durchsichtig. Das Terrain kann als Platte oder Ebene bezeichnet werden, auf der die Gebäude stehen (vgl. Abbildung 3.2, Vorschaubild rechts). Trotz der expliziten Kennzeichnung als Terrain ist das 3D-Objekt mit einem flachen Quader gleichzusetzen. Da X3D nur vier primitive Formen kennt, müssen sowohl Klassen als auch Pakete und das System auf einen Quader abgebildet werden.

Außerdem nutzt dieses Element ausschließlich „Constant Mapper“. D.h. dass die angegebenen, unveränderlichen Werte direkt in die Visualisierung eingehen. Andere 3D-Elemente nutzen unter Umständen komplexere Funktionen um Werte abzubilden.

Beim „Identity Mapping“ (ID) gehen Metriken der Entität unverändert in die Darstellung ein und ist die genaueste Art und Weise um Softwaremetriken darzustellen. Das 3D-Objekt der Klasse macht Gebrauch davon, indem es die Anzahl der Attribute (engl. *number of attributes*, NOA) auf die Länge und Breite abbildet. Die Höhe ergibt sich aus der Anzahl der Methoden (engl. *number of methods*, NOM).

„Linear Mapping“ (LIN) wird hauptsächlich für Farbspektren genutzt. Der Mapper erhält einen Start und einen Endwert und unterteilt die Definitionsmenge in eine festgelegte Anzahl von Bereichen, um diese auf entsprechend abgestufte Farben abzubilden. „Magnitude“ nutzt dieses Verfahren, um Softwarepakete anhand ihrer Tiefe in der Hierarchie einzufärben. Je tiefer das Paket liegt, umso heller ist die Färbung. Ebenso wird die lineare Zuordnung für die Einfärbung der 3D-Objekte der Klassen genutzt, abhängig von der Anzahl der Quelltextzeilen, die der Klasse zuzuordnen sind⁶.

Da jetzt das äußere Erscheinungsbild den entsprechenden Softwareelementen zugeordnet werden kann, wird nachfolgend untersucht, welche Regeln für die Anordnung der Kindelemente gelten. Tabelle 3.2 gibt einen Überblick über die Daten. Anhand der vorangegangenen Anforderungsermittlung ist von der zu implementierenden Metapher bekannt, dass die Gebäude keine geschachtelten Objekte darstellen werden, was eine Betrachtung des Klassenelements unnötig macht.

⁶ Weitere Mapping-Methoden wären das „Box plot based Mapping“ und das „Threshold based Mapping“, die Wettel im Zusammenhang mit „CodeCity“ nutzt (vgl. [Wettel 2010, 32f]). Diese sind für die Standardvisualisierung nicht relevant.

Element	Anordnung	horizontaler Rand	horizontaler Abstand	vertikaler Rand	vertikaler Abstand
System	rectangle packing	8 Einheiten	8 Einheiten	0,2 Einheiten	1 Einheiten
Paket	rectangle packing	8 Einheiten	8 Einheiten	0,2 Einheiten	1 Einheiten

Tabelle 3.2: Standardwerte der Anordnung von Kindelementen in „Magnitude“

Die Attributnamen sind frei übersetzt und beschreiben folgende Eigenschaften:

- horizontaler/vertikaler Rand: horizontale/vertikale Distanz zwischen dem Element und seinem Elternelement
- horizontaler/vertikaler Abstand: horizontale/vertikale Distanz zwischen dem Element und seinen Geschwisterelementen

Die vertikalen Distanzen können in der Zielmetapher vernachlässigt werden, da das Werkzeug auch Anordnungen erlaubt, welche die 3D-Objekte übereinander auf der Y-Achse arrangieren. Dies ist in der Standardvisualisierung nicht vorgesehen. Auch wenn „Magnitude“ gleichgroße Freiräume zu den Eltern- und Geschwisterelementen vorsieht, sollte hier beachtet werden, dass die „View Configuration“ ein Werkzeug zur Anpassung ist. Die Entfernungen sollten schlussendlich parametrisierbar sein.

Für die Darstellung hierarchischer Datenstrukturen bietet sich im Allgemeinen eine „Treemap“ nach Shneiderman an (vgl. [Shneiderman 1992]). Dabei wird eine gegebene Fläche anhand der gewichteten Anteile der Kindelemente rekursiv aufgeteilt. Dieses Prinzip ist auf die Stadtmetapher nicht anwendbar, da die Kindelemente Gebäude mit konstanter Grundfläche sind. Aus diesem Grund nutzt „CodeCity“ einen Algorithmus, der dem „bin-packing“-Problem ähnelt (vgl. [Wettel 2010, 34]) und in Tabelle 3.2 als „rectangle packing“ bezeichnet wird. Die Anordnung erfolgt nach dem Vorbild eines Lightmap-Algorithmus⁷, basierend auf einem K-dimensionalen-(KD-)Baum (vgl. [Wettel 2010, 34]).

Lightmaps dienen im Allgemeinen der Speicherung von Beleuchtungsinformationen von 3D-Objekten. KD-Bäume sind unbalancierte Binärbäume, die zum Aufteilen eines k-dimensionalen Raums genutzt werden, um Punkte aus dem R^k zu speichern. Der vorhandene Raum wird dabei nacheinander in den verschiedenen Ebenen geschnitten, um letztendlich eine Aufteilung zu erhalten, in der jeder Punkt aus der gesamten Punktmenge in einem eigenen Sektor liegt und somit klar identifizierbar ist.

Die Anordnung der Elemente in „CodeCity“ erfolgt anhand der Grundflächen der Gebäude. Sie werden der Größe nach geordnet und dann der Reihe nach im rechteckigen Distrikt mit den entsprechenden Abständen zueinander angeordnet. So erhält die Visualisierung einen Schwerpunkt im Koordinatenursprung. Bei jedem Hinzufügen eines Elements wird geprüft,

⁷ <http://www.blackpawn.com/texts/lightmaps>

ob der Distrikt ein möglichst quadratisches Seitenverhältnis beibehält. Der Algorithmus baut das Layout von innen nach außen auf, da zuerst die innersten Distrikte ausgerichtet werden müssen, um ihre Größe zu bestimmen. Erst wenn ihre Größe bekannt ist, können sie zusammen mit ihren Geschwisterelementen angeordnet werden. Wettel legt den für „CodeCity“ verwendeten Algorithmus als Pseudocode in seiner Arbeit dar (vgl. [Wettel 2010, 36]). Dieser ist auf die vorzunehmende Implementierung übertragbar.

3.1.3 Analyseergebnisse

Die Ergebnisse der Analyse lassen sich gut in einer per Hand erstellten Beispielsvisualisierung⁸ festhalten. Abbildung D.1 visualisiert exemplarische Gebäude. Die Anordnung erfolgt nach dem Vorbild des Algorithmus, indem die Elemente mit der größten Grundfläche als erstes im Distrikt ausgerichtet werden. Die Abstände der Elemente zueinander – ablesbar aus der Vogelperspektive – entsprechen den festgehaltenen Werten aus der „View Configuration“ (vgl. Tabelle 3.2). Das System-Element, das normalerweise transparent ist, wurde zur Veranschaulichung auf 80% Transparenz gesetzt. Dieser Entwurf dient der Veranschaulichung der zusammengetragenen fachlichen Daten und wird später als Vorlage für die Modell-zu-Text-Transformation genutzt.

3.2 Auswahl und Analyse der Referenzmetapher

Um die Anforderungen an die Zielmetapher zu erweitern, bietet es sich an, eine Referenzmetapher auszuwählen, die bereits im Visualisierungsgenerator vorhanden ist. Auf diese Art sollen systemspezifische Eigenheiten ermittelt werden und erste Einblicke in die Verarbeitung der Daten gewonnen werden. Der Generator verfügt zurzeit über drei Metaphern.

Abbildung 3.3 zeigt die Schachtelungs- (vgl. [Rekimoto & Green 1993]), Recursive Disk- (vgl. [Müller & Zeckzer 2015]) und Weltraummetapher (vgl. [Graham u. a. 2004]) im Vergleich. Die vorliegende, leicht abgewandelte Schachtelungsmetapher stellt Programmpakete durch Würfel und Klassen anhand von Kugeln dar. Die zugehörigen Elemente werden innerhalb der Formen angeordnet und erfordern, dass die Objekte transparent sein müssen, damit alle Bestandteile für den Benutzer sichtbar bleiben. Die Recursive Disk-Metapher stellt Softwareelemente durch flache Scheiben dar. Pakete sind (graue) Ringe, eine Klasse jeweils eine (violette) Scheibe und zugehörige Attribute (gelb) und Methoden (hellblau) sind deren Ringsegmente. Die Weltraummetapher stellt eine Variation der Sonnensystemmetapher von Graham u. a. [2004] dar. Sie ordnet die zugehörigen Kindelemente im Kreis um das Elternelement an. Auch hier sind Pakete als Würfel dargestellt und Klassen als Kugeln.

⁸ auf CD enthalten: „Beispielsvisualisierung.x3d“

Diese zirkulare Anordnung in Verbindung mit der Neigung der Ringe ist für die Stadtmetapher nicht von Nutzen, da die Gebäude sich auf einer waagrechten Ebene befinden sollen.

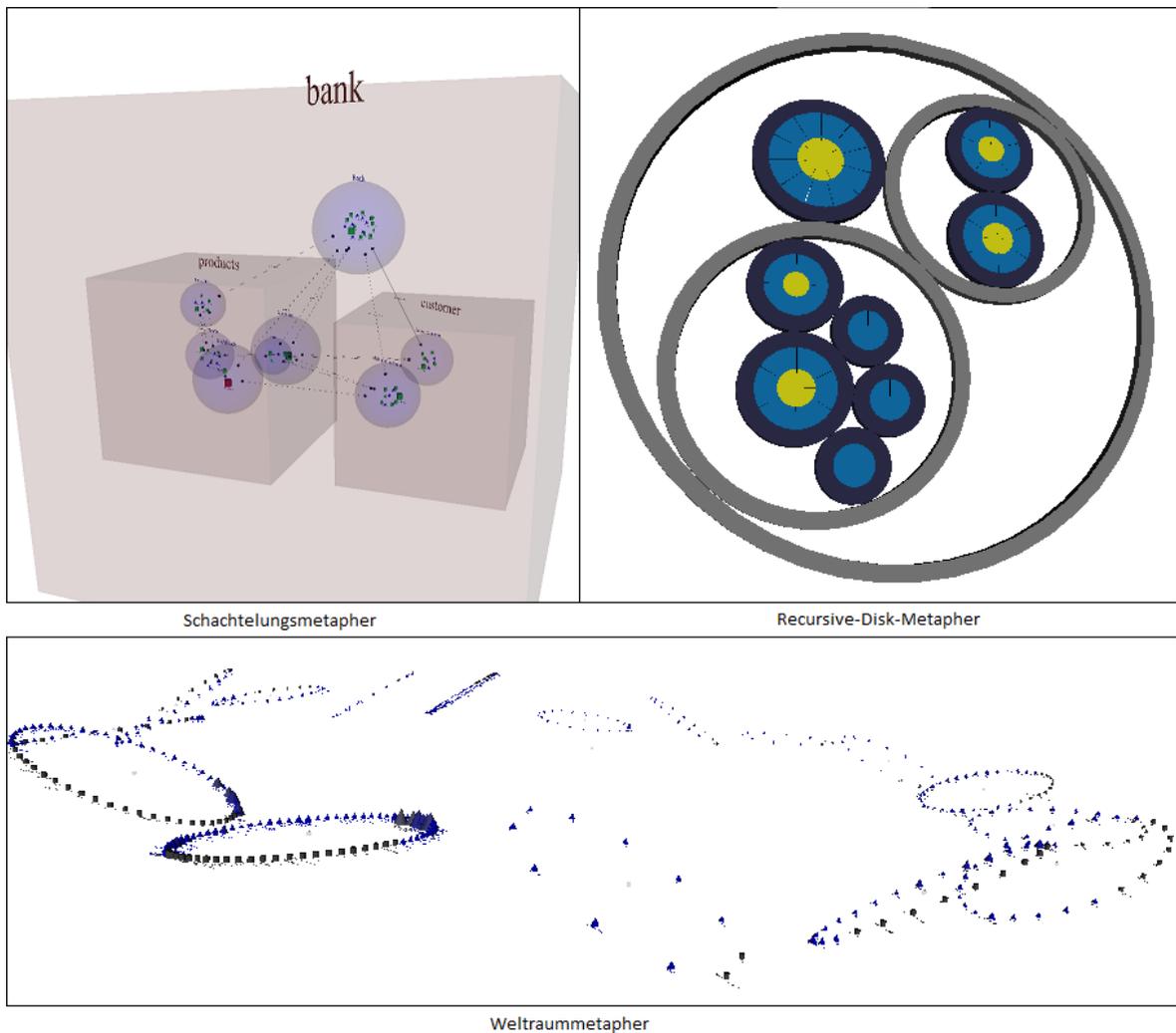


Abbildung 3.3: Metaphern des Visualisierungsgenerators

Aufgrund der verwendeten Formen und der Tatsache, dass sie die drei Dimensionen gut ausnutzt, macht die Schachtelungsmetapher den Eindruck als Referenzmetapher geeignet zu sein. Allerdings beruht das Konzept auf der Schachtelung von Elementen, die in der Zielmetapher laut Anforderungsermittlung (s. Abschnitt 3.1.2) nicht vorgesehen ist.

Die Recursive Disk-Metapher wirkt im Vergleich eher zweidimensional, teilt sich aber wichtige Merkmale mit der Zielmetapher. Die Anordnung der Elemente ist darauf ausgelegt, möglichst lagestabil zu sein und macht eine Überlappung auf der Ebene unzulässig (vgl. [Müller & Zeckzer 2015]). Die Scheiben sind in diesem Fall auf einem einzelnen Niveau angeordnet, während die Stadtmetapher Gebrauch von mehreren Plateaus macht. Essentiell ist aber die Anordnung der Gebäudegrundflächen, die in den Distrikten platziert werden müssen. Alle Klassenquader stehen mit ihrem Fundament auf derselben Plattform wie ihre

Geschwisterelemente. Aufgrund dessen stellt die Recursive Disk-Metapher die beste Referenz zur Implementierung der Stadtmetapher dar.

3.2.1 Grundlagen der Referenzmetapher

Die Analyse der Referenzmetapher beginnt ebenfalls mit einer Übersicht über ihre grundlegenden Eigenschaften. Die Metapher ist von der Forschungsgruppe erarbeitet worden und stellt Klassen und Pakete als ineinander geschachtelte Scheiben dar.

In ihrem Zentrum werden jene Typen in einem Kreis dargestellt, die in der betreffenden Klasse definiert wurden, und werden selbst auch durch Scheiben verkörpert, auf die der nachfolgend beschriebene Aufbau ebenfalls zutrifft. Sollten verschachtelte Elemente nicht vorhanden sein, entfallen sie einfach. Die Attribute, die in der Klasse enthalten sind, werden als Ringsegmente veranschaulicht, die sich um den innenliegenden Kreis mit definierten Typen legen. Für den Fall, dass keine Typen in der Klasse enthalten sind, werden die Attribute als Kreissektoren angeordnet. Ebenso verhält es sich mit den Klassenmethoden.

Wenn die Klasse über Attribute verfügt, werden die Methoden als Ringsegmente um die Attribute gelegt. Sind keine Attribute definiert, aber Typen innerhalb der betrachteten Klasse vorhanden, ordnen sich die Methoden als Ringsegmente um den innenliegenden Kreis an. Sind außer Methoden keine weiteren Klassenelemente existent, werden die Methoden der Klasse als Kreissektoren dargestellt. Die Klassen sind von Ringen umschlossen und liegen innerhalb von Paketen, die ebenfalls durch Ringe verkörpert werden.

Die Anordnung der Pakete und Klassen erfolgt anhand ihrer Fläche, die sich aus den Flächen ihrer Unterelemente ergibt. Die Größe der Attribute errechnet sich hierbei aus ihrer Anzahl, und die Fläche der Methoden aus der Anzahl der in ihnen enthaltenen Anweisungen. Jene Klasse mit der größten Nettofläche⁹ rückt in das Zentrum der Visualisierung. Von ihr ausgehend werden die nächst kleineren Klassen und Pakete derselben Ebene im Uhrzeigersinn in Form einer Spirale positioniert. Da sich die Methodengröße auf die Anzahl der Statements stützt und die Größe der Attribute auf ihre Anzahl, bilden diese beiden Klassenbestandteile den elementaren Anteil an den Klassengrößen. Da die in der Klasse enthaltenen Typen dem gleichen rekursiven Aufbau folgen, bis sich in ihnen nur noch Typen befinden, die keine weiteren Typen in sich definieren, beziehen sie ihre eigene Größe nur noch durch Attribute und Methoden. Auf diese Weise lässt sich feststellen, dass die Darstellung jene Klassen in den Vordergrund rückt, die den größten Anteil am Programm haben (vgl. [Müller & Zeckzer 2015]).

⁹ Da sich Ringe als Begrenzungen um die Elemente spannen ergeben sich unter Umständen große Leerräume zwischen den einzelnen Elementen. Die Bruttogröße eines Elements bezieht diesen Zwischenraum bei der Messung mit ein. Wird dieser Raum von der Größe abgezogen erhält man die Nettofläche.

3.2.2 Erweiterung der Anforderungen

Der Visualisierungsgenerator benötigt für eine Modell-zu-Modell-Transformation neben dem FAMIX-Metamodell das Metamodell der zu visualisierenden Metapher. Das Metamodell der Recursive Disk-Metapher wird nachfolgend analysiert, um ein entsprechendes Konzept für die Zielmetapher zu erarbeiten.

Der Generator erzeugt unter anderem ein Ecore-Modell, das mithilfe entsprechender Werkzeuge¹⁰ für Klassendiagramme genutzt werden kann. Das Diagramm (vgl. Abbildung A.1) zeigt die kompositionellen Beziehungen zwischen den generierten Klassen der Metapher.

Es ist erkennbar, dass als Basis ein Wurzelement namens `Root` genutzt wird. Die Klasse enthält das `Document`, das wiederum über `Disk` verfügt. `Disk` ist die zentrale Klasse des Modells. Sie enthält wichtige, identifizierende Eigenschaften, die aus den FAMIX-Daten entnommen werden können. Methoden und Attribute, die zur jeweiligen Scheibe gehören – insofern vorhanden – sind Instanzen von `DiskSegment` und werden über die Felder `methods` und `data` in Beziehung zum Elternelement gesetzt. Zudem kann man erkennen, dass die Klasse `Disk` sich zweimal selbst referenziert, als Felder `parent` und `disks`. Letzteres bestätigt die Schachtelung der Scheiben (vgl. Abbildung 3.3).

Die Klasse `Reference` dient der Darstellung der Vererbungsbeziehungen. Da diese nicht Gegenstand der Anforderung ist, ist eine Berücksichtigung untergeordnet, sollte aber im Metamodell vorhanden sein. Deshalb wird diese Klasse ohne Modifikation von der Recursive Disk-Metapher übernommen. Dasselbe gilt für die Klasse `Position`. Sie dient der Positionierung des Elements und der späteren Beschriftungen, weshalb sie zusammen mit den entsprechenden Feldern im Metamodell enthalten bleiben sollte.

`Document` fungiert als Container für die darzustellenden Elemente, und kann mit dem Modellelement `System` aus „CodeCity“ gleichgesetzt werden. Das Wurzelement `Root` ist `Document` übergeordnet. Dieser Aufbau wird auf das Metamodell der Zielmetapher übertragen. Dies begünstigt die Standardisierung der Dateien und wird die Verarbeitung durch den Generator erleichtern.

Die Stadtmetapher stützt sich auf die Abbildung von zwei Hauptelementen. Zum einen den `District` und zum anderen das `Building`. Wie die Analyse der Anforderungen (s. Abschnitt 3.1.2) gezeigt hat, werden beide in X3D durch dieselbe geometrische Form repräsentiert: einen Quader. Aufgrund dieser starken Ähnlichkeit wäre es vorstellbar nur eine Klasse

¹⁰ siehe bspw. <http://www.eclipse.org/ecoretools/doc/index.html>

für beide Elemente zu implementieren. Aus Gründen des Verbalisierungsprinzips (vgl. [Balzert 2009, 46ff]) wurde diese Option nicht umgesetzt. Stattdessen werden `District` und `Building` von einer Superklasse namens `Entity` erben, die Gemeinsamkeiten beider Subklassen kapselt. An dieser Stelle kann `Disk` als Vorlage dienen. Aus ihr werden alle Attribute übernommen, die auch Elemente der Stadtmetapher nutzen können, z.B. vollqualifizierte Namen, Elementtypen, Farbe etc.

Da zur Ermittlung der Methoden- und Attributanzahl für Klassen auf jene Elemente zugegriffen werden muss, bietet es sich an, Informationen über Attribute und Methoden ebenfalls zu speichern. „CodeCity“ zeigt auf, wie die Stadtmetapher erweitert werden kann, um aus diesen Informationen Nutzen zu ziehen (vgl. [Wettel 2010, 38ff]). Sollte zu einem späteren Zeitpunkt entschieden werden, diese oder ähnliche Optionen umzusetzen, entfällt eine Nachbearbeitung des Quelltextes. Außerdem bietet das Mitführen von Attributen und Methoden eine zusätzliche Möglichkeit zur Kontrolle. Hierfür wird die Klasse `DiskSegment` der Recursive Disk-Metapher als Vorbild genommen, da diese ebenfalls die Aufgabe hat Unterelementen von Klassen eine Form zugeben. Die zusätzliche Klasse `EntitySegment` erhält jedoch im Falle der Stadtmetapher keine Visualisierung. Ähnlich wie `Disk` werden auch für `EntitySegment` alle Eigenschaften übernommen, von denen sie sinnvoll Gebrauch machen kann.

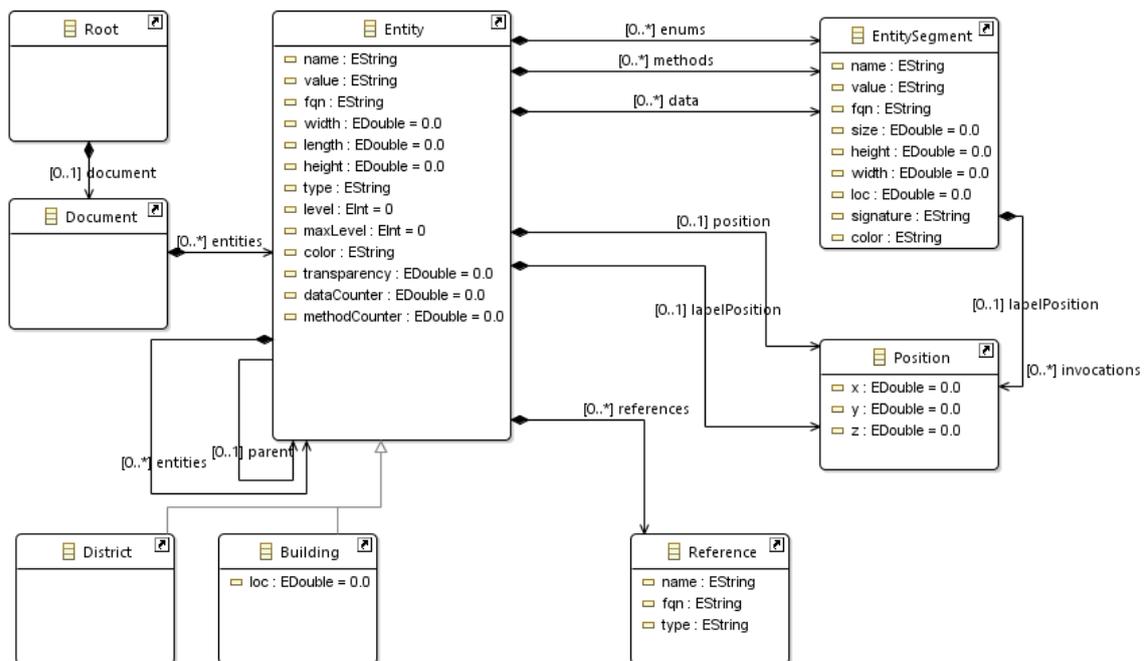


Abbildung 3.4: Klassendiagramm der Stadtmetapher

3.3 Das Metamodell

Durch die Analyse der Referenzmetapher wird ersichtlich, wie die Konzeption des Metamodells erfolgen sollte. Abbildung 3.4 zeigt das zu erzeugende Metamodell. Dieses Modell muss nun in der entsprechenden Datei¹¹ per Xtext umgesetzt werden.

Die Klassen `Reference` und `Position` werden direkt in das neue Metamodell übernommen. Deshalb wird ihre Implementierung in diesem Abschnitt nicht weiter betrachtet, stattdessen liegt das Augenmerk auf den Hauptentitäten `Entity`, `District` und `Building`. Bei einer Vererbungsbeziehung in Xtext werden die gemeinsamen Eigenschaften der Subklassen bei der Generierung automatisch der Superklasse zugeordnet. Listing 3.1 zeigt wie diese Beziehung deklariert wird.

```
1. Entity:
2.   District | Building
3. ;
```

Listing 3.1: Vererbungsbeziehung - Entity wird zur Superklasse

```
1. District:                                // no LOC
2.   '(District'
3.   (('id: ' name=INT_ID '))
4.   (('name' value=MSESTRING '))
5.   (('fqn' fqn=MSESTRING '))
6.   (('type' type=MSESTRING '))           //only „package“ & namespace
7.   (('width' width=DECIMAL '))          //base area (width x length)
8.   (('length' length=DECIMAL '))
9.   (('height' height=DECIMAL '))
10.  (('level' level=INT '))
11.  (('maxLevel' maxLevel=INT '))'?
12.
13.  (('color' color=MSESTRING '))'?
14.  (('transparency' transparency=DECIMAL '))'?
15.  (('position' position=Position '))'?
16.  (('labelPosition' labelPosition=Position '))'?
17.
18.  (('entities' entities+=Entity* '))'?
19. //for CITY: no use yet. but keep it for later additions
20.  (('data' data+=EntitySegment* '))'?
21.  (('dataCounter' dataCounter=DECIMAL '))
22.  (('methods' methods+=EntitySegment* '))'?
23.  (('methodCounter' methodCounter=DECIMAL '))
24.  (('enums' enums+=EntitySegment* '))'?
25.
26.  (('references' references+=Reference* '))'?
27.  (('parent' parent=[Entity] '))'?
28.  ')'
29. ;
```

Listing 3.2: Implementierung von District

Die Klassen `District` und `Building` werden nun mit den erforderlichen Feldern implementiert. `Building` erhält dieselben Informationen, wie in Listing 3.2 aufgezeigt, mit der

¹¹ org.svis.xtext.city/src/org/svis.xtext/City.xtext

Ausnahme, dass den Gebäuden ein zusätzliches Feld `loc` hinzugefügt wird, das die Anzahl der Quelltextzeilen speichert.

Zeilen 3 bis 11 zeigen die wichtigsten Daten, die aus einer FAMIX-Datei bzgl. eines Softwareelements entnommen werden können. Auch wenn Gebäude eine quadratische Grundfläche haben, die mit nur einer Längenangabe definiert werden kann, werden die Felder `width`, `length` und `height` in die Superklasse ausgelagert. Somit ist bei der Abarbeitung des später zu implementierenden Algorithmus keine weitere Fallunterscheidung zu beachten.

In den Zeilen 13 bis 16 ist neben den Informationen über die Farbe und Transparenz das Feld `position` enthalten. Es speichert den Punkt, an dem das zugehörige 3D-Objekt ausgerichtet wird. Es ist wichtig zu wissen, dass ein Quader in X3D am Mittelpunkt ausgerichtet wird, um dies in späteren Berechnungen zu berücksichtigen. Die Klasse `Position` kapselt die x-, y- und z-Koordinaten im dreidimensionalen Raum.

Zeilen 18 bis 24 befassen sich mit den möglichen geschachtelten Elementen. Um nach dem Auslesen der FAMIX-Daten nicht noch einmal über die Methoden und Attribute iterieren zu müssen, werden Zähler namens `dataCounter` und `methodCounter` implementiert, damit ihre Anzahl festgehalten wird. `References` umfasst die Relationen des aktuellen Elements und `parent` stellt eine Referenz zum übergeordneten Element dar.

Die Implementierung von `EntitySegment` wird an dieser Stelle nicht explizit erklärt, da sie keinen primären Nutzen für die Zielmetapher bringt. Die Klasse ist vielmehr ein Platzhalter für spätere Ausarbeitungen, ohne dass sie momentan ein Layout zugewiesen bekommt.

Schlussendlich muss noch das Wurzelement und das Dokument, in denen die Entitäten verpackt werden, um im Generator weitergeleitet werden zu können, definiert werden.

```
1. Root:
2.  document=Document?
3. ;
4.
5. Document:{Document} // 'Document' equals 'System' from CodeCity
6. '(' entities+=Entity* ')'
7. ;
```

Listing 3.3: Implementierung der „Container“-Elemente

`Root` besitzt bei der Instanziierung kein `Document`, weshalb für jedes `Root`-Element ein neues Dokument über die `Factory` erzeugt werden muss. `Document` selbst kann eine Liste an Entitäten enthalten. An dieser Stelle erleichtert die implementierte Vererbungsbeziehung zwischen `Entity`, `District` und `Building` die Prüfung auf Ausnahmefälle. Sollte ein Softwaresystem tatsächlich über eine Klasse verfügen, die in keinem Paket liegt, kann auch diese dargestellt werden (vgl. Abbildung D.1), da ein Gebäude an die Stelle einer Entität

treten kann. Andernfalls könnte diese Information durch die Visualisierung unterschlagen werden. Nach jeder Änderung des Metamodells müssen die veränderten Komponenten neu generiert werden¹². Die erstellten Klassen sind im entsprechenden Verzeichnis einsehbar¹³.

3.4 Der Workflow

Die Datei des Workflows wird in einem eigenen Verzeichnis für die Stadtmetapher im Test-Ordner des Generators abgelegt¹⁴. Damit die Module `org.svis.generator` und `org.svis.generator.test` auf die Daten der Stadtmetapher zugreifen können, müssen sie den Projekten als Abhängigkeiten hinzugefügt werden. Eine Anleitung dazu bietet Abbildung B.2.

Zu Beginn werden beim Workflow benötigte `EPackages` registriert. Die nachfolgenden Listings zeigen beispielhaft Transformationsschritte und Zwischenergebnisse. Bisher nicht implementierte Module werden zu Beginn auskommentiert und sukzessive hinzugefügt.

```
1. //...
2. Workflow {
3. //...
4.
5. //transformation 1: from FAMIX to CITY
6. component = org.svis.generator.city.s2m.Famix2City {
7.   modelSlot = "FAMIX"
8. }
9.
10. //output 1a: FAMIXstats
11. component = org.eclipse.xtext.generator.GeneratorComponent {
12.   register = org.svis.generator.FamixStatsStandaloneSetup {}
13.   slot = 'FAMIXstats'
14.   outlet = {
15.     path = "output/city"
16.   }
17. }
18.
19. //output 1b: model as *.xml
20. component = org.eclipse.emf.mwe.utils.Writer {
21.   modelSlot = "CITYwriter"
22.   uri =
23.     "platform:/resource/org.svis.generator.test/tmp/city/city1.xml"
24. }
25. //...
26. }
```

Listing 3.4: Workflow-Komponenten zur Modell-zu-Modell-Transformation

Die erste Transformation erfolgt vom FAMIX-Modell zum CITY-Modell. Die hinzugefügte Komponente ist die zu erstellende Xtend-Klasse `Famix2City` (Zeile 6), die in Abschnitt

¹² `org.svis.xtext.city/src/org/svis/xtext/GenerateCity.mwe2` als Workflow ausführen.

¹³ `org.svis.xtext.city/src-gen/org/svis/xtext/city/`

¹⁴ `org.svis.generator.test/src/org/svis/generator/test/city/Famix2City.mwe2`

3.5 eingehender betrachtet wird. Ihr wird der Slot des Input-Readers als Modell-Slot hinzugefügt, damit die Klasse auf die eingelesenen Daten zugreifen kann. Als Ergebnis muss neben dem Stadtmodell die FAMIX-Statistik entstehen (Output 1a). Sie generiert aus dem Slot `FAMIXstats` (Zeile 13) eine JSON-Datei, die im Zusammenhang mit der Optimierung in X3DOM benötigt wird.

Das Ergebnis der Transformation wird im Writer als XML-Datei ausgegeben (Zeile 21) und zur manuellen Überprüfung genutzt. Beide Slots für die Ausgabe der Ergebnisse müssen am Ende der Modell-zu-Modell-Transformation mit den passenden Datenstrukturen belegt werden.

Die Modellmodifikation erfolgt im zweiten Schritt über die Klasse `City2City`, die Gegenstand des Abschnitts 3.6 ist, zusammen mit der Implementierung des Layout-Algorithmus. Der Aufbau der Writer-Komponente für die Ergebnisse der Modifikation ist identisch mit dem vorangegangenen. Mit der Ausnahme, dass die Eigenschaft `CloneSlotContents` auf `true` gesetzt wird, damit der Generator keinen Fehler aufgrund eines bereits existierenden Modells wirft. Nachdem das Stadtmodell sein Layout erhalten hat, werden die Daten im Slot `CITYv2` zugänglich gemacht, um letztlich eine Modell-zu-Text-Transformation durchführen zu können, wie das nachfolgende Listing 3.5 zeigt.

```
1. //...
2. Workflow {
3. //...
4.
5. //transformation 3: from City to X3D
6. component = org.eclipse.xtext.generator.GeneratorComponent {
7.   register = org.svis.generator.city.m2t.City2X3DStandaloneSetup {}
8.   slot = 'CITYv2'
9.   outlet = {
10.     path = "output/city"
11.   }
12. }
13.
14. //...
15. }
```

Listing 3.5: Workflow-Komponenten zur Modell-zu-Text-Transformation

Im Gegensatz zu den vorangegangenen Transformationen benötigt diese eine `GeneratorComponent` (Zeile 6), die ein Setup registriert. Dieses Setup wird anhand der Vorlage der Referenzmetapher angefertigt und gibt die zu erstellende Klasse `City2X3D` zurück. Das Endergebnis der Transformationen wird dann im Ordner `output` ausgegeben (Zeile 10).

3.5 Modell-zu-Modell-Transformation

Nach der Implementierung des Workflows kann die Umsetzung der ersten Transformation erfolgen. Die Klasse `Famix2City`¹⁵ hat die Aufgabe, die FAMIX-Daten auszulesen und anhand des Metamodells entsprechenden Elementen zuzuordnen. Ihr Aufbau ähnelt dem der Referenzmetapher, da sich bei der Zuweisung nur die Zielelemente unterscheiden.

Da `Famix2City` eine Subklasse von `WorkflowComponentWithSlot` ist, muss die Methode `invokeInternal` überschrieben werden. Sie initialisiert das Log, liest die FAMIX-Daten aus und setzt die Transformation in Gang. Nach der Abarbeitung müssen die Slots mit den entsprechenden Modellen besetzt werden, damit die nächste Komponente im Workflow Zugriff erhält.

Der Kern der Implementierung ist die rekursive Suche und Zuweisung von FAMIX-Elementen, deren Daten auf Elemente der Zielmetapher abgebildet werden. Listing 3.6 zeigt die initialisierende Suchoperation, welche die Elementmenge des eingelesenen Dokuments durch Filter eingrenzt.

```
1. famixDocument.elements
   .filter(typeof (FAMIXNamespace))
   .filter[!value.startsWith("java") && !value.startsWith("com") &&
           !value.startsWith("org") && !value.contains("Default Package")]
   .filter[parentScope == null]
   .forEach[ns|ns.toDistrict(level)];
```

Listing 3.6: Suchoperation für FAMIX-Namensräume der ersten Hierarchieebene

Zu Beginn werden nur Pakete ohne Elternelemente gesucht (dritte Filteranweisung), weil diese auf der ersten Ebene des Systems liegen. Zusätzlich schließt die Operation Pakete eingebundener Bibliotheken aus. Diese Eingrenzung muss abhängig vom eingelesenen Modell angepasst werden. Für jedes zutreffende Element wird die Methode `toDistrict` aufgerufen. Der Parameter `level` zählt die momentane Hierarchieebene mit, in der sich der Algorithmus befindet.

Die Methode erstellt ein neues Distrikt-Element und weist ihm die vorhandenen FAMIX-Daten zu. In ihr wird nun rekursiv nach Kindelementen gesucht, deren `parentScope` dem aktuell verarbeiteten Objekt entspricht. Die Unterelemente werden dann ebenfalls durch die Methode `toDistrict` abgearbeitet bzw. von der Methode `toBuilding`, wenn es sich um eine Klasse handelt. Letztere unterliegt derselben Funktionsweise, liest aber zusätzlich Referenzen aus und sucht nach Unterelementen der Typen `FAMIXMethod` und `FAMIXAttribute`, um ihre Gesamtzahl zu bestimmen. Die rudimentären Daten der Methoden und Attribute werden in zusätzlichen Funktionen nach demselben Schema gespeichert und

¹⁵ `org.svis.generator/src/org/svis/generator/city/s2m/Famix2City.xtend`

dem Modell hinzugefügt. Auf eine detaillierte Betrachtung der Elementeigenschaften wird an dieser Stelle verzichtet, da zu diesem Zweck der Quelltext zu Rate gezogen werden kann. Zur Überprüfung der Verarbeitung bietet sich die erstellte XML-Datei an. Alle Felder der Klassen werden als Attribute dargestellt. Anhand eines überschaubaren Beispiels kann die Korrektheit der Implementierung getestet werden. Durch die Speicherung von Methoden und Attributen im Modell, werden diese auch in der Ausgabedatei berücksichtigt und können verglichen werden. Die Schachtelung der Elemente sollte im Voraus bekannt sein, damit die Zuordnung validiert werden kann. Die Verarbeitung einer umfangreicheren Software kann durch eine bereits umgesetzte Metapher überprüft werden. In diesem Falle boten sich sowohl „CodeCity“ als auch die Referenzmetapher an. Dabei genügt in den meisten Fällen eine stichprobenartige Überprüfung, indem ein Abschnitt mit speziellen Elementen oder starker Verschachtelung abgeglichen wird.

Im nächsten Schritt wird die Modellmodifikation durchgeführt. Die Implementierung des Layout-Algorithmus ist sehr spezifisch für die umzusetzende Metapher und bietet demnach kaum Möglichkeiten zur Abstraktion. Aus diesem Grund wird diesbezüglich nicht quelltextnah gearbeitet, stattdessen wird mehr Wert auf das grundlegende Vorgehen gelegt, wobei die Implementierung den Quelldateien entnommen werden kann.

3.6 Modellmodifikation

Die Modellmodifikation erfolgt mittels verschiedener Klassen, die die Layout-Logik kapseln und den Vorgang als Hilfsklassen unterstützen.

Die Xtend-Klasse `City2City`¹⁶ führt die ersten Ergänzungen im Modell durch und ruft dann den Layout-Algorithmus auf. Alle Distrikte und Gebäude werden aus dem übergebenen Modell gelesen, damit ihnen ihre Maße zugewiesen werden können, auf denen der Algorithmus arbeiten wird. Minimalwerte sorgen dafür, dass Längenmaße von null Einheiten bei Gebäuden ohne Attribute oder Methoden vermieden werden. Dies würde zu einer nicht erkennbaren Darstellung führen.

Wie schon zuvor erwähnt, erfolgt die Implementierung des Layout-Algorithmus anhand des vorhandenen Pseudocodes innerhalb der Java-Klasse `CityLayout`¹⁷. Dieser arbeitet ohne Abstände zwischen den Elementen und ist nicht vollständig spezifiziert, aber eine detaillierte Betrachtung ist auch nicht erforderlich. Alle Konzepte sind leicht auf Java-Äquivalente übertragbar. Eine Übersicht des Aufbaus liefert Abbildung 3.5. Die farblichen Abgrenzungen verdeutlichen Sinneinheiten, in die der Algorithmus untergliedert werden kann.

¹⁶ org.svis.generator/src/org/svis/generator/city/m2m/City2City.xtend

¹⁷ org.svis.generator/src/org/svis/generator/city/m2m/CityLayout.java

Require: *elements* are sorted by size, descending
Ensure: *elements* are efficiently laid out efficiently and without overlapping

1. $ptree.root.rectangle.size \leftarrow \sum_i elements[i].size$
2. $covrec \leftarrow (0,0)$
3. **for** *el* in *elements* **do**
4. *pnodes* \leftarrow empty leaf nodes in *ptree* with $pnode.rectangle.size \geq el.size$
5. *preservers* \leftarrow new dictionary
6. *expanders* \leftarrow new dictionary
7. **for** *pnode* in *pnodes* **do**
8. **if** placing *el* in *pnode* would preserve the size of *covrec* **then**
9. *waste* \leftarrow amount of remaining space if *pnode* was split to place *el*
10. add *pnode* : *waste* to *preservers*
11. **else**
12. *ratio* \leftarrow aspect ratio of *covrec* if *pnode* was used to place *el*
13. add *pnode* : *ratio* to *expanders*
14. **end if**
15. **end for**
16. **if** *preservers* is not empty **then**
17. *targetnode* \leftarrow key corresponding to the lowest value in *preservers*
18. **else**
19. *targetnode* \leftarrow key corresponding to value in *expanders* closest to 1
20. **end if**
21. **if** *targetnode.rectangle* perfectly fits *el* **then**
22. *fitnode* \leftarrow *targetnode*
23. **else**
24. *fitnode* \leftarrow perfect fitting node for *el* after splitting *targetnode*
25. **end if**
26. *fitnode.occupied* \leftarrow *TRUE*
27. move *el* to *fitnode.rectangle.position*
28. **if** *fitnode* is a boundary expander **then**
29. expand *covrec* to the newly covered area
30. **end if**
31. **end for**

Abbildung 3.5: „CodeCity“ Pseudocode
(in Anlehnung an [Wettel 2010, 36])

Der Algorithmus ordnet die Elemente eines einzelnen Distrikts an. Zu diesem Zweck müssen die Unterelemente absteigend nach der Größe ihrer Grundfläche geordnet werden. Der gelbe Abschnitt markiert die Initialisierung der Datenstrukturen. *Ptree* ist ein KD-Baum, der über ein Wurzelement verfügt, welches die Informationen eines Rechtecks enthält (vgl. [Wettel 2010, 35]). Es stellt die zur Verfügung stehende Gesamtfläche dar, in der die Kindelemente organisiert werden. Als Anfangsmaße erhält es die aufsummierten Längen und Breiten der Elemente. Sollte ein Unterelement keine Größe definiert haben, weil es sich um einen verschachtelten Distrikt handelt, dessen Größe noch ermittelt werden muss, wird an dieser Stelle der Algorithmus rekursiv aufgerufen. *Covrec* ist ein zusätzliches Rechteck, um die tatsächlich durch Elemente belegte Fläche zu verfolgen. *Pnodes* werden alle freien

Knoten aus `pree` zugeordnet, in denen das aktuelle Element `e1` untergebracht werden kann.

Im orangenen Abschnitt werden die Ergebnisse aus `pnodes` klassifiziert. Für jeden leeren Knoten wird ermittelt, ob die Positionierung von `e1` im aktuellen Knoten die Größe von `covrec` erhält oder erweitert. Im Falle von ersterem wird der Knoten im Dictionary¹⁸ `pre-server` mit dem verbleibenden Raum gespeichert, ansonsten wird er `expander` zugewiesen, inklusive dem entstehenden Seitenverhältnis.

Der dritte Abschnitt (grün) wählt den Zielknoten aus, in dem `e1` platziert wird. Knoten die `covrec` erhalten werden vorgezogen, da eine kompakte Anordnung angestrebt wird.

Der blaue Abschnitt schneidet den Zielknoten passend zu, insofern dies nötig ist, und kennzeichnet den passenden Knoten als „belegt“. Im letzten Schritt erhält `e1` seine ermittelte Position und `covrec` wird auf den neuen Wert aktualisiert, sollte dies nötig sein. Wetzel [2010] erklärt den Algorithmus zusätzlich an einem Beispiel, das Abbildung 3.6 entnommen werden kann.

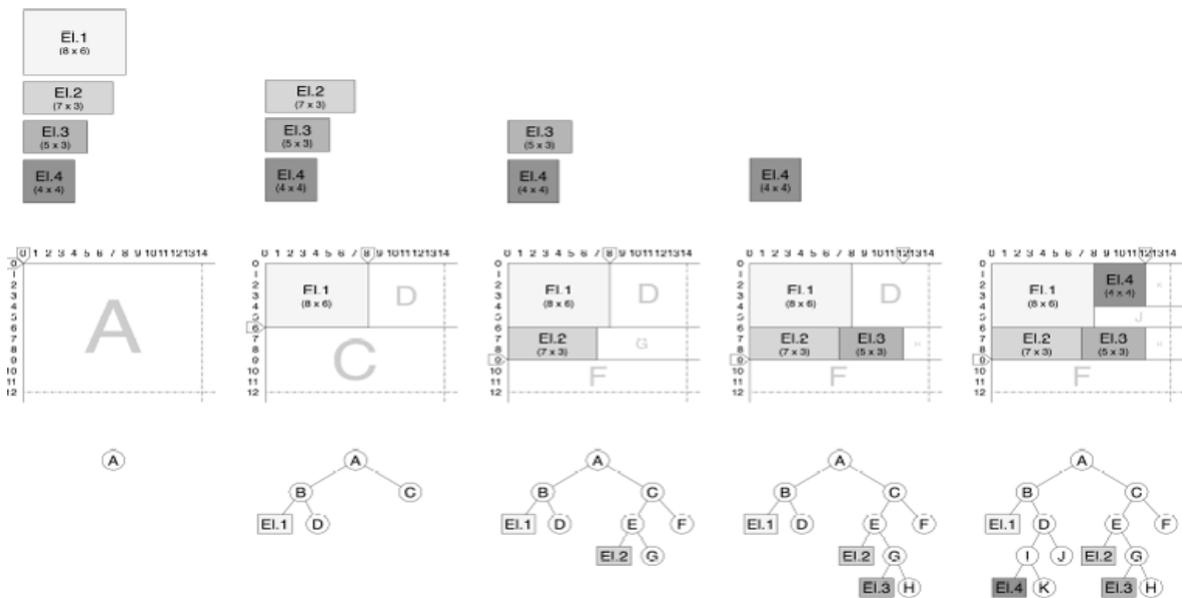


Abbildung 3.6: Beispiel: Vier Elemente (oben) angeordnet (Mitte) mittels KD-Baum (unten) (vgl. [Wetzel 2010, 37])

Rechteck A verdeutlicht das Wurzelement des KD-Baums. `Covrec` wird durch die kleinen Pfeile an der Skala verfolgt; sie zeigen die maximale Ausdehnung der belegten Fläche an. Die Erweiterung des KD-Baums erfolgt während des Zuschneidens des ausgewählten Knotens (blauer Abschnitt), wodurch neue Blätter entstehen. Der Algorithmus schneidet das

¹⁸ „CodeCity“ wurde in Smalltalk programmiert, woran auch der Pseudocode angelehnt ist. Dictionaries entsprechen Java-Maps/-Hashmaps/-LinkedHashmaps.

vorhandene Rechteck, bspw. den Wurzelknoten A, horizontal anhand der Höhe des Elements in zwei Kindknoten B und C. In den ersten Unterknoten wird `e1` platziert und erzeugt durch einen Schnitt anhand der Elementbreite ein weiteres Blatt D. Dieser Vorgang wird für jedes einzufügende Element wiederholt.

Bei genauerer Betrachtung fällt auf, dass `covrec` bei jeder Anwendung des Algorithmus mit `(0,0)` initialisiert wird (vgl. Abbildung 3.5, Zeile 2). Dies hat zur Folge, dass jeder Distrikt ein unabhängiges Koordinatensystem verwendet. In X3D ist die Verwendung relativer Koordinaten durch Gruppierung denkbar, allerdings ermöglichen absolute Positionierungen die Abarbeitung unabhängig von Verschachtelungen. Deshalb beinhaltet die Implementierung nach der Anordnung aller Elemente eine Anpassung, in der alle Kindelemente Koordinaten abhängig von der Position ihrer Elternelemente erhalten (vgl. Listing C.2).

Um den Algorithmus ohne größere Änderungen übernehmen zu können, sind einige Hilfsklassen nötig (vgl. Abbildung C.1), bspw. `CityKDTree`¹⁹ und damit einhergehend `CityKDTreeNode`²⁰. Erstere ist der Behälter für die Datenstruktur und verfügt über die Baumwurzel des Typs `CityKDTreeNode`. Im Knotentyp sind die Informationen gespeichert, die die Belegung betreffen, die Maße des Rechtecks, das er darstellt, und welche Unterknoten ihm zuzuordnen sind. Der Baum selbst beinhaltet die Methode, um leere Blätter in einer Liste zu sammeln und weiterzugeben.

Die Klasse `CityRectangle`²¹ übernimmt die Maße der Gebäude, um auf ihnen arbeiten zu können. Aufgrund der Tatsache, dass Entitäten des Modells generiert werden, können die Klassen nicht um das Java-Interface `Comparable` erweitert werden, da es eine manuelle Änderung erforderlich macht, auf die verzichtet werden soll. Das Interface wird gebraucht, um einen Sortieralgorithmus zu implementieren, damit die Gebäude bzw. die Rechtecke der Größe nach geordnet werden können. Somit implementiert `CityRectangle` das Interface stellvertretend (vgl. Listing C.1), erhält die notwendigen Daten und führt die Positionsdaten anschließend mittels eines Verweises auf das ursprüngliche Element zurück.

Zur Überprüfung der Implementierung empfiehlt sich auch hier, zuerst ein kleines System verarbeiten zu lassen. Dabei sollte der Algorithmus schrittweise validiert werden, bspw. entsprechend der Sinnabschnitte aus Abbildung 3.5. Durch die Ausgabe von Informationen in

¹⁹ org.svis.generator/src/org/svis/generator/city/m2m/CityKDTree.java

²⁰ org.svis.generator/src/org/svis/generator/city/m2m/CityKDTreeNode.java

²¹ org.svis.generator/src/org/svis/generator/city/m2m/CityRectangle.java

der Konsole lässt sich leicht überprüfen, ob die erzeugten mit den erwarteten Werten übereinstimmen. Den Quelltext in verschiedene Methoden zu untergliedern kann durchaus hilfreich sein, ebenso wie Werkzeuge zur Fehlererkennung.

Sobald die Richtigkeit des Algorithmus gesichert ist, kann die Implementierung der letzten Transformation erfolgen. Alternativ kann diese auch den Tests des Algorithmus vorgezogen werden, um die Datenausgabe durch eine visuelle Darstellung zu überprüfen.

3.7 Modell-zu-Text-Transformation

Die Modell-zu-Text-Transformation überträgt abschließend die erzeugten Daten in eine X3D-Datei, die die fertiggestellte Visualisierung enthalten wird. Die Klasse `City2X3D`²² muss dazu das Interface `IGenerator` implementieren, da hier eine andere Schnittstelle erforderlich ist. Durch das Interface muss die Methode `doGenerate` überschrieben werden. Neben der Ausgabe der Log-Informationen filtert sie alle Entitäten des Inputs und speichert sie in einer Liste. Danach wird eine Datei erzeugt, die aus drei Teilen besteht: X3D-Head, X3D-Model und X3D-Tail (vgl. Listing 3.7).

```
1. //file structure
2. fsa.generateFile("famix/model.x3d", toX3DHead
                       + entities.toX3DModel
                       + toX3DTail)
```

Listing 3.7: Operation zur Erzeugung einer neuen X3D-Datei

Notwendige Metadaten und Tags sind ausgelagert worden und werden über diese Methoden ins Dokument geschrieben. Diese Funktionen wurden in der Datei `X3DUtills.xtend`²³ definiert. Die Methode `toX3DModel` verleiht den Komponenten ihr Aussehen. Sie beinhaltet eine Schleife, die die übergebenen Entitäten durchläuft. Abhängig von deren Typ wird sie an eine von zwei Methoden übergeben. Pakete werden von `toDistrict` bearbeitet und Klassen von `toBuilding`. Listing 3.8 verdeutlicht den Aufbau der beiden Methoden.

Der vollqualifizierte Name der Entitäten wird zuerst auf Sonderzeichen überprüft. Da alle Elemente im Voraus absolute Positionen erhalten haben, spielt bei diesem Transformationsschritt die Reihenfolge der Abarbeitung keine Rolle. Der Aufbau des Templates²⁴ ist dem Entwurf, welcher der Analyse der Zielmetapher folgte (vgl. Abbildung D.1), nachempfunden. Distrikte und Gebäude sind sich in ihrer Darstellung in X3D so ähnlich, dass beide Methoden identisch aufgebaut sind. Dies kann sich später durch etwaige Erweiterungen allerdings ändern, weswegen die getrennte Abarbeitung beibehalten wurde.

²² `org.svis.generator/src/org/svis/generator/city/m2t/City2X3D.xtend`

²³ `org.svis.generator/src/org/svis/generator/X3DUtills.xtend`

²⁴ Templateausdrücke in Xtend, eingeschlossen in drei Apostrophen, dienen der Quelltextgenerierung. Variable Ausdrücke werden durch Guillemets gekennzeichnet.

```

1. def String toDistrict(Entity entity){
2.   //transformation to a District-component
3.   var fqcn = entity.fqn.replaceAll('\\.', '_').replaceAll('<', '_')
      .replaceAll('>', '_');
4.   var x = entity.position.x;
5.   var y = entity.position.y;
6.   var z = entity.position.z;
7.   var width = entity.width;
8.   var length = entity.length;
9.   var height = entity.height;
10.  var color = entity.color;
11.  '''
12.  <Group DEF='«fqcn»'>
13.    <Transform translation='«x + " " + y + " " + z»'>
14.      <Shape>
15.        <Box size='«width + " " + height + " " + length»'></Box>
16.        <Appearance>
17.          <Material diffuseColor='«color»'></Material>
18.        </Appearance>
19.      </Shape>
20.    </Transform>
21.  </Group>
22.  '''}

```

Listing 3.8: Datenübertragung auf X3D

Nach der Implementierung sollte die Visualisierung getestet werden. Zuerst sollte an einer einfacheren Datenstruktur ermittelt werden, ob alle Objekte korrekt angeordnet sind und sich nicht überlagern. Dann sollten auch kompliziertere Systeme auf fehlerhafte Darstellung überprüft werden. Abbildung D.2 zeigt das erste Zwischenergebnis der Implementierung.

3.8 Anpassungen und Ergänzungen

Um die Stadtmetapher zu vervollständigen müssen abschließend noch drei Aspekte hinzugefügt werden: die Darstellung verschachtelter Klassen, die Abstände zwischen den dargestellten Elementen und die Farbabstufungen. Dies soll im Folgenden erläutert werden.

Die Verschachtelung löst „CodeCity“ dadurch, dass innen liegende Klassen demselben Paket zugeordnet werden, dem auch das Elternelement angehört. Diese Konzeption ist nicht in Wettel [2010] angeführt, lässt sich aber durch die Überprüfung von Visualisierungen bestätigen; ansonsten könnten Relationsbeziehungen nicht fehlerfrei dargestellt werden. Um dieses Problem aufzulösen, müssen Anpassungen im Metamodell und in der *Famix2City* erfolgen. Die Methode `toBuilding`, die für die Datenzuweisung für Klassen eingesetzt wird, muss um einen Parameter ergänzt werden, damit festgehalten werden kann, in welchem Paket sich das aktuelle Gebäude befindet. Dieser Verweis wird in einer neuen Entitäteneigenschaft gespeichert: `modelContainer`. Wird nun eine Klasse nach inneren Elementen durchsucht, wird die Methode mit der Referenz auf das übergeordnete Paket aufgerufen und diesem hinzugefügt.

Um Abstände zwischen den Entitäten zu erzeugen, wird *CityLayout* angepasst, indem drei Variablen hinzugefügt werden:

- `SYST_HORIZONTAL_GAP`: Abstand zwischen den Kindelementen des System-Bereichs
- `PCKG_HORIZONTAL_MARGIN`: Rand zwischen Distrikten und ihren Kindelementen
- `PCKG_HORIZONTAL_GAP`: Abstand zwischen den Kindelementen von Distrikten

Die Abstände werden als Pufferzone bei der Erstellung der Rechtecke während der Transformation eingehen. Sie werden zum einen bei der Errechnung der Wurzelfläche eines neuen KD-Baumes berücksichtigt, damit der vorhandene Platz nicht zu klein ausfällt. Zum anderen fließen sie bei der Übertragung der Modelldaten auf die zur Anordnung genutzten Rechtecke ein. Das Layout wird anhand der neuen Bruttofläche der Elemente berechnet. Der Vorteil hierbei ist, dass die Klasse `CityRectangle` so konzipiert ist, automatisch den Mittelpunkt der ihr zugewiesenen Fläche zu errechnen, der als Koordinate an die zugehörige Entität weitergegeben wird. Auf diese Weise ist das Gebäude automatisch im Zentrum ausgerichtet und erhält einen umliegenden Rand zu seinen Nachbarn. Dieser beträgt die Hälfte des für die jeweilige Variable definierten Wertes, sodass zwei nebeneinander angeordnete Elemente durch den vollen Abstand voneinander getrennt werden. Da dieser umliegende Rand aber auch bei Gebäuden eingerechnet ist, die sich an den Grenzen des Distrikts befinden, muss dies bei der Größenfestlegung der Distrikte beachtet werden. Dies hat auch Auswirkungen auf die Erzeugung absoluter Koordinatenwerte zusammen mit dem Abstand zum Elternelement, wie bspw. Listing C.2 verdeutlicht. Durch diese simple Anpassung nähert sich die Visualisierung sehr stark an ihr Vorbild an (vgl. Abbildung D.3)

Die Einfärbung der visualisierten Elemente erfolgt unter Zuhilfenahme einer eigenen Klasse zur Erstellung von Farbverläufen im RGB-Raum, basierend auf `java.awt.Color`²⁵. Bei Betrachtung einiger „CodeCity“-Beispiele fällt auf, dass Pakete gleichmäßig skaliert eingefärbt werden, abhängig von der Tiefe ihrer Verschachtelung. Für Klassen hingegen wird eine von fünf Farbnuancen anhand der Anzahl ihrer Quelltextzeilen ausgewählt. Diese Designentscheidung hat Wettel möglicherweise auf der Basis getroffen, dass das menschliche Auge kleine Farbunterschiede nicht wahrnehmen kann (vgl. [Wettel 2010, 73]). Aus diesem Grund wurde die Hilfsklasse `CityColorGradient`²⁶ so konzipiert, dass sie für beide Anwendungsfälle genutzt werden kann.

`CityColorGradient` nutzt zwei Farbwerte als Anfangs- und Endpunkt des Gradienten und definiert einen Definitionsbereich (engl. *domain*), der auf den Farbverlauf abgebildet

²⁵ <http://docs.oracle.com/javase/7/docs/api/java/awt/Color.html>

²⁶ org.svis.generator/src/org/svis/generator/city/m2m/CityColorGradient.java

wird. Die Anzahl an Unterteilungen legt fest, in wie viele Bereiche sich beide Räume zerlegen. Es wird ein vereinfachter Konstruktor genutzt, der die Parameter des Definitionsbereichs auslöst. Aus der Angabe der Schrittweite wird automatisch ein Definitionsbereich erzeugt, deren obere Grenze die Schrittweite und die untere dem Wert null entspricht, sodass genau ein ganzzahliger Wert exakt auf eine Nuance aus dem Gradienten abgebildet wird. Diese Vereinfachung wird bei der Färbung der Distrikte angewandt, da dort jeder Ebene genau eine Farbe zugeordnet wird.

Die Berechnung des Farbverlaufs ist in beiden Fällen identisch. Es wird die Distanz zwischen den RGB-Komponenten der beiden Farben errechnet und durch die entsprechende Anzahl an Unterteilungen dividiert. Durch schrittweises Aufsummieren entstehen die Zwischenfarben, die zusammen mit der Obergrenze ihres Teilbereiches als Schlüssel-Wert-Paar abgespeichert wird. Um die entsprechende Farbe für ein Element abzurufen, wird der Wert der jeweiligen Determinante mit den Schlüsseln aus der Datenstruktur abgeglichen und wahlweise als Array oder Zeichenkette zurückgegeben. Die Randwerte der Gradienten sind parametrisiert und an die Darstellung der Referenzmetapher angepasst, was zu einer geringfügigen Abweichung von „CodeCity“ führt (vgl. Abbildung D.4).

4 Abstrahiertes Prozessmodell

Wie zu Beginn dieser Arbeit erwähnt, ist eine Abstraktion des Implementierungsprozesses nur bei nicht-metapherspezifischen Abläufen möglich. Abbildung 4.1 zeigt einen abstrahierten Prozess für die generative und modellgetriebene Softwarevisualisierung. Die dargestellten Artefakte sind über ungerichtete Assoziationen mit jenen Aufgaben verbunden, die Gebrauch von ihnen machen. Es ist ersichtlich, dass am Anfang der Konzeption in Betracht gezogen werden muss, ob eine Vorlage existiert. „CodeCity“ hat im Vorfeld genug Informationen geliefert, um Metriken zu identifizieren, die andernfalls durch Tests hätten ermittelt werden müssen. Eine Betrachtung der Referenzmetapher ist nicht in jedem Fall nötig oder sinnvoll. Dieser Schritt wäre auch bei der Implementierung der Stadtmetapher vernachlässigbar gewesen, ermöglichte aber eine intensivere Einarbeitung und ein besseres Verständnis für den Visualisierungsgenerator.

Handelt es sich um eine neu konzipierte Metapher ist man zwar freier in der Festlegung der Metriken, es muss aber mehr Zeit in die Feinabstimmung investiert werden. „CodeCity“ ist ein erprobtes Werkzeug, in das mehrere Jahre Entwicklung geflossen sind. Die Konfiguration der Darstellung muss gut angepasst werden, damit die Informationen in der Visualisierung für den Nutzer klar erkennbar und auswertbar sind. Eine erste Beispielvisualisierung auf der Zielplattform ist nicht nur eine veranschaulichende Zusammenfassung der nötigen Metriken, sondern impliziert auch den Aufbau des Quelltextes, in den das Modell abschließend transformiert wird und gibt Aufschluss über die Lage der Elemente zueinander und daraus folgende Berechnungen, die in der Modellmodifikation gebraucht werden.

Die Implementierung beginnt mit der Erstellung des Metamodells. Im Prozessmodell zeigen die Gruppierungen, in welchen Teilen des Generators die verwendeten Ressourcen vorliegen. Das Metamodell befindet sich im Stammverzeichnis der Zielmetapher²⁷.

Es wird automatisch bei der Erstellung eines neuen Xtext-Projekts kreiert und verfügt über die durch die Erzeugung der Infrastruktur generierten Ressourcen. Der Workflow zur Softwarevisualisierung im Testverzeichnis des Generators stellt den Kontext der Abarbeitung dar und liest die Quelldaten ein. Die Modell-Slots verbinden die Transformationsschritte, indem sie das bearbeitete Metaphermodell an diesen Punkten auslesen und gleichermaßen wieder zur Verfügung stellen. Die Output-Writer werden hauptsächlich zur Kontrolle genutzt, um die Zwischenergebnisse der Modellumwandlungen überprüfen zu können.

²⁷ org.svis.xtext.<modellname>

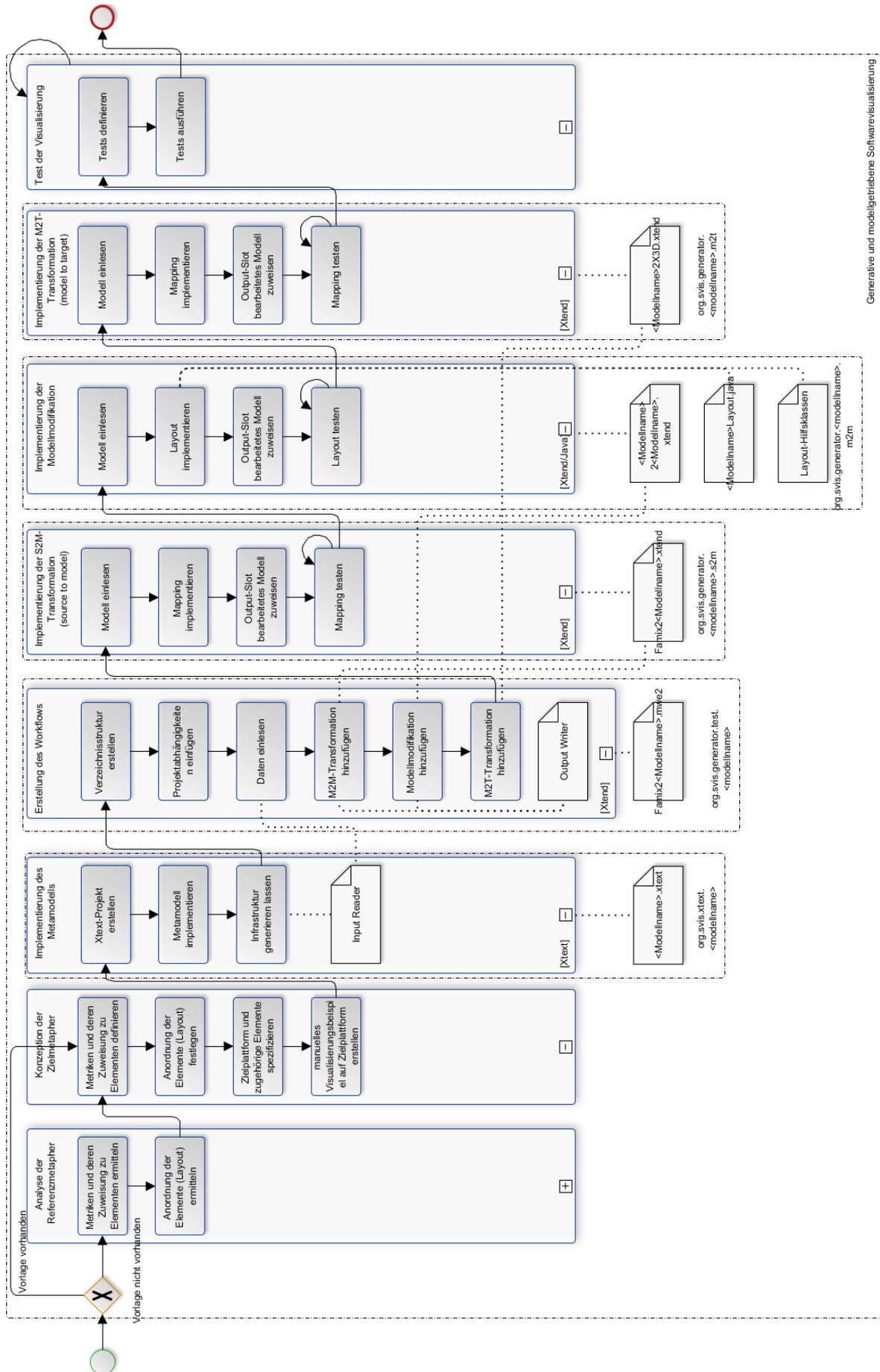


Abbildung 4.1: Abstrahiertes Prozessmodell basierend auf der Implementierung der Stadtmetapher

Die Transformationen ähneln sich sehr im Aufbau. Nach dem Einlesen des aktuellen Modells, erfolgt die Implementierung der Umformungslogik. Das Ergebnis wird im vorgesehenen Slot hinterlegt und für weitere Bearbeitungen zugänglich gemacht. Das wiederholte Testen der Zwischenergebnisse beugt der Weiterführung von Logikfehlern vor. Abweichungen in den berechneten Ergebnissen haben auf kleinere Beispiele selten Auswirkungen, können aber in größeren Projekten zu Fehldarstellungen führen. In Verbindung mit einer analytischen Konzeption der Zielmetapher, vermeiden iterative Tests häufige Überarbeitung und Neumodellierung. Zum Abschluss der Implementierung bietet sich das Definieren von Tests an, die das gesamte System umfassen. Auch angebundene Projekte können dabei einfließen, bspw. das Verhalten der Metapher im Zusammenhang mit der Benutzerschnittstelle.

Zusammenfassend lässt sich feststellen, dass ein abstrahierendes Prozessmodell der Strukturierung durch die Verwendung von Modellen zu verdanken ist. Sie organisieren die vorhandenen Problem- und Lösungsräume und Standardisieren den Ablauf der verschiedenen Transformationen. Dem stehen die unstrukturierten, metapherspezifischen Vorgänge gegenüber, für die keine Abstraktion möglich ist.

5 Zusammenfassung und Ausblick

Im Verlauf dieser Arbeit wurde die Stadtmetapher für den Visualisierungsgenerator implementiert, anhand derer eine Abstraktion erfolgte, um ein Prozessmodell für die generative und modellgetriebene Softwarevisualisierung zu erarbeiten.

Dabei stellte die Anforderungsanalyse den Ausgangspunkt dar, durch die eine ausgiebige Planung der Implementierung möglich wurde. Darauf aufbauend erfolgte die Erstellung eines Metamodells mittels Xtext. Der Workflow und die Transformationen, verfasst in Xtend, orientierten sich teilweise an der Referenzmetapher und teilweise an der Vorlage „Code-City“. Auch wenn sich während der Abstraktion herausstellte, dass eine Referenzmetapher nicht zwingend erforderlich ist, beschleunigte ihre Analyse die Implementierung erheblich und sollte nicht unterschätzt werden. Im Zentrum des Prozessmodells stehen eine annähernd exakte Ermittlung der Anforderungen und iterative Tests, um Fehler zu vermeiden.

Es sei zu erwähnen, dass die implementierte Stadtmetapher lediglich einen Prototypen darstellt. „CodeCity“ bietet als umfangreiches Werkzeug mehr Analysemöglichkeiten, als der Generator zurzeit umsetzen kann. Demzufolge müsste bspw. eine Visualisierung für Methoden und Attribute hinzugefügt werden (vgl. [Wettel 2010, 38ff]), um Nutzern die gleichen Informationen zugänglich zu machen wie die Recursive Disk-Metapher. Ebenso sollte in Betracht gezogen werden, die Age Maps und Time Travels umzusetzen, die „CodeCity“ nutzt, um Systemevolution mittels Farbskalen zu visualisieren (vgl. [Wettel 2010, 60ff]).

In Anbetracht der Anforderungen, die das generative Paradigma an den Visualisierungsgenerator stellt, müsste genau genommen eine stärkere Modularisierung vorgenommen werden. Zu diesem Zweck wäre eine Auslagerung des Layout-Algorithmus nötig, ebenso wie den Mapping-Konfigurationen bzgl. der Entitäteneigenschaften, da bei der Softwarevisualisierung diese Vorgänge die kleinsten gemeinsamen Komponenten der Mitglieder dieser Systemfamilie darstellen. Das Konfigurationswissen müsste dahingehend erweitert werden, dass es legitime Metapher-Layout-Kombinationen erkennt und für den Anwender optimiert. So bliebe dem Nutzer nicht nur die Auswahl der entsprechenden Metapher, sondern es ließen sich vor allem generische Layouts kreieren.

Wie sich durch das abstrahierte Prozessmodell zeigen lässt, sorgt vor allem das modellgetriebene Paradigma für eine strukturierte Organisation der Problem- und Lösungsräume und trägt einen großen Anteil dazu bei, eine Abstraktion für die Softwarevisualisierung zu ermöglichen. Die „großen Unbekannten“ des Prozessablaufs bleiben nach wie vor die metapherspezifischen Abläufe, wie bspw. die Implementierung des Layouts, die nicht durch Modelle erfasst werden können.

Zusammenfassend betrachtet, bietet die generative Softwarevisualisierung vielversprechende Ansätze, um Softwareanalyse zu verbessern. Während Werkzeuge wie „CodeCity“ umfangreich aber stark spezialisiert sind, bietet ein Visualisierungsgenerator potenziell die Vorteile einer Vielzahl von Metaphern, indem der Anwender die Modifikationen in Abhängigkeit von seiner zu visualisierenden Software auswählen kann.

Anhang A – Metamodell Recursive Disk-Metapher

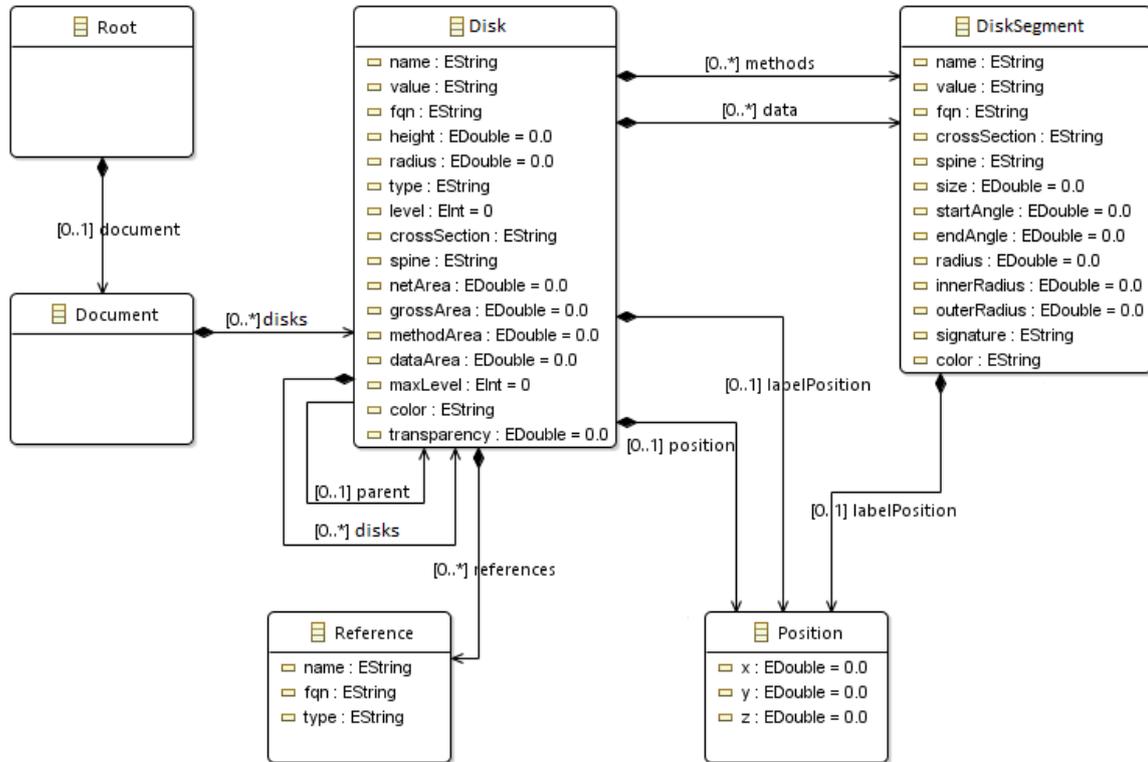


Abbildung A.1: Klassendiagramm der Referenzmetapher mit Attributen

Anhang B – Hilfestellung für Eclipse-Konfigurationen

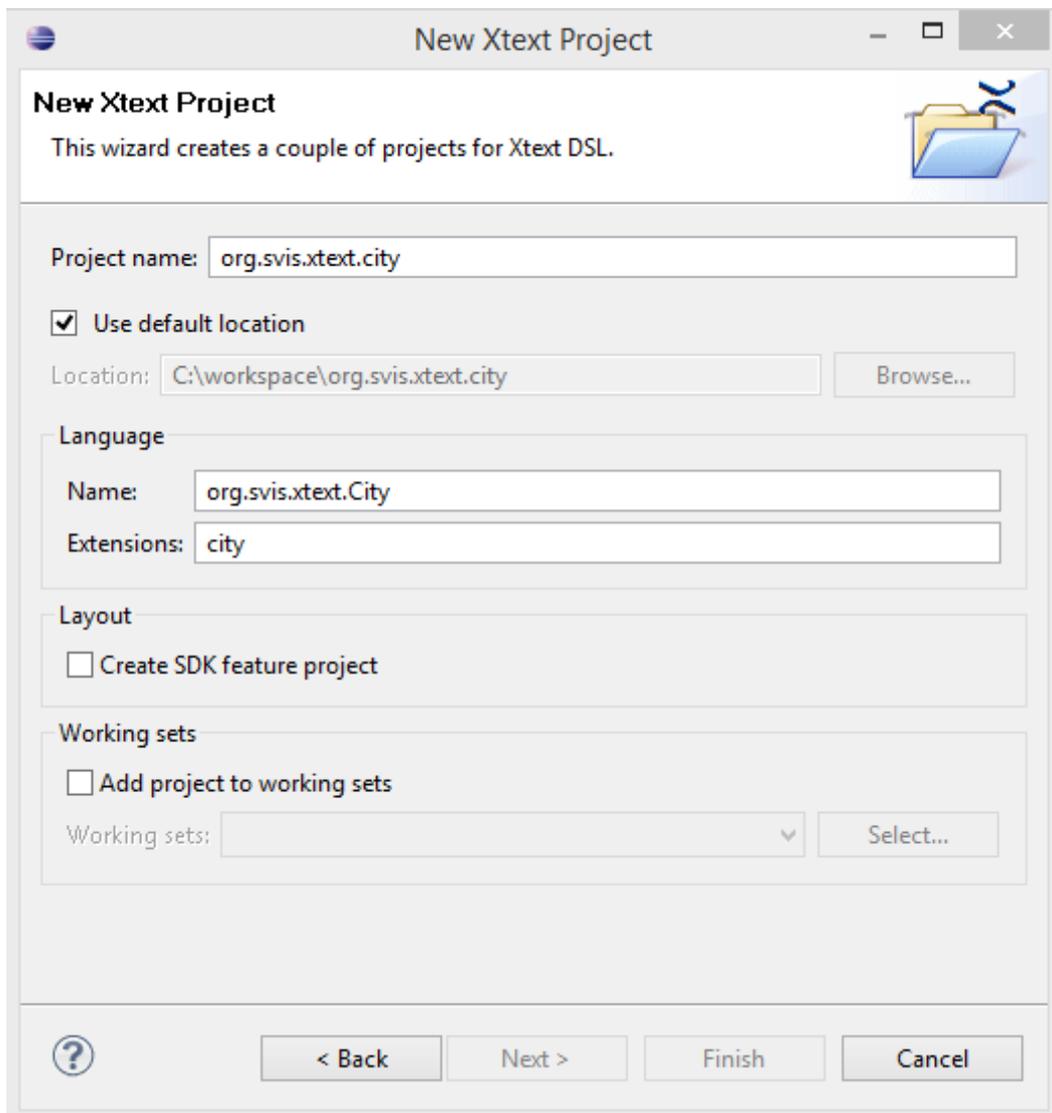


Abbildung B.1: Eclipse-Assistent zur Erstellung eines neuen Xtext-Projekts

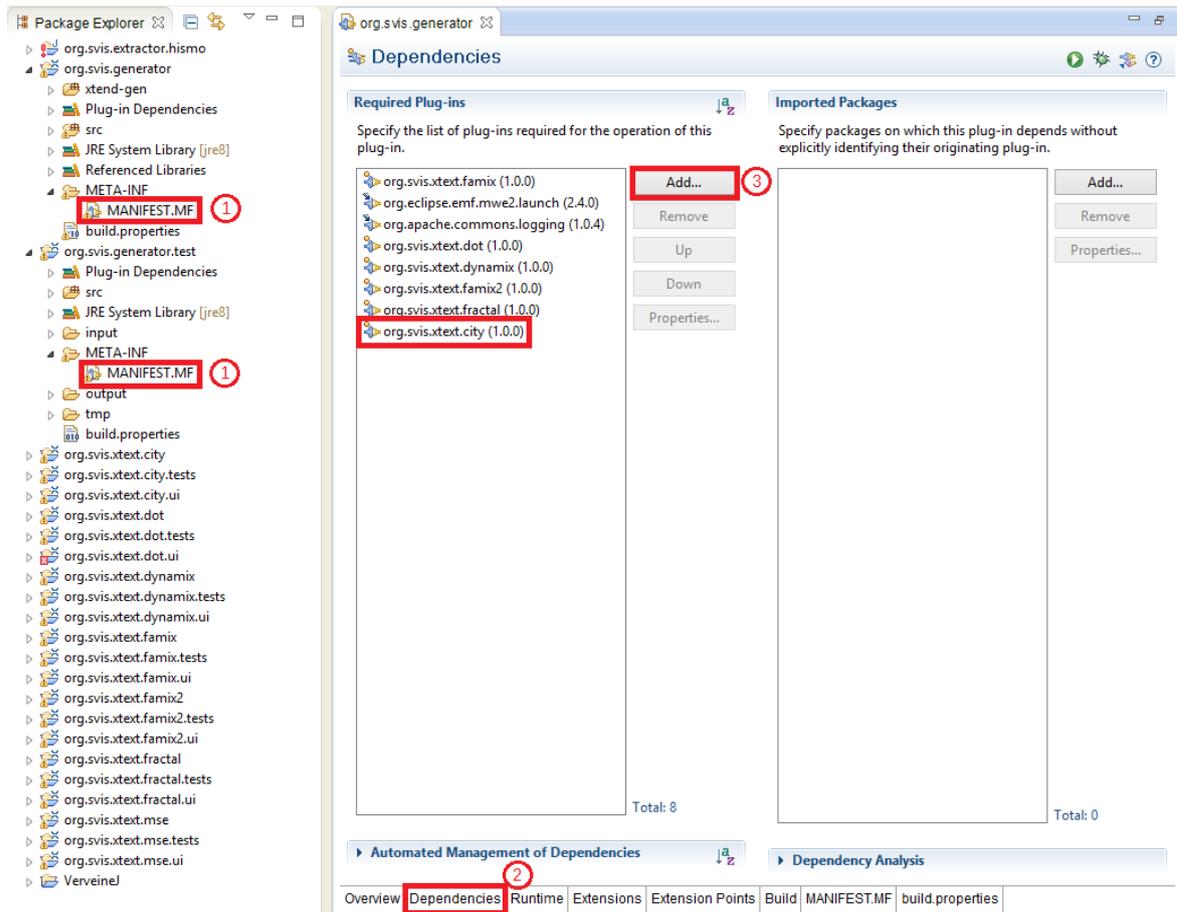


Abbildung B.2: Projekt-Abhängigkeiten über MANIFEST.MF hinzufügen

Anhang C – Konzepte zur Durchführung der Modellmodifikation

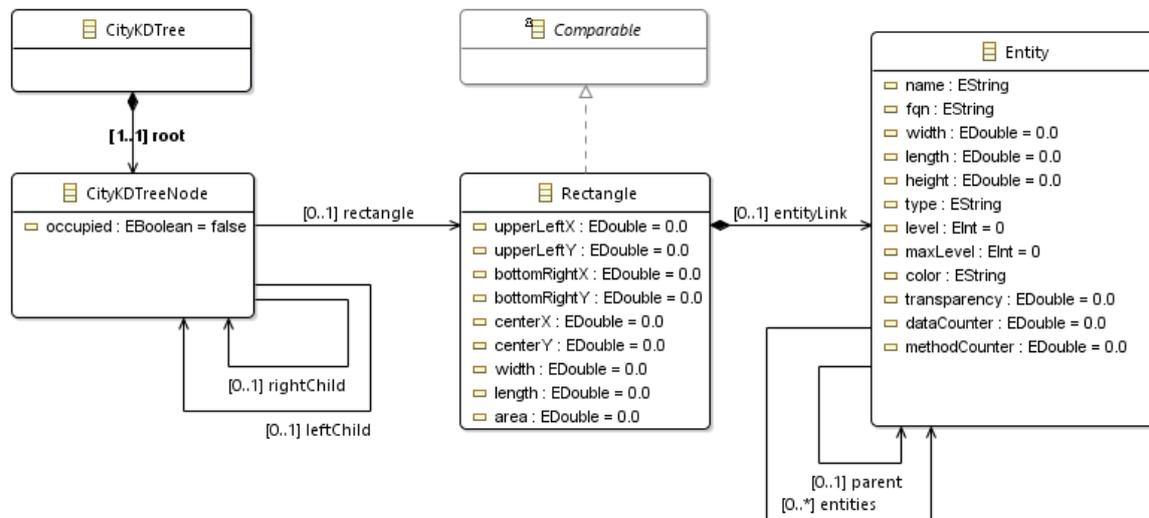


Abbildung C.1: Hilfsklassen des Layout-Algorithmus (Entity-Klasse vereinfacht)

```

1. //..
2. public class CityRectangle implements Comparable<CityRectangle> {
3. //..
4.
5. @Override
6. public int compareTo(CityRectangle second) {
7.
8. //1. Criterion: entities with larger base area shall be aligned
   first
9. int firstComparison = Double.compare(this.area, second.area);
10. if(firstComparison ==0){
11.
12. //2. Criterion: if base area is equal, entities with a taller
   height are arranged first (otherwise they block user's sight)
13. int secondComparison = Double.compare(this.entityLink.getHeight(),
   second.entityLink.getHeight());
14. if(secondComparison ==0){
15.
16. //3. Criterion: if height is equal, align those with a larger
   width first
17. int thirdComparison = Double.compare(this.width, second.width);
18. if(thirdComparison ==0){
19.
20. //Default: if everything fails, align those first that were
   inserted into list earlier
21. return 1;
22.
23. }else{ return thirdComparison; }
24. }else{ return secondComparison; }
25. }else{ return firstComparison; }
26. }
27. //..
28. }
  
```

Listing C.1: Implementierung des Sortieralgorithmus für Rectangle

```
1. private static void adjustPositions(EList<Entity> children, double
    parentX, double parentY, double parentZ){
2.     for(Entity e: children){
3.         double centerX = e.getPosition().getX();
4.         double centerZ = e.getPosition().getZ();
5.         double centerY = e.getPosition().getY();
6.         e.getPosition().setX(centerX+parentX+PCKG_horizontalMargin-
            PCKG_horizontalGap/2);
7.         e.getPosition().setZ(centerZ+parentZ+PCKG_horizontalMargin-
            PCKG_horizontalGap/2);
8.         e.getPosition().setY(centerY+(e.getLevel()-1)*DISTRICT_HEIGHT);
9.
10.        if(e.getType() == "FAMIX.Namespace"){
11.            double newUpperLeftX = e.getPosition().getX()-e.getWidth()/2;
12.            double newUpperLeftZ = e.getPosition().getZ()-e.getLength()/2;
13.            double newUpperLeftY = e.getPosition().getY()-e.getHeight()/2;
14.            adjustPositions(e.getEntities(), newUpperLeftX, newUpperLeftY,
                newUpperLeftZ);
15.        }
16.    }
17. }
```

Listing C.2: Berechnung absoluter Elementpositionen unter Berücksichtigung von Abständen

Anhang D – Entwicklungsstadien der Stadtmetapher

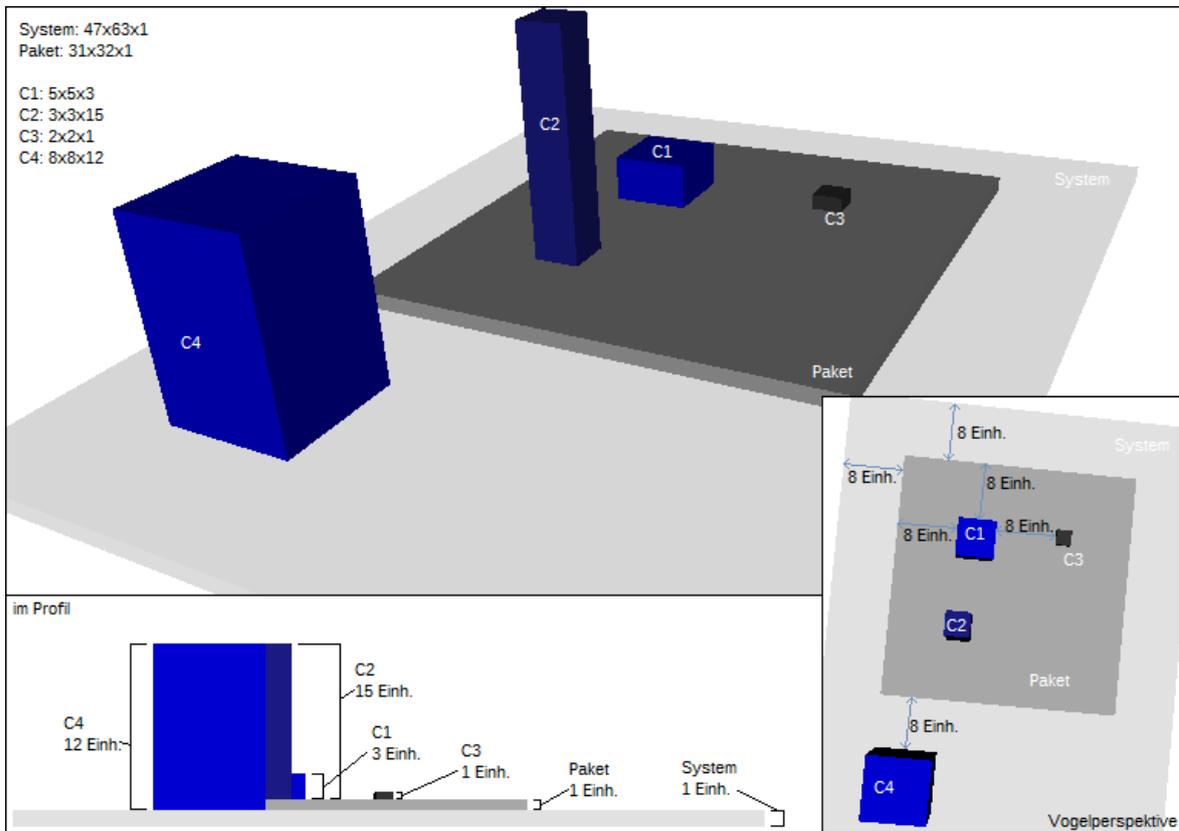


Abbildung D.1: Zusammenfassende Beispielvisualisierung in X3D

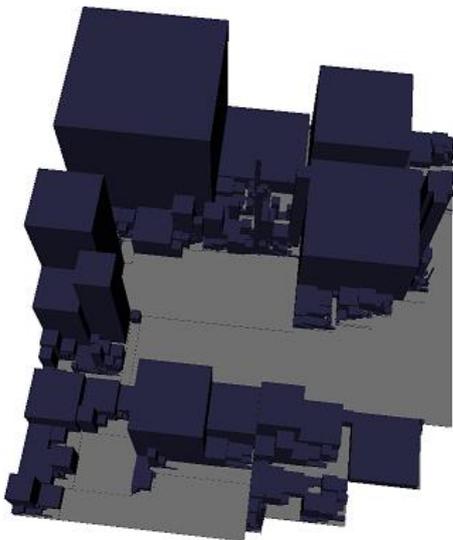


Abbildung D.2: Visualisierung nach der Implementierung der Kernaspekte

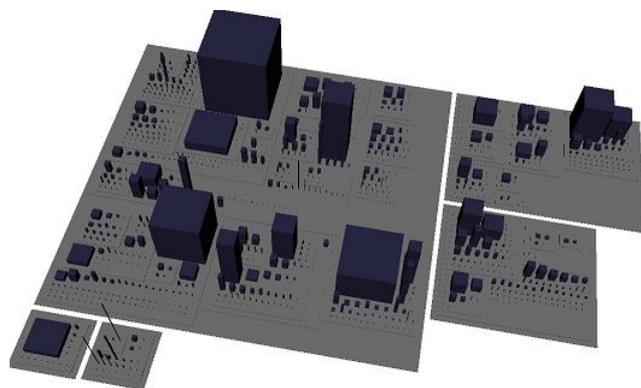


Abbildung D.3: Visualisierung nach der Implementierung von Abständen

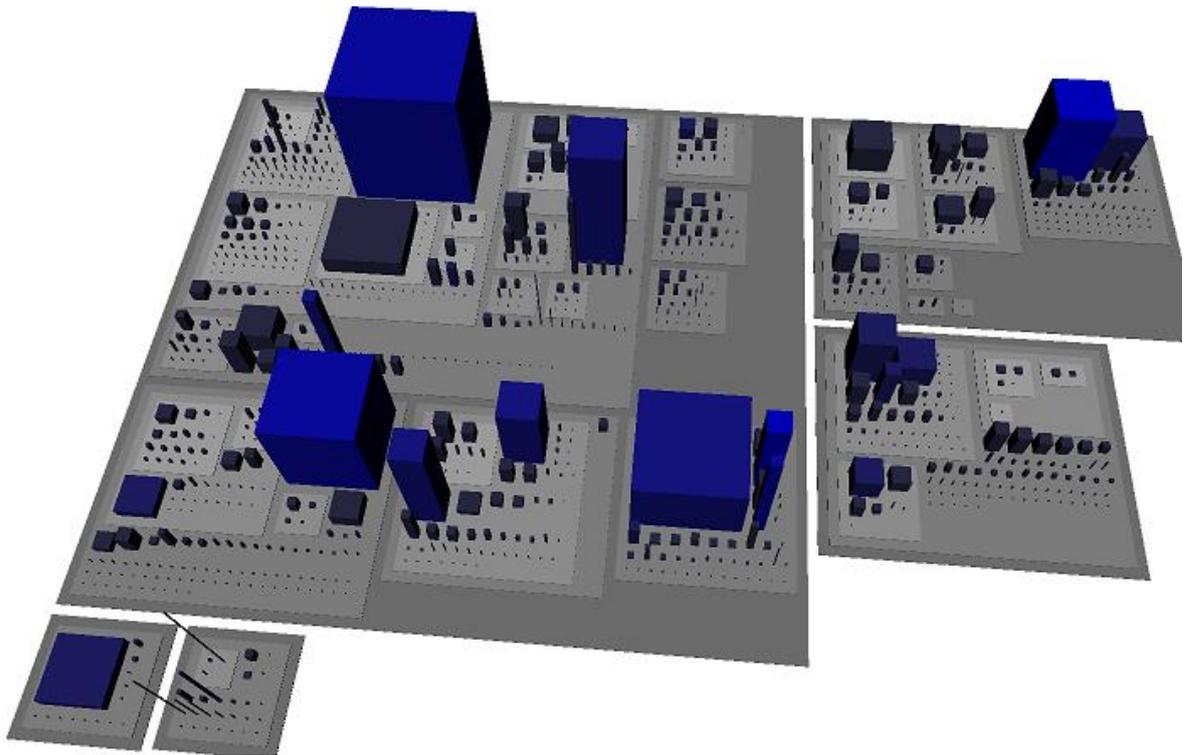


Abbildung D.4: Visualisierung nach der Implementierung von Farbgradienten

Quellen- und Literaturverzeichnis

Atkinson, C. & Kuhne, T., 2003. Model-driven Development: A Metamodeling Foundation. In *Software, IEEE*, 20(5). S. 36–41.

Balzert, H., 2009. *Lehrbuch der Softwaretechnik – Basiskonzepte und Requirements Engineering* 3. Aufl., Heidelberg: Spektrum Akademischer Verlag. ISBN: 978-3-8274-1705-3.

Bassil, S. & Keller, R.K., 2001. Software Visualization Tools: Survey And Analysis. In *Proceedings 9th International Workshop on Program Comprehension*. S. 7–17.

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

Ducasse, S., Anquetil, N., Bhatti, U., Hora, A.C., Laval, J., Girba, T., 2011. MSE And FAMIX 3.0: An Interexchange Format And Source Code Model Family. URL: <http://hal.inria.fr/hal-00646884/> [Zugegriffen 13.01.2015].

Graham, H., Yang, H.Y., Berrigan, R., 2004. A Solar System Metaphor For 3D Visualisation Of Object Oriented Software Metrics. In *Australasian Symposium On Information Visualisation*. S. 53-59.

Maletic, J.I., Leigh, J., Marcus, A., Dunlap, G., 2001. Visualizing Object-Oriented Software In Virtual Reality. In *Proceedings of the 9th International Workshop on Program Comprehension*. S. 26-35.

Müller, R., Kovacs, P., Schilbach, J., Eisenecker, U.W., 2011. Generative Software Visualization: Automatic Generation Of User-Specific Visualizations. In *International Workshop On Digital Engineering*, S. 45–49.

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

Müller, R. & Zeckzer, D., 2015. The Recursive Disk Metaphor: A Glyph-based Approach For Software Visualization. In *Proceedings Of The 6th International Conference On Visualization Theory And Applications*.

o. V., 2015. What Is X3D - Web3D Consortium. URL: <http://www.web3d.org/x3d/what-x3d> [Zugegriffen 14.01.2015].

o. V., 2014a. Xtend User Guide. URL: „[http://www.eclipse.org/xtend/documentation/2.7.0/Xtend User Guide.pdf](http://www.eclipse.org/xtend/documentation/2.7.0/Xtend%20User%20Guide.pdf)“ [Zugegriffen 14.01.2015].

o. V., 2014b. Xtext Documentation. URL: “[https://eclipse.org/Xtext/documentation/2.7.0/Xtext Documentation.pdf](https://eclipse.org/Xtext/documentation/2.7.0/Xtext%20Documentation.pdf)“ [Zugegriffen 14.01.2015].

Panas, T., Epperly, T., Quinlan, D., Sæbjørnsen, A., Vuduc, R., 2007. Communicating Software Architecture Using A Unified Single-View Visualization. In *Proceedings Of The 12th IEEE International Conference on Engineering Complex Computer Systems*. IEEE, S. 217–228.

Rekimoto, J. & Green, M., 1993. The Information Cube: Using Transparency in 3D Information Visualization. In *Proceedings Of The Third Annual Workshop On Information Technologies And Systems*, S. 125-132.

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

Shneiderman, B., 1992. Tree Visualization With Tree-maps: 2-d Space-filling Approach. In *ACM Transactions On Graphics*, 11(1), S. 92–99.

Stahl, T., Völter, M., Efftinge, S., Haase, A., 2007. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management* 2. Aufl., Heidelberg: dpunkt.verlag. ISBN: 978-3-89864-448-8.

Wettel, R., 2010. *Software Systems As Cities*. Dissertation. Lugano: Università Della Svizzera Italiana. URL: <http://www.inf.usi.ch/phd/wettel/download.php?f=Wettel10b-PhDThesis.pdf>.

Ehrenwörtliche Erklärung

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

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

Denise Zilch

Leipzig, 3. Februar 2015